Exercise 4

Introduction

This exercise tells us to design and implement a Dynamic Programming algorithm that solves the software assignment problem. programming is an algorithm design technique based on the division of the problem into subproblems and the use of optimal substructures. In fact, it is applicable to all the problems that have these two properties:

- Optimal substructure, i.e. the solution of the problem can be obtained by the combination of optimal solutions to its sub-problems
- *Overlapping* sub-problems, i.e. means that the space of sub-problems must be small, and the algorithm solves each sub-problem only once.

This can be done using one of the following two approaches:

- *Top-Down*: the problem is broken into sub-problems, these are solved, and the solution is remembered, in case it is necessary to solve them again. It is a combination of recursion and memorization.
- *Bottom-up*: all the subproblems that may be necessary are solved first and then they are used to build the solution to wider problems.

Solution Chosen

In our solution we use a dictionary called “tab” to save the solutions of the subproblems, in order to do not calculate them over and over and easily combine them to obtain the solution of a bigger problem. This dictionary is composed of a key and a value which represent respectively a vertex and the solution found up to that vertex. In the function “call_func(g,u)”, that takes in input the tree graph g and the root vertex u, we create the dictionary, that initially is empty, and after that we call the function that contains the dynamic programming algorithm.

The first step in dynamic programming is the definition of sub-problems. In our case the minimum sub-problem is an edge that has as one of endpoints a leaf: in this case we decided to never install the software on a leaf, preferring instead the other endpoint because it surely has a number of incident edges greater than the leaf.

The function “func(g, u, tab, parent)” is the function that contains the dynamic programming algorithm. It takes in input the graph, a vertex (initially is the root), the dictionary tab (explained above) and parent vertex of the current one. We first check if the vertex is in the dictionary, i.e. it has been already calculated the solution for that vertex, in order to do not calculate it over and over.

For each vertex, the optimal solution can be one of this two cases:

- The solution contains the vertex and the sum of optimal sub-solutions of its children
- The solution does not contain the vertex. In this case it is formed by its children (they must be included in the solution to respect the requirements of the problem) and the sum of optimal sub-solutions of its children

For each vertex we calculate both the solutions, starting from the optimal sub-solutions of its children already calculated, and we save in two list called respectively *incl_list* and *excl_list*. Finally, we choose the solution that has the minimum number of vertices inside.

The algorithm, starting from the root of the tree, recursively visits the tree until it reaches a leaf (i.e. the minimum sub-problem), so it starts to combine all the solutions in a bottom-up approach starting from this simple base case and saving them in the dictionary *tab.* At the end, the solution of the entire problem can be found in the dictionary using as key the root of the tree.

Computational complexity

The algorithm performs very simple operation, i.e. evaluate the two possible solutions starting from the sub-solutions already calculated and choose the list with the minimum length (i.e. the solution with the minimum number of vertices) and store it in the dictionary. All this operations takes constant time to be performed. Since the algorithm visits all the vertices of the tree only once, it overall takes a time linear with the number of the vertices, O(n).