Report for CSC3100 Assignment 3

Problem 1: Tree Problem

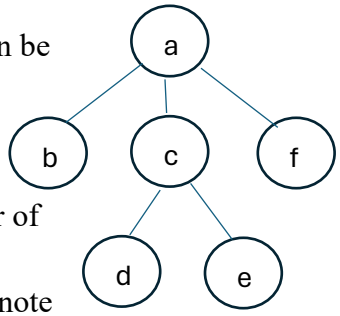
Language: Java

Possible Solution:

Construct a tree and search over all possible solutions. However, this is impossible. Enumeration is one of the most time-consuming algorithms which takes exponential time complexity. It must work bad due to for a single node having n children, there are A_n^n permutations only for these child nodes. The number of permutation would multiply in a tree having multiple branches, making the calculation extreme.

Observations:

1. The DFS algorithm uses a data structure stack. Thus, we can use stack to check if the output follows a DFS order. Recursion that has a stack logic can also be used. Here I choose a real stack for having solved similar problem;
2. To carry out DFS, we need to construct a tree. However, in this problem, we needn't construct a tree structure. It's enough that we only use some information of the tree including only the parent of each node and children each node has;
3. In a DFS order array, for a node being the root of a subtree of with the size n, the following (n-1) numbers must be the decadences of the node. The structure can be represented (for instance) as $\{a\_(b, c\_(d, e), f)\}$, which is shown:
4. Also note that in the example array, the nodes in the same parameter can be arbitrarily reorganized.



Algorithm:

1. For basic setting, use two integer arrays separately containing the father of each node and the number of the direct children of each node. We can obtain both from the input data that gives the father of each node. Also note that the number of children information is an initialization, which will be changed in the following steps;
2. We directly check the given array whether it follows a DFS order. First add 1 into the stack as initialization;
3. For the following nodes, check if its father is the node on the top of the stack. If not, the given array does not follow the DFS order, and the code returns false immediately. Otherwise, reduce the number of children of the node on the top of the stack by 1. Then, we check if the newly input node is a leaf by checking whether the number of children of the node is 0. If so, do not add the node into the stack and iteratively check the top node of the stack whether its number of children is 0, and pop it if it is. Otherwise, add the node into the stack;

4. When all the numbers in the 3rd input row are checked and the code does not return false, the given array follows a DFS order. There are properties that the stack must be empty, and the number of children array should only contain 0 after all numbers in the input array were checked, which may be used for further algorithms.

Complexity Analysis:

The time complexity of the algorithm for each test case is O(n). All the iterative operations were down at the same time of iteratively inputting numbers with constant time complexity. Thus, the time complexity equals to time complexity of reading the input, where (1+(n-1)+n) input are read. Therefore, the total time complexity is O(n).

The space complexity of the algorithm is also O(n). There are only 3 data structures in each small test case: 2 arrays of size *n* and 1 stack of maximum size *n* (only in linear tree case). Thus, the algorithm has an O(n) space complexity.

Advantages of the Algorithms:

1. O(n) space and time complexity, showing the algorithm is one of the most efficient algorithms;
2. Concise tree simulation that avoids using nodes and tree class to construct a real tree. Furthermore, the adjacent list is also not used in the simulated tree, which saves large space;
3. Fully observe the stack essence of DFS algorithm. Making the coding intuitive;
4. However, the algorithm can be further improved by using recursion instead of using a stack, which while safe an extra memory space for the stack. However, the code would be slightly complicated to write.

Problem 2: Gift

Language: Java

Possible Solution:

No other possible solutions come up to me except finding the minimum spanning tree.

Observations:

1. Originally, we could buy any gift with a constant given price *w*. Thus, we could set the cost to all gifts as 3 initially, which will be updated;
2. If buy one gift, then we could use all discounts attached to buying this gift for buying other gifts, i.e., the cost of buying other gifts is possible to be updated if lower;
3. The graph is undirected, no negative weight (no sending with extra cash);
4. Also note that A_{ij} refers to “no discount for gift *j* after buying gift *i*” but not “buy *i* get *j* free” (which should be noted but not).

Algorithm:

1. The key idea of the algorithm is using Prim's Algorithm, in which each node (gift) is only visited (bought) once with the lowest current cost and can form a minimum spanning tree. Besides, the whole problem is an undirected graph and there is no negative circle, which is appropriate for Prim's Algorithm;
2. For the initial setting, we need 2 integer arrays of size *n* respectively representing whether each gift node is visited or not (actually Boolean array), and the final cost of getting each gift, together with a 2-d integer array containing all discount prices given in the table. Besides, we choose iteratively going over all nodes to find the smallest instead using a priority queue as taught in lectures;
3. In each iteration, we visit a node and call it as current node (take it as given, the first current node is gift 1). We update all adjacent nodes of the current node and choose the next node with the lowest cost after the updating;
4. The adjacent nodes to be updated should follow the rules that is not visited (haven't been "current node"), and the new cost is smaller than the original cost (otherwise not necessary to update);
5. The node to be chosen as the next "current node" must have the lowest cost among all nodes which haven't been visited;
6. The iteration should end when all nodes are visited (all gifts bought), which could be realized by simply stopping iteration after *n* (number of all nodes/gifts) times.

Complexity Analysis:

The time complexity of the algorithm is simply $O(n^2)$. The algorithm is of $O(n^2)$ time complexity because there is only one function *prim()* in the main function. The function *prim()* has 2 parallel for-loops of *n* times in a for-loop of *n* times. Therefore, the time complexity of the function *prim()*, aka the algorithm part in the main function, is $O(n^2)$. Besides, the input has an $O(n^2)$ time complexity since there are $n+2$ numbers of input in total and output part has an O(n) time complexity since the result is the summation of an integer array of size *n*.

The space complexity of the algorithm is $O(n^2)$. The data structure defined with the largest theoretical size is the 2-d integer array showing all discount prices of size n^2.

Advantages of the Algorithm:

1. By using for-loop instead of priority queue, the code becomes simpler to be implemented and may be faster in complicated cases that edges are much more than nodes due to not using complicated data structures, though in simple cases it could be slow;
2. Relatively low time complexity and space complexity. Both are with time complexity $O(n^2)$, which is enough to solve the problem.