CS325: Analysis of Algorithms, Winter 2024

Practice Assignment 4

Due: Tue, 3/7/24

Homework Policy:

1. Students should work on practice assignments individually. Each student submits to CANVAS one set of *typeset* solutions in pdf format.
2. Practice assignments will be graded on effort alone and will not be returned. Solutions will be posted.
3. The goal of the assignments is for you to learn solving algorithmic problems. So, I recommend spending sufficient time thinking about problems individually before discussing them with your friends.
4. You are allowed to discuss the problems with others, and you are allowed to use other resources, but you *must* cite them. Also, you *must* write everything in your own words, copying verbatim is plagiarism.

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.

A graph is bipartite if its vertices can be divided into two disjoint sets L and R such that every edge connects a vertex in L to one in R. Trees are connected graphs without cycles. The proof can proceed through a simple induction:

1. Choose any node of the tree as the root and place it in set L.
2. Put all the children (direct neighbors) of this root node into set R.
3. For each node in R, put their children (nodes not yet categorized) into set L, and so forth.
4. Since there are no cycles in a tree, this process will not create any conflicts, i.e., no node will be required to be in both sets L and R.
5. Hence, the nodes of the tree can be divided into two sets such that each node is only connected to nodes in the other set, satisfying the definition of 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.

If a graph G is bipartite, then every cycle in G has an even number of edges. The proof is as follows:

Suppose G is bipartite, and there exists an odd-length cycle C. Consider an edge e in C. Since C is an odd-length cycle, removing e still leaves a cycle, and that cycle is also odd-length. However, this contradicts the assumption that G is bipartite because after removing e, every node in C should be in the same set (either L or R), not alternating between L and R.

Therefore, if G is bipartite, every cycle in G has an even number of edges.

Conversely, if every cycle in G has an even number of edges, then G is bipartite. The proof is as follows:

Assume that every cycle in G has an even number of edges. We can use depth-first search (DFS) to partition the nodes into L and R. Start from any node, place it in L, and recursively assign its adjacent nodes to the opposite set. If we encounter a node already assigned to L during DFS, assign its adjacent nodes to R, and vice versa. This ensures that every edge has endpoints in different sets, making G bipartite.

(c) Describe and analyze an efficient algorithm that determines whether a given undirected graph is bipartite.Try to work on this problem as you read about graph search algorithms.

We can use either BFS or DFS to determine whether a graph is bipartite. Starting from any node, place it in L, and then recursively perform BFS or DFS on its neighbors. If we encounter a node already assigned to L, assign its neighbors to R, and vice versa. If during traversal we find a node whose neighbors are already assigned to the same set, the graph is not bipartite. Otherwise, it is bipartite.

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.

1.
    - Create a priority queue (min heap) containing nodes for each symbol **c_i** with frequency **f[i]**.
    - Initialize each node as a single symbol with its corresponding frequency.
    - Assign a unique code (initially empty) to each symbol.
    - While there are more than one node in the priority queue:
        - Extract the two nodes with the lowest frequencies (let's call them **N1** and **N2**).
        - Create a new internal node **N** with frequency equal to the sum of **N1** and **N2**.
        - Set **N1** as the left child of **N** and **N2** as the middle child.
        - Insert **N** back into the priority queue.
    - The last remaining node in the queue is the root of the ternary Huffman tree.
    - Traverse the ternary Huffman tree:
        - For each leaf node (symbol **c_i**):
            - Start from the leaf and move towards the root.
            - If the path to the root goes left, append '0' to the code.
            - If the path goes middle, append '1'.
            - If the path goes right, append '2'.
            - The resulting code for **c_i** is the reverse of the path taken.
2. **Proof**
    - We need to show that the ternary Huffman code minimizes the total encoded length.
    - Consider any other ternary prefix-free code. We'll prove that its total length is greater than or equal to the length of the Huffman code.
    - Let **T** be the other ternary tree corresponding to the given frequencies.

- If **T** is not full (i.e., has a node with only one child), we can reduce its depth by merging nodes, resulting in a tree with lower cost. Thus, **T** cannot be optimal.
- If **T** is full, we can show that the Huffman code is optimal by induction on the number of symbols

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

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

exists an algorithm that can determine whether an input belongs to the language of the problem or not

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

There are decision problems for which no efficient algorithm exists,

(c) $P = NP$. Unknown

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

If any NP-complete problem can be solved in polynomial time, then all problems in NP would also be solvable in polynomial time, which would imply P = NP

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

If there exists a polynomial-time reduction from problem A to CIRCUIT SAT, then A is also NP-hard

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

If A can be solved in polynomial time, it means there exists a polynomial-time algorithm to verify

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

Being in NP does not automatically make a problem NP-complete

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

While it is suspected that NP-complete problems cannot be solved in polynomial time, this has not been proven.

(i)