Practice 2

Guillermo Girón García

15 - 11 - 2019

Abstract

In this practice I'm going to create an algorithm for the Steiner Tree Problem. This problem consists of a graph with n total nodes, out of which m are optional to construct a generator tree. I will study how long it takes depending on the number of optional nodes.

1 Methodology

Before we get into the main function, we need a few assistants:

- Graph class
  - I used a class to represent graphs. This class uses a vector of vectors to represent the cost matrix, giving access to said matrix. It also provides the costeTotal function to quickly calculate the total cost of the graph, as well as general features like the movement, copy and assignment operatos.
- Apo class
  - This is a generic class used to store elements in a partially ordered tree. It functions like a heap with preference: it gives access to the element at the top, which always will be the highest valued element. In our case, it will be used by prim's algorithm to store edges, and the smallest cost edge will always be at the top. Its functions to add and remove an element are of order O(log a), 'a' being the number of elements.
- Prim function
  - It implements Prim's algorithm, returning the given graph's smallest generator tree. Its order is O(a*log n), 'a' being the number of edges and 'n' the number of vertices. In the worst case, the graph will be complete, so the number of edges will be a = n*(n-1)/2, so the order will be O(n^2*logn).
- Submatrix function

  - This function returns a subgraph of a given graph. Said subgraph's vertices will be the ones pointed by a vector of indexes. Its order is O(m), 'm' being the number of vertices of the final subgraph.

- Binary number class

  - I used a class to represent binary numbers. Said number is initialized at zero with a given number of digits. It gives access to every digit as well as a function to increment the binary number by one, whose time complexity is O(n) where 'n' is the number of digits.

Note: The apo and graph classes, as well as prim's algorithm have been taken and adapted from EDNL' sources.

With all of this said, it is time to get into the main function: the Steiner function (called Treeparse in the cpp file).

This function receives the given graph and an integer NBase. Said integer says how many of the graph's vertices are mandatory, and it is assumed that they are the first few vertices of the graph's matrix.

It starts simply by calculating the generator tree with every optional vertex. Then, it creates a binary number with as many digits as optional vertices. It iterates increasing thevalue of the binary number by one every iteration, to go through every possible combination (except the "empty" combination, which is calculated later separatedly).

In each iteration it then creates a subgraph of the main graph, consisting of the mandatory vertices plus the optional vertices indicated by the binary number. Then, it calls prim's function with said subgraph to obtain its minimum generator tree, and checks whether it is the best one at the moment, storing it if true. Lastly it returns the last tree it stored (which is the best one found).

2 Results and discussions

Every time complexity discussion will always assume the worst case scenario.

The Steiner function iterates through every possible combination of optional elements, meaning that its biggest loop belongs to the order of O(2^m), where 'm' is the number of optional vertices.

However, in every iteration of this loop, the biggest operation is the call to prim's function with a subgraph of order O(k^2*logk), 'k' being the number of vertices of the subgraph in each iteration. K will start equal to NBase, increasing every few iterations all the way up to n. To simplify calculations, I will assume k to be equal to n in every iteration.

Lastly, the time complexity of the Steiner function would be:

$$O = 2^m * n^2 * log(n)$$

It belongs to the exponential order, which is not considered efficient. Figure 1 showcases the huge growth in time that the algorithm gets as the number of vertices increase.

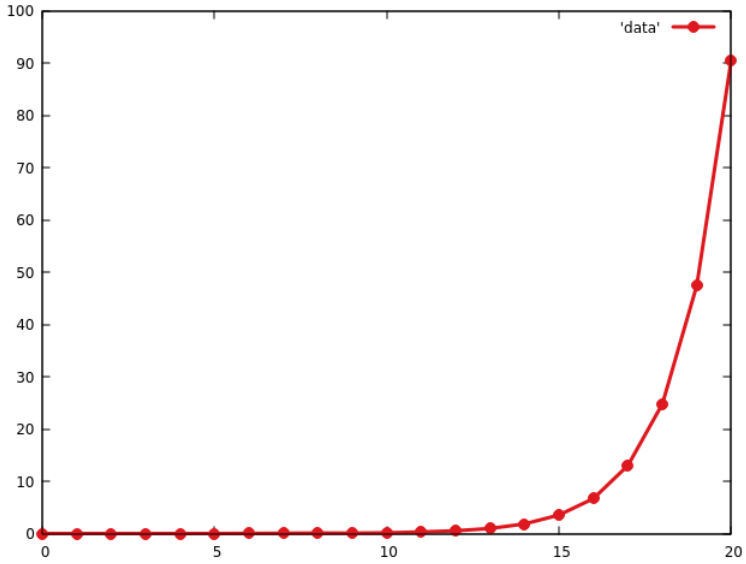

Figure 1: Growth of time (seconds) taken by the Steiner algorithm as the number of optional vertices increase. For this example, the total graph always has 30 vertices (The results of the figure were calculated by the test.cpp file)

3 Conclusions

We observe that, in fact, the Steiner Tree Problem is largely inefficient, and even for relatively small quantities of twenty to thirty optional vertices the execution time increases immensely. Albeit the actual algorithm of Steiner is inefficient, everything else from Prim's algorithm to the copy operation bewteen graphs is almost as efficiently implemented as it can possibly be within the known standards. This results in the total number vertices not playing a big role in the algorithm's total time, as it's the amount of optional vertices which causes the large amounts of combinations to check and is responsible of the exponential growth.

4 Reference

- EDNL material, theory and sources extracted from Campus Virtual.