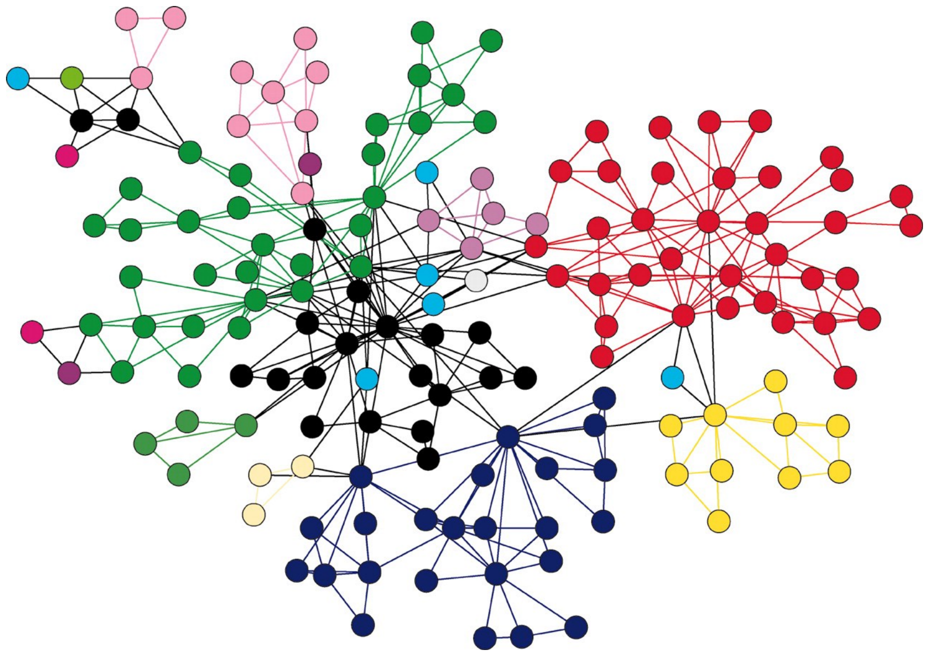
Second Data Structures Project

Graphs

Student: Thiago Chaves Monteiro de Melo
Registration Number : 180055127



1 Problem and solution

Several kind of problems in the computational area are originated from dealing with graphs an trying to extract information from them. A subset of this area is finding a Minimum Spannig Tree (*MST*) of a graph. The *MST* is defined as the minimum set of edges of a graph that connect all the vertices and have the minimum sum of weights. Here we have an example of a graph with it's *MST* highlighted.

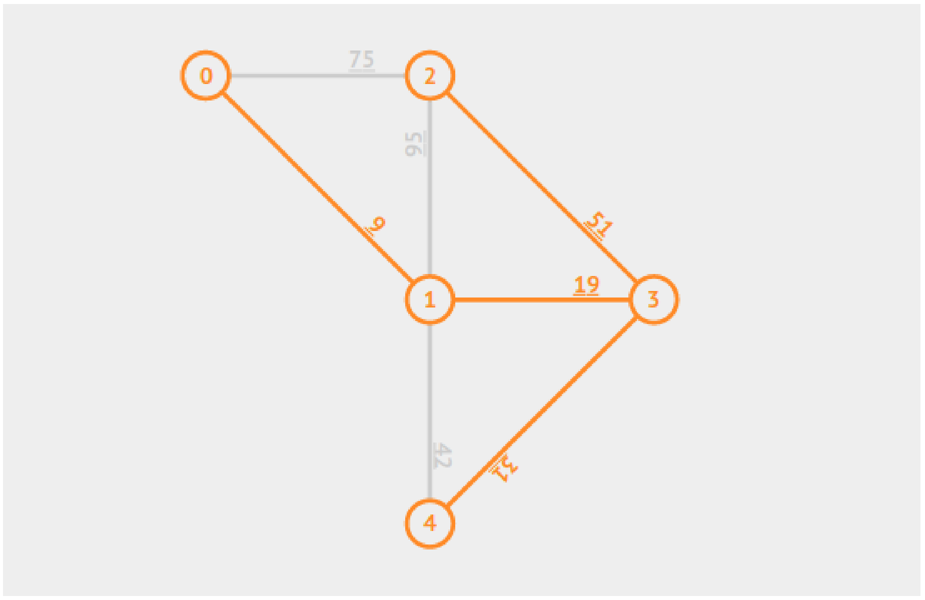


Figure 1: Example of an Graph with it's MST costing 110

To resolve that problem it was given some information about the graphs that would be used. All of them are planar graphs, in other words, they doesn't have any edges that are crossing each other, and they have to be undirected, that means that for any vertices connected $\{\alpha, \beta\}$, the connection $\alpha \to \beta$ is the same as $\beta \to \alpha$. Furthermore, all existing edges have a weight (this don't need to respect triangular inequality) and the graph is connected ($\forall$ vertex ν_1, $\exists$ a path to a vertex ν_2).

With this information it's possible to deduct some information:

- The *MST* must not have any cicle. *Proof by contradiction:* If it exists a cicle in the graph it means that there is at least one edge that can be deleted and the graph will continue to be connected, so, it's not a *MST* because this edge necessarily have a weight.

- Now, knowing that *MST* cannot have cicles, it can be affirmed that for a graph with ν vertices, his *MST* has necessarily $\nu - 1$ edges. *Proof by induction:* if you start a graph having only one node, there can't be any edges on it. Now, if you add one vertex, to keep the graph connected, an connection has to be added too. If we add one more vertex, a new connection has to be subjoint, and so on. As long as the graph remain acyclic, for a set of ν vertices there will be at most $\nu - 1$ edges.

Knowing the specifications of the given problem, the procedure taken was create a software with an implementation of an classical algorithm called *Prim's algorithm*. This will be explained with more details later on.

2 Input and output data

The program receives as input data from a file that begins with the number of vertices of the graph followed by an adjacency matrix of vertices that represent all edges and the cost of them. A line μ of this matrix contain all connections of the vertex ν_μ.

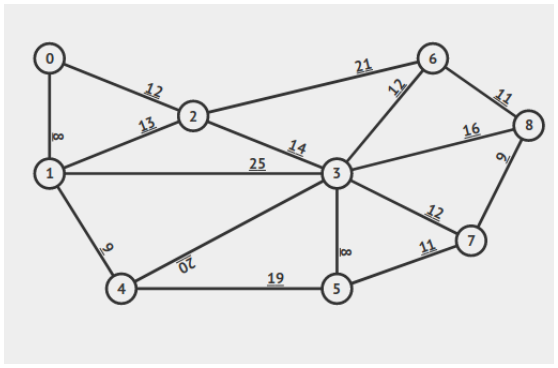
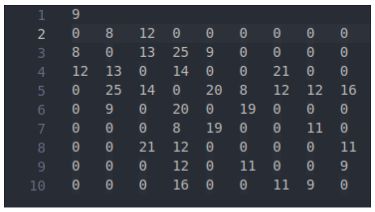


Figure 2: Visualization of input numbers



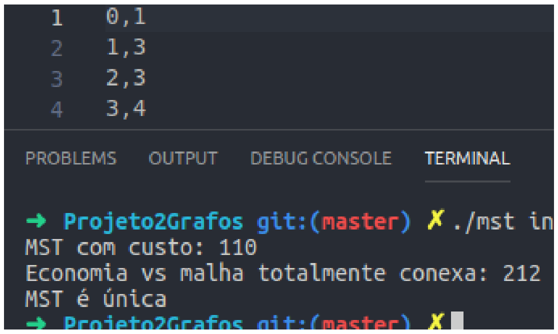
```
9
0   8   12  0   0   0   0   0   0
8   0   13  25  9   0   0   0   0
12  13  0   14  0   0   21  0   0
0   25  14  0   20  8   12  12  16
0   9   0   20  0   19  0   0   0
0   0   0   8   19  0   0   11  0
0   0   21  12  0   0   0   0   11
0   0   0   12  0   11  0   0   9
0   0   0   16  0   0   11  9   0
```

Figure 3: Example of an input File

It's valid to notice that we have an simetric matrix. This is caused by the fact that the graph is undirected.

The output of the program is separated in two diferent sections. The first one comes in a form of terminal message, printing out estatistics about: the cost of *MST*, the cost saved in comparison to the sum of costs of all edges and if exist more than one *MST*.

The second part of the output is written in a text file. It has to contain a list of all edges used in the *MST* found, orderd by the vertexes in each edge.


```
0,1
1,3
2,3
3,4
```

```
➜ Projeto2Grafos git:(master) ✗ ./mst in
MST com custo: 110
Economia vs malha totalmente conexa: 212
MST é única
➜ Projeto2Grafos git:(master) ✗
```

Figure 4: Output if the graph in Figure 1 was used as input

3 The Program

3.1 Program Modules

The program was subdivided into four modules:

- *Inout.c*: The functions to handle input and output data, and store that information in memory.
- *List.c*: Methods that act and operate on the list structures.
- *Grafo.c*: Functions to manipulate graphs and extract information from them.
- *main.c*: This is where all the modules above are combined to make the principal logic of the program.

Interdependence The only module that is independent alone is the *Lista.c* module. The *Grafo.c* module needs lists to function correctly and the input/output module depends on this two structures. Finally, but not less important, *main.c* depends on all this three modules previously cited.

3.2 Abstract Data Structure

The abstract data structure (ADT) and structures used to resolve the MST problem are listed bellow here:

- ADT List;
- Structure for $Vertices$;
- Structure for $Edges$;

The graph is formed by $Vertices \cup Edges$ structures. An *edge* contain an integer to store it's cost and an array with size of 2, to store the vertices that it is connecting. A vertex contain an integer that represent his *id* (an identifier) and a list of adjacency that carry pointers to edges that this vertice has. The functions that operates in this structures are:

- MST_Prim: The implementation of the *Prim's algorithm*;
- Path_Cost_List: Calculate the total path cost of a list of *Edges*;
- Path_Cost_Array: Calculate the total path cost of an array of *Edges*;
- Free_Graph: Free the space that was allocated in the vector of *vertices*;
- Order_Edge_Array: Receives an array of *edges* and sort the items in it according to the especifications of the problem;

The ADT of List is formed by a structure called List that have an integer representing it's length and pointers to structures *cel*'s. This structure *cel* is the one that actually holds the information that the list is storing. Besides that, this ADT has the following functions:

- CreateList: Allocates space for a new List;
- ListVazia: Check if a list has any information inside of it;
- InsertStart: Insert an element in the start of the list;
- InsertEnd: Insert an element in the end of a list;
- AccessElement: Return the *index* element of the list;
- RemoveStart: Remove an element from the start of the list;
- RemoveEnd: Remove an element from the end of the list;
- FreeList: Liberate the space allocated by the list and all the elements inside of it;

Lastly, the functions of module *Inout.c* that read input information, store it in memory, and print output details on a file and in terminal:

- Print_Output_File: Print the information in the format that it was especifed before;
- Read_Input_Graph: Read a file with a matrix of adjacency and returns an vector of *vertices* with all information of the file stored;
- Print_Graph: Takes a vector of *vertices* as input and print all vertices and what connection each one of them have;
- Print_Output_Terminal: Receives the information collected by the other functions and print the information necessary in terminal;

3.3 Prim's algorithm

In this section will be discussed about the most important function in the program, the *MST_Prim*. Prim's algorithm try to find best locally optimal solution and expand the area where it's acting. It is used find minimum spanning trees on a undirected graph, meaning that it's possible to use in this problem. The algorithm operates by storing the vertices that were already visited, and one at a time visiting every vertice with the best path possible, until every vertice is visited. Here we have a pseudocode to sumarize what was said.

```
1  Create a set $mstKeys$ to store the integer values;
2  Create a set $expVert$ to keep track of what vertices were already explored;
3  Initializate all items of $mstKeys$ with infinite values;
4  Initializate all items of $expVert$ with a "not explored" status;
5  $mstKeys$[Start Value] = 0;
6  while Any vertice is not explored do
7      Find a vertex $v$ that has minimum key from $mstKeys$ and was not explored;
8      Explore $v$;
9      Update the key values from every vertice adjacent of $v$ that has the connection cost smaller than the last key;
10 end
```

Algorithm 1: Prim's Algorithm

Even though this algorithm is pretty simple, it's very powerfull. Knowing that in every iteration of the loop in line 6 a new vertex from the graph is explored, it can be deducted that this loop will iterate η times, being η the number of vertices in the graph.

Example For the pourpose of demonstration of what the algorithm is doing, we can use the graph in image 1 as an example. First some vertex is picked as a start point to Initializate the algorithm. In this example the vertex 0 was chosen, so it's key value is 0, and the rest of the vertices are ∞:

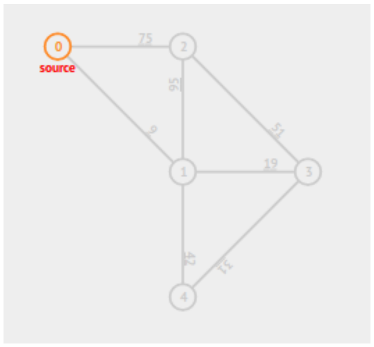


Figure 5: First step of Prim's algorithm

Now, the smaller key that it's vertex is not explored is 0. So this vertex is "explored" and all the keys of vertices adjacent of it are updated.

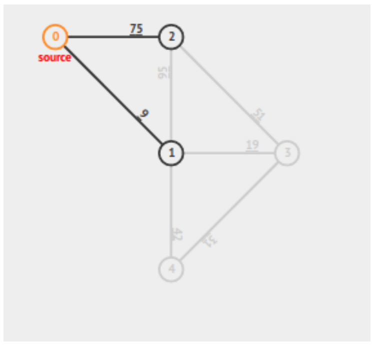


Figure 6: Updating keys of adjacent vertices

With this, the key of vertex 1 and 2 is now 9 and 72, respectively. So, again a vertex with smaller key that is not explored is chosen, and then the process repeat until all of the vertices are explored.

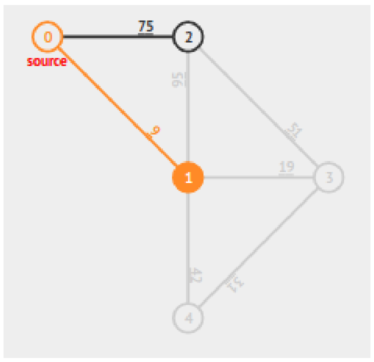


Figure 7: Vertex 1 is chosen

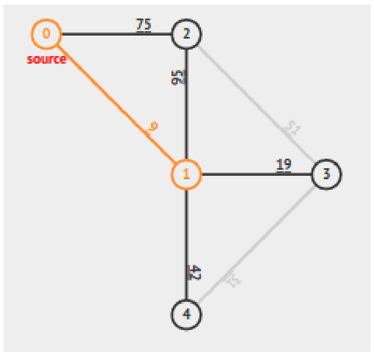


Figure 8: Keys of vertices adjacent to 1 are updated

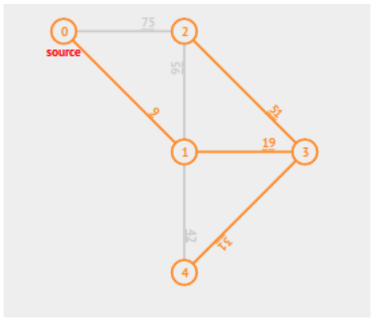


Figure 9: Final result after the process is repeated 6 times

3.4 The Complexity

To analyze the complexity of the program first some notations will be created to facilitate this process. When making reference to lists or arrays, it's length will be represented by the letter n. For graphs, the number of vertices in it will be represented by the symbol ν and the number of edges will be μ.

In module *List.c* we have some functions that operates in constant time, $\mathcal{O}(1)$, because the just make operations of assignment of values and allocation of memory. This functions are: CreateList, ListVazia, InsertStart, InsertEnd, RemoveStart, RemoveEnd. Still in this module, exist some functions that have complexity $\mathcal{O}(n)$: AccessElement and FreeList. AccessElement makes a sequencial search for a element in the list, so it's best case is $\Omega(1)$, but in the worst case, where it would have to search in the entire list for the element, is $\mathcal{O}(n)$. FreeList pass through entire list freeing the space allocated, so it's complexity is $\Theta(n)$, because the best and worst case are the same.

For the module *Grafo.c*, the functions that make operations just one time on the entire structure that is passed to them are: Path_Cost_List, Path_Cost_Array so its complexities are $\mathcal{O}(n)$.

The function Free_Graph, has not a straight foward analysis that the other functions had. For every vertice in the graph, the function free all data in list of adjacency of this vertice and in the end free the array of vertices. Knowing that the amount of edges that are stored are the double from the actually existing edges, because the characteristic of the graph being undirected, the total amount of free operations that this function will do is $2\mu + 1$, so the complexity of the functions depends linearly on the number of edges of the graph. Concluding, it's complexity is $\mathcal{O}(\mu)$.

The next function to be analised in this module is Order_Edge_Array. It takes as parameter an array of edges and sort its elements in descending order. The method used were an insertion sort, with a single diference that, because every edge has two elements to be sorted with the first being the smaller and having more weight on the sorting order. Here it's showed the algorithm used.

```
1  Let $\omega = \infty$;
2  Let an $ComparisonValue_n$ for an $\mu_n$ edge be equal to: $(\nu_{n1} \times \omega) + \nu_{n2}$;
3  for every edge $\mu$ from the parameter array do
4      if First vertex $\nu_1$ is greater than the second vertex $\nu_2$ of $\mu$ then
5          Swap $\nu_1$ and $\nu_2$;
6      end
7      while $ComparisonValue_n$ is smaller than $ComparisonValue_{n-1}$ do
8          Swap $\mu_n$ and $\mu_{n-1}$;
9      end
10 end
```

Algorithm 2: Insertion sort with two keys values

The lines 2 and 3 of Algorithm 2 are creating a method to compare diferent edges even if they have two diferent values. The if statment on line 4 is there to be sure that the second vertex of the edge is always greater than the first. Now, the loop "While" on line 7 have a worst case of doing the swaps n times, on the case of the last item being the smallest value of the set. Knowing that, complexity of this function, that operate n times and have another loop inside of it with a worst case of n, is $\mathcal{O}(n^2)$.

MST_Prim Before starting the main loop that actually "explore" the vertices of the graph, in line 2 and 3 of Algorithm 1 are the initilization some auxiliar variables. After noticing that this initialization needs to pass through all elements of the array of "Keys" and "Exploration" it can be concluded that this part alone of the algorithm have complexity $\mathcal{O}(\nu)$.

Now, entering in the loop of line 6 that, as discussed before occurs ν times, it is separeted in three main tasks. The first of them, that is searching for the minimum key, has complexity $\mathcal{O}(\nu)$ because the elements are stored in an array and to be sure that an element is the smallest, every element of the array needs to be compared to it. The second tasks, exploring a vertex, has complexity $\mathcal{O}(1)$ this is only setting a value in the array of "explored" vertices. The last part, that is deciding if the key value have to be updated after comparing it to the edge cost of vertices around the vertex that is being explored, have some a inductive way of analizing. It can't be known how many connection every vertex has, but, knowing that this part will be executed for every vertex of the graph, it can be concluded that in overall there will be μ comparisons.

With this information, the total complexity of the algorithm is the sum of the partial complexities calculated before. So, we have:

$$\begin{aligned} Complexity(Prim'sAlgorithm) &= \mathcal{O}((2 \times \nu) + \nu \times \nu + \nu \times 1 + \mu) \\ &= \mathcal{O}(\nu^2 + \mu) \\ &= \mathcal{O}(\nu^2) \end{aligned}$$

It's possible to discart the μ part because in terms of complexity, ν^2 is expected to be much greater the μ.

In the last module, *Inout.c*, going from the simplest function to the most complex function in this module:

Print_Output_Terminal only print the parameters in terminal, so its complexity is $\mathcal{O}(1)$;

Print_Output_File $\mathcal{O}(\nu^2)$, because it have to order $\nu - 1$ edges, using Order_Edge_Array metioned before, and then print them on a output file;

Print_Graph That $\mathcal{O}(\nu \times \mu)$, because it has to print every vertex and connection of it;

Read_Input_File $\mathcal{O}(\nu^2)$, because it reads an square matrix of adjacency with ν^2 elements and store its data in memory;

4 Appraising the algorithm

5 Software execution

- First item
- Second item

1. One
2. Two

Function1 BBla bla

Function2 Blol bl

BLU	BLE
XU	XE
THU	THA