Assignement 2

Iván Piña Arévalo
ivan.pinaarevalo@alum.uca.es

April 20, 2019

Abstract

In this practice, we will make a Steiner tree problem implementation. This problem is very known in the area of the computational complexity. Although several algorithms exist as today, none of them is able to solve the problem in polynomial time. Once the algorithmics is implemented, we will make a study of his complexity, analysing the worst case.

1 Introduction

First, we will speak about the 'Steiner tree problem'

The Steiner problem consists in, given a graph, to find the graph minimal expansion three with the possibility of use intermediate nodes.

To find a minimal expansion tree with algorithms whose temporary order is known, it is important to note:

- Prim: It was designed by Vojtech Jarnich in 1930, afterwards by Robert C. Prim. It has O(n^2) order
- Kruskal: It was published by Joseph Kruskal in 1965. It has an order, in the worse case of O(n^2).

In this analysis the Prim Algorithm will be used.

2 Methodology

All implementations have been made in C++

The personal computer which the time keeping has been made, has the following qualities:

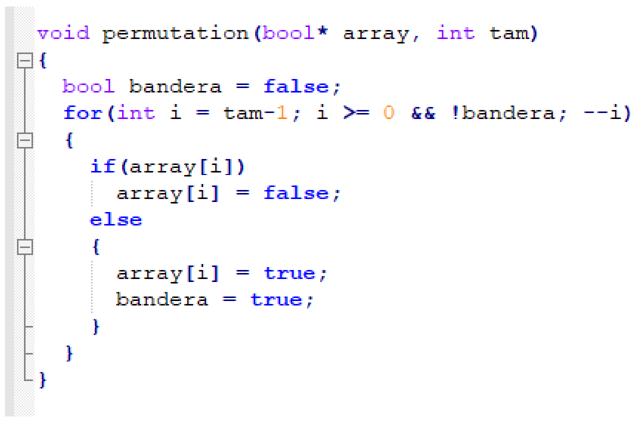
- Operating system: Manjaro (Arch)
- RAM memory: 8GB DDR4
- Processor: i7-7700K

2.1 Binary Combinations

The first practice challenge has been to implement an algorithm able to generate combinations. The structure of data selected has been a vector of boolean type, because it allows us to work with each position independently. As well as, to generate all possible combinations the binary system is eunough for us, being 1 true und 0 false.

I have decided to implement all neccesary instructions in a function, in order to increment the legibility of the main function. Also, as regards to efficiency I have used pointers, so that it is not neccesary that the function returns something in explicit way. (What it does is update the array that it receives as the first argument)

Next, the function is showed



```
void permutation(bool* array, int tam)
{
  bool bandera = false;
  for(int i = tam-1; i >= 0 && !bandera; --i)
  {
    if(array[i])
      array[i] = false;
    else
    {
      array[i] = true;
      bandera = true;
    }
  }
}
```

Figure 1: Function that generates the following combination

2.2 Implementing Prim's algorithm

In order to implement the algorithm, I have used the pseudo-code available in the practice statement.

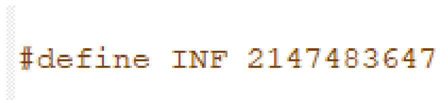
As a regard to the code readbility, I have designed a Prim class in which I have implemented the Prim algorithm. The object creation cost is irrelevant faced to his temporary analisys. So we'll avoid a slightly legible and overloaded main.

An important detail as regard to the implementation is the Prim Algorithm requisites. In order to be able to realize Prim over a graph, the graph must have the following features:

- Related.
- Un-guided.
- Labelled arcs

However, by calculating all possibilities including optional nodes, there will be situations in which the graph will not be related. An example of this situation happens when only the optional vertex number 7 is included

Thus, it has been necessary to include the INF macro, which takes the maxim value of an int, in particular 2.147.483.647.

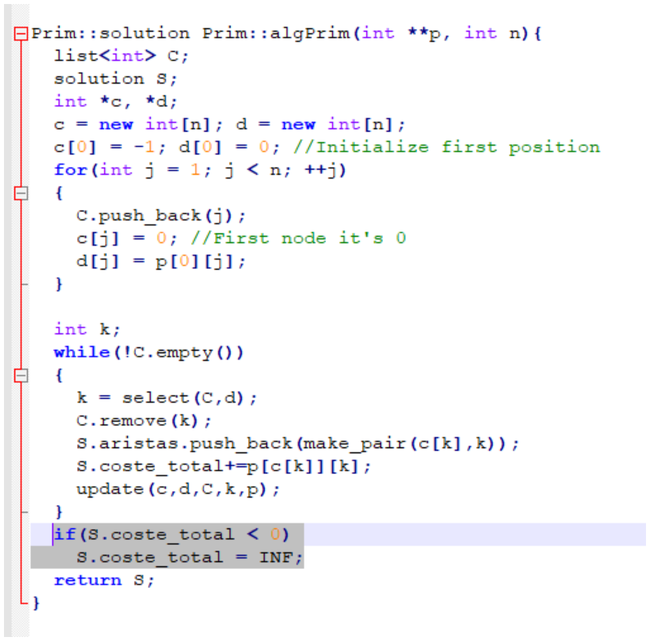


```
#define INF 2147483647
```

Figure 2: INF macro, defined in Prim.hpp

As well, the edges weight has not been made randomly. Such weight allows differenciate clearly the real result returnded when the graph is not related. If the algorithm returns a value equal to INF as the result is because the graph is not related

Another problem founded, has been that INF is the maximun value that a variable type INF can store. Due to this, in certain appeals the algorithm returned total negative cost. We have solved it checking the return value in the Prim algorithm main function. In the code it is named algPrim.



```
Prim::solution Prim::algPrim(int **p, int n){
  list<int> C;
  solution S;
  int *c, *d;
  c = new int[n]; d = new int[n];
  c[0] = -1; d[0] = 0; //Initialize first position
  for(int j = 1; j < n; ++j)
  {
    C.push_back(j);
    c[j] = 0; //First node it's 0
    d[j] = p[0][j];
  }

  int k;
  while(!C.empty())
  {
    k = select(C,d);
    C.remove(k);
    S.aristas.push_back(make_pair(c[k],k));
    S.coste_total+=p[c[k]][k];
    update(c,d,C,k,p);
  }
  if(S.coste_total < 0)
    S.coste_total = INF;
  return S;
}
```

Figure 3: Checking if the result is valid

2.3 Computing an ST

Costs matrix has been selected as behalf of the graph. As well, a submatrix which to dump rows and columns of the selected nodes is necessary.

Finally, once these three parts are been developped, we have merge all in a file named main.cpp

And now, we can observe the chosen graph to the implementation of this analysis

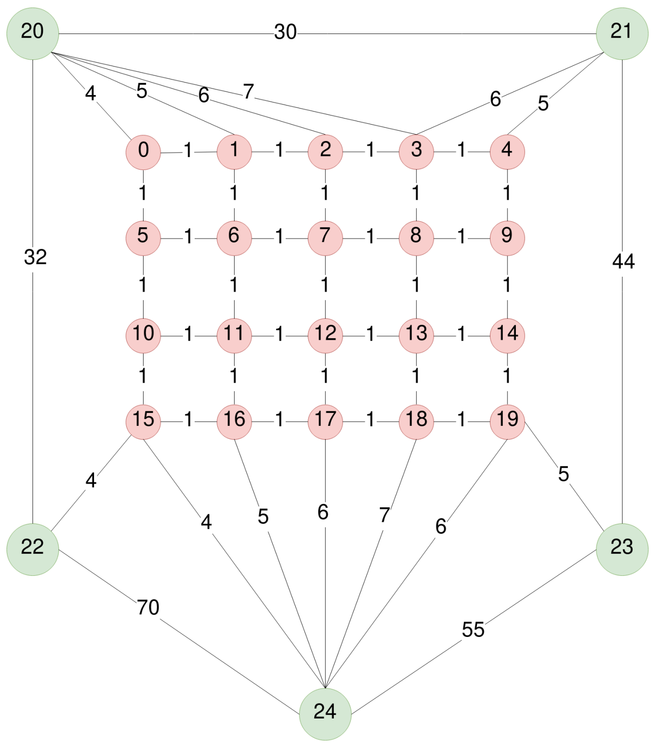
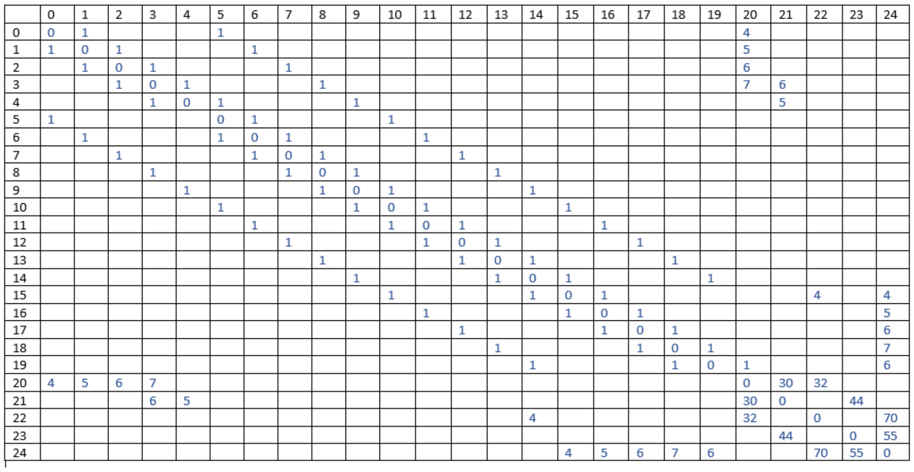


Figure 4: Graph used

As we can see, red and green nodes exist. Green nodes are obligatory. They are going to be included every time. On the other hand, red nodes are optional and they can be used or not



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
0	0	1				1															4				
1	1	0	1				1														5				
2		1	0	1				1													6				
3			1	0	1				1												7	6			
4				1	0	1				1												5			
5	1					0	1				1														
6		1				1	0	1				1													
7			1				1	0	1				1												
8				1				1	0	1				1											
9					1				1	0	1				1										
10						1				1	0	1				1									
11							1				1	0	1				1								
12								1				1	0	1				1							
13									1				1	0	1				1						
14										1				1	0	1				1					
15											1				1	0	1						4		4
16												1				1	0	1							5
17													1				1	0	1						6
18														1				1	0	1					7
19															1				1	0	1				6
20	4	5	6	7																	0	30	32		
21				6	5																30	0		44	
22															4						32		0		70
23																						44		0	55
24																4	5	6	7	6			70	55	0

Figure 5: Associated costs matrix

The positions of the matrix are empty because nodes are not associate, thus they have cost infinite.

2.4 Measurement of time

Lastly, it hould be note that the measurement of time spent has been been using 'clock' function, available in the ctime library

Also, in order to make easier the times counter, I have decided to put the counter inmediately before the loop while it encapsulates all possibilities. The counter stops at the loop output, once all calculations have been made. I've done it this way to eliminate the creation times of matrices, equal in all cases.

2.5 Call of the program

We have used the parameter argc from main to give the number of optional nodes. The graph has 20 optional nodes, so the number must be between 0 and 20. The correct way to call the program is

./program 3

This means the program will compute obligatory vertex + 3 optional vertex (17,18 and 19)

3 Result and discussions

Taking all execution times, we have obtained the results which we can see in the graphic:

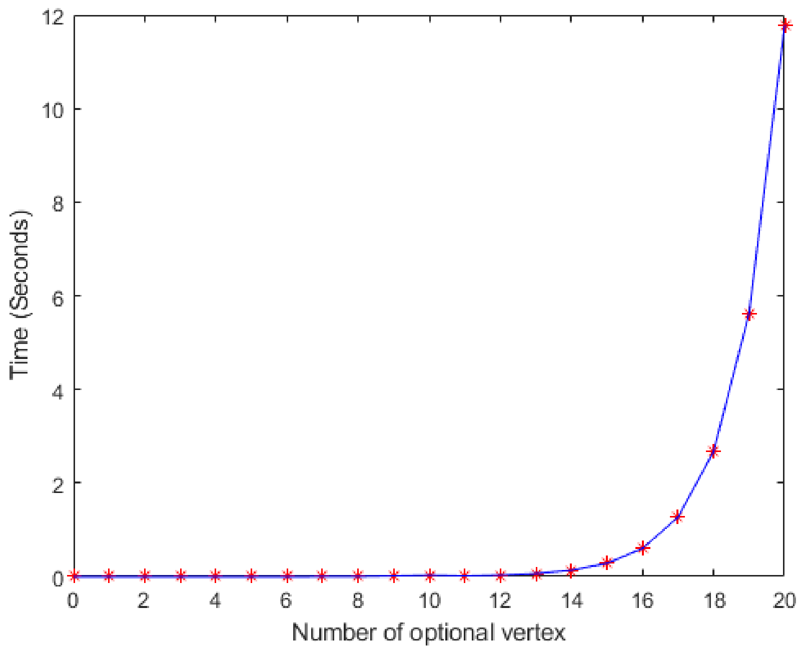


Figure 6: Time versus optional vertex

As we can see, the growth curve follows an exponential layout.

Reflecting on the algorithm behaviour, this result is what us were expecting. We know that Prim algorithm has an O(n^2) as a size of the entry, in this case the number of vertices of the graph

However, the algorithm target is to know the minimun cost of the spread tree with the possibility of using optional nodes. Now that we do not know what is the best node combination in advance, we must execute the Prim algorithm in each possible combination.

In mathematical term, been "x" the optional nodes number, we have tha $x!$ appeals to the algorithm are made.

As a result, the algorithm follows a factorial order. By maximun rule, we can discard the O(n^2) of the Prim algorithm (it is not relevant in comparison with $x!$)

A possible option would be to select an apex subset based on the smallest edget. In addition, the possibilities in which the graph is unconnected could be discard. In this way we would reduce the possibility number to compute.

4 Conclussions

Due to this algorithm order, it is only computable for a small set of entries broad enough small. While it is necessary to check all possibilities, this algorithm will remain non-computable.

Following Karp's list, made in 1972, we can observe that this problem is NP-complete. This affirmation agrees with the results obtained in this analysis

To conclude, in Richard M. Karp's words (36'53" in the movie on the virtual campus)

"If a problem is NP-complete, then we cannot expect to find a fast algorithm to solve it in all cases.

Instead, we must look for algorithms that handle most instances, or, in the case of optimization problems, algorithms that give near-optimal solutions"

5 References

- Slides of signature ”Diseño de Algoritmos”, chapter 1.
- P, NP, and NP-Completeness: The Basis of Computational Complexity. Oded Goldreich.
- NP-Complete Problems. Univeresity of California Berkeley. Richard M. Karp
- Introduction to the theory of computation, second edition. Michael Sipser.