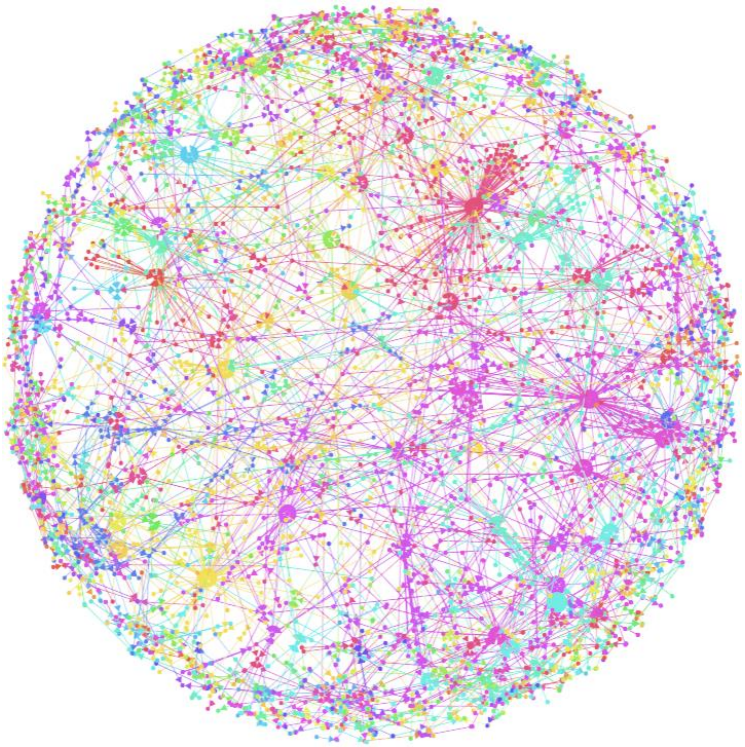
DATA STRUCTURE

GRAPH

MINIMAL SPANNING TREE



Gustavo Antonio Souza de Barros

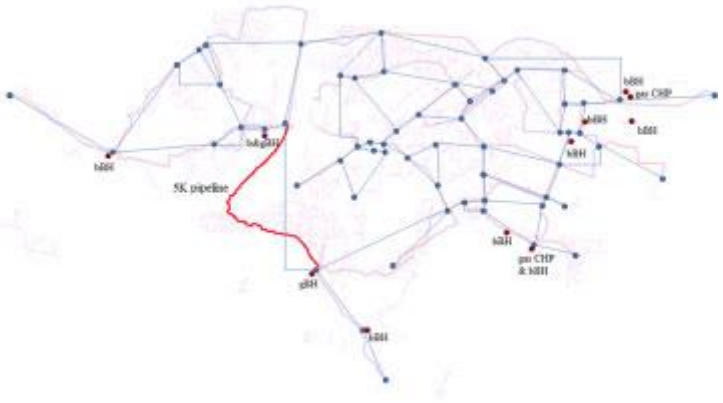
18/0064487

Brasília, 2018

1. Introduction

An electricity provider has a project to deliver light for small cities and villages around the country. The company wants to spend the minimum amount of money to make it happen. A study of the local is made, storing information about the buildings and the cost to position poles between two nearby houses.

To resolve this, it's necessary to interconnect all the houses while spending the less money possible. We can imagine a city being like a graph, a vertex corresponds to a house and an edge to the distance between then, the edge's weight simulates the cost to build a light pole there.



In order to find the cheapest path possible that connects all the houses, we need to use a minimum spanning tree algorithm.

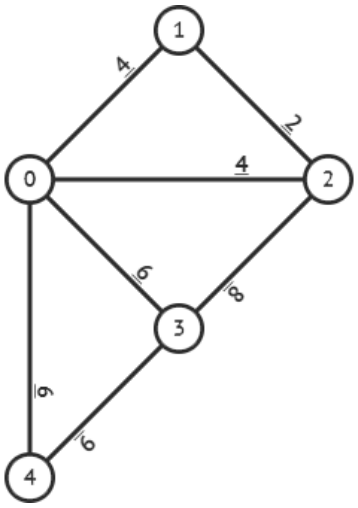
2. The Algorithm

To understand the implemented algorithm, we need to know what is a minimum spanning tree. An MST is a subset of edges that connects all vertices in an edge-weight graph. An MST has some features:

- It has the minimum possible total edge weight
- The size is determined by the Number of vertices minus one (V-1)
- Don't form any cycles
- A graph can have more than one MST

There are different ways to find an MST, but we will talk about one of the classical, the Prim's algorithm.

First of all, we need to read our graph, for that is used an adjacency list to store the vertex data. This data contains basically the edges that the vertex forms (origin and destination) and their weight. To understand how it works, here is an example:



First, we need to transform this graph in an adjacent matrix. Every row corresponds to a vertex, the columns will be the associated vertex. If there's an edge that connects them, we put it weight in the matrix, if there's no connection we put 0. The adjacent matrix for this graph is:

```
1    5
2    0    4    4    6    6
3    4    0    2    0    0
4    4    2    0    8    0
5    6    0    8    0    6
6    6    0    0    6    0
```

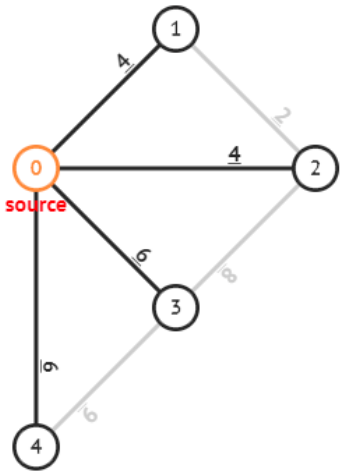
We transform this matrix to an adjacency list. For this, we will create a vertex array, each vertex has a list associated that contains the edges data. Example:

```
Vertex[0]: List{0,1 -> 0,2 -> 0,3 -> 0,4}
      edge: x,y (the weight is included in the struct)
      Next element in the list: ->
```

Now that we have the data stored, we need to use it to find the MST. The initial vertex is 0 by default. There are three auxiliary arrays in this program: Key, KeyPossibilities and AlreadyVisited, all with V elements, their index corresponds to a vertex. Key will store the less weighted path to the vertex. KeyPossibilities stores the number of different paths that we can choose to each vertex, this will be important later. AlreadyVisited says if the vertex was already reached by some MST edge. We have a auxiliary list too called anotherPaths, it will store if we can go through another key to find the MST, this will be important as well in the end of the program.

In the initial phase, the first one can be counted as visited, so we set alreadyVisited[START] = True, all the rest were not visited yet, so we set all False. The start vertex key is 0, because there was no cost to visit it, all another keys are set to infinite, because doing that we can find smaller paths easier.

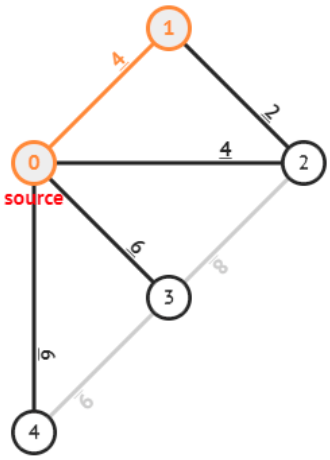
So now, the prim's algorithm has to appear, we look at the start vertex list and using the function set_keys we change the key's array to store the minimum weighted paths to the visible vertices.




```
key[vertex] = smaller weight found to go to
it
      key[1] = 4
      key[2] = 4
      key[3] = 6
      key[4] = 6
```

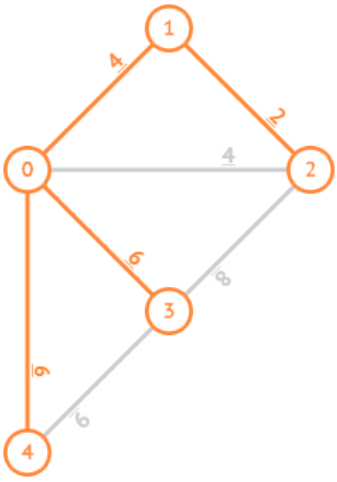
After that, we enter the function minimum_key, that finds the smaller weight path among the ones determined before in the key's array. In this case we would choose key[1], so the path 0,1 is already in the MST. In this case, notice that we could choose key[2] as well, so we store it in the anotherPaths list. With the information that 0,1 is part of the MST, we can set the keys that the vertex 1 provides.

We call set_keys now for vertex 1 and refresh the keys looking for a smaller key than the ones set before. The array of visited vertex protect us for going back and putting an edge two times, also has an additional feature that prevents us to change the keys for these visited ones, because we will need it in the final of the execution.



key[2] is now refreshed to 2, because it is smaller than the previous one.

Knowing that the MST has V-1 edges, we do a loop making this process this number of times and find this:



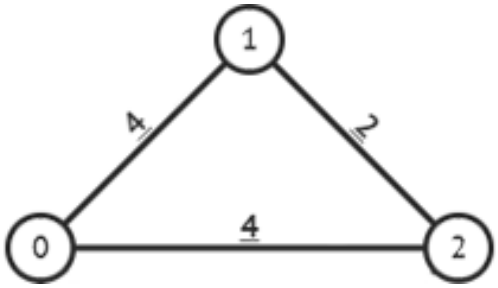
In the algorithm, the MST is stored in an array of edges with V size. But why it has V size even with the information that the MST has V-1 elements? With the logic implemented, the MST is stored with help of minimum key loop, this loop doesn't include the first vertex, the starter one, creating a "ghost" index. We can easy resolve this problem with the starter vertex being always 0, but how we are choosing to make it work with any starter value, was created a way to solve this. To resolve this, another loop removes the ghost index and stores the data in a array with V-1 elements.

So now we have three pendencies: the MST cost, the economized value in comparison to a all connected graph and if the MST is unique. The first one is simple, we can sum all the key values, keys_cost function do it for us. We already have the MST cost, we just need the Graph total cost, that can be made in the moment we read the matrix file, we sum all the values of the matrix and divide it by two. It's necessary to divide them because we read the same vertex two times in the matrix: a,b = b,a, after that we just make graphCost minus mstCost.

To find out that if the MST is unique is a more complicated problem, we test it based on our two functions, minimal_key and set_keys. Every time we set a Key, we test if there's another equal path that could be this key and increments in KeyPossibilites array.

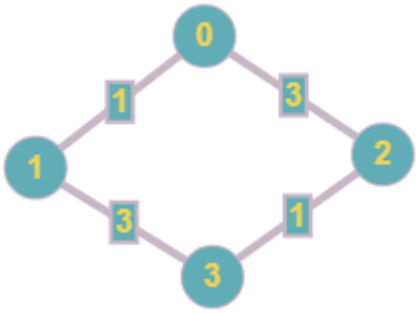
But that not necessary proves that the MST is not unique, because in the future we can set another key smaller than that one and don't use it. So when we choose a key to proceed in the function minimal_key, we test if it had more than one possibilities before, so if KeyPossibilities[ChosenVertex] > 1, the MST is not unique.

Even with these conditions, there's a problem, depending of us choose for the start vertex, it can generate problem, look at the case below:



0 has two equal minimum edges going to another vertex. We would choose key[1] and proceed, finding the MST in sequence. But this would not enter the conditions for not-uniqueness, and the result would say the MST is unique, even if is not. That's is fault of the bad choose of the start. When we choose 0 the values of key[1] and key[2] would be 4, in the moment that we set the keys, it would consider two paths differently, so the keys wouldn't be equal. In the moment we enter the choice of the keys, we would choose key[1] and see that KeyPossibilities were not incremented, giving us the output that is not unique. Even with one solution being choosing a better vertex to start, this would not be enough to another complicated cases. So was created a most complete algorithm, with another condition, allowing us to start where we choose and solving the problem for several cases.

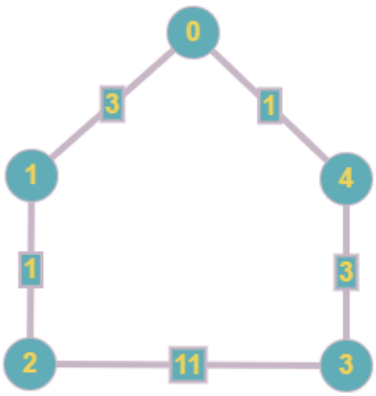
This moment is where the anotherPaths list shines, because with it, we can see if there's another path that could be used to find an MST. Look at this case now:



Starting at 0, we would choose the edge 0,1 because it is the smaller one, so now we have two choices, 0,2 or 1,3. Suppose that we choose 0,2 so we store 1,3 in the anotherPaths list, we complete the MST going through 2,3. Now we look at anotherPaths list:

```
anotherPaths{1,3}
```

How 1,3 is another path that could be part of the MST depending on our choices, we know that we can find another MST, so the one we find is not unique. But now look at this:



Starting at 0, we would choose 0,4. After that, we could choose 0,1 or 4,3. Suppose that we choose 0,1. So we store 4,3 in the anotherPaths list. Now we go through 1,2. For the final MST edge we would choose 4,3 and finish it. The MST for this graph is unique, but our anotherPaths list isn't empty, making our program to accuse that the MST is not unique. To solve this, we need to go through anotherPaths list removing the edges that are now part of the MST. How 4,3 is part of the MST, we would remove it from the anotherPaths list making it to be empty, so now our program is correct, accusing the MST as unique.

The final part of the program consists in adapting the MST for the output, first we swap, if necessary, the first number and the second, this is made in the function adjust_mst. It doesn't change nothing in the MST, because inverting the order of the edge's origin and destination changes nothing, it continues to be the same edge.

Next, we make an insertion sort, because mostly edges are already in ascending order, so we have a low complexity. First, we sort by the origin (first number), then sort by destination if the origins are equal.

With the output file set at the execution, using the print_output_file function we write the MST in it, and with the print_output function, we print the terminal output.

3. How to execute

To compile the program, you just need to write the command **make** in the terminal, it creates an executable file called mst.out. This will automatic execute the command:

```
gcc -g mst.c io.c list.c graph.c -o mst
```

The input and the output file need to be set in the execution command, first the input one containing the matrix of the graph, the second is the name of the file that you want to print the MST.

```
./mst input.txt output.txt
```

4. Routine List

Functions starting with _ are extra functions, used mainly for tests, so we are not going to make the complexity for them, because they don't impact in the real program.

- create_list:
  Creates a brand-new list and allocates a head node.
- insert_list:
  Inserts an edge in a list.
- _print_list:
  Prints information about the edges that are part of a list.
- free_list:
  Deallocates the memory used for a list and of it edges too.
- empty_list:
  Returns true if a list is empty and false is it's not.
- remove_list:
  Receives origin and destination as parameter and deletes the matching edge of a list.
- graph_mst:
  The most important function of the program, calculates the MST, is responsible for another features like modifying the MST cost and tests the conditions for uniqueness, returns the boolean that stores if the MST is unique or not.

- set_keys:

  Receives a vertex as parameter, uses it to refresh the keys that are reachable by it. Stores the possibilities for that key too.
- minimum_key:

  Selects the smaller key, and returns it vertex, this vertex will be used for set_keys in the next loop. If a key is chosen and has more than one possibility it already sets the unique boolean as false. If we have more than one minimum key, it chooses one and stores the rest in anotherPaths list.
- keys_cost:

  Sums all the keys, this is result in the MST cost.
- _print_graph_info:

  Prints information about the graph read in the input.
- read_input_file:

  Reads the input file containing the adjacency matrix of a graph and stores in an adjacency list.
- insertion_sort_mst:

  Insertion sort the MST edges by the origin first, and after by destination
- adjust_mst:

  For convention, the first number of the edge (origin) needs to be smaller than the second (destination), so it swaps the edge if necessary.
- print_output:

  Responsible for printing the output data: MST Cost, the total economized by it, and if it is unique or not.
- print_output_file:

  Prints the MST found in a separate file determined in the program's execution.
- _print_mst:

  Prints the MST found in the terminal.

5. Complexity

- create_list:

  O(1).
- insert_list:

  O(1).
- free_list:

  O(V). The worst case depends of the number of the edges that this list contains. A vertex can contain maximum V-1 edges.
- empty_list:

  O(1).
- remove_list:

  O(V). Runs all the list searching for the one that matches the parameters. Maximum V-1.
- graph_mst:

  $O(E+V^2)$. First, we have a loop that runs V times. After well call the functions set_keys and minimum_keys V-1 times. Another loop with V size. We call free_list and keys_cost one time. We will call the number of elements in a list M. The sum of all the elements of all the list equals 2E. So when we do a loop V*M we are doing this 2E times. O(V+V(M+V)+V+E+V). That can be simplified to $(E+V^2)$.
- set_keys:

  O(V). Runs all the list refreshing the keys. Maximum size of the list: V-1.
- minimum_key:

  O(V). There are two loops inside it, the two runs V times, resulting in an overall O(V+V) complexity, simplified to O(V).
- keys_cost:

  O(V). Runs all the key's array with V size.
- read_input_file:

  $O(V^2)$. Read all the adjacency matrix, which is quadratic.
- insertion_sort_mst:

  $O(V^2)$. Insertion sorting the MST results in a worst case $O((V-1)^2)$, which can be simplified by $O(V^2)$. But, with our MST being sometimes in ascending order already it can be executed in linear time or next to it in these cases. $\Omega(V)$.

- adjust_mst:
  O(V). Runs the MST with V-1 size.
- print_output:
  O(2).
- print_output_file:
  O(V). Runs all the MST array (V-1).

Overall complexity:

Functions called in main function(in order):

read_input_file - $O(V^2)$

graph_mst - $O(E+V^2)$.

adjust_mst – O(V)

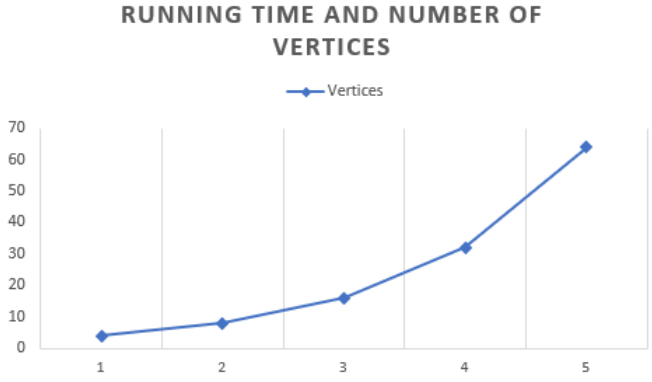
insertion_sort_mst – $O(V^2)$

print_output – O(2)

print_output_file – O(V)

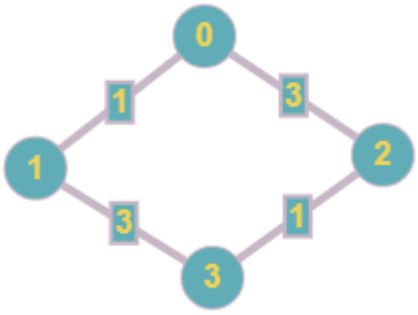
We have a loop deallocating the lists V times, the sum of all the lists elements equals 2E, so we are doing this 2E times. In the end we free the vertex list. All this represents O(E+V).

So, for the overall complexity we have $O(V^2+E+V^2+V+V^2+2+V+E+V)$, which can be simplified to $O(E+V^2)$. This result is expected, because the graph_mst function, which contributes to the costliest part of the program has that complexity aswell. With the graphic below, we can see how the running time grows depending of the number of vertices.



6. Test Cases

We will go deeper in two cases that were already quoted in this report. They were chosen to be in this report because they are simple and test the algorithm looking for logical errors. Let's start with this one (grafo7.txt – saida7.txt), we talked about it in Page 6.



By default, we will start at 0, so the initial arrays are:

```
Key[0] = 0, Key[1] = INF, Key[2] = INF, Key[3] = INF.

Visited[0] = 1, Visited[1] = 0, Visited[2] = 0, Visited[3] = 0.
```

Refreshing the keys for the 0 vertex will give us (Blue = Visited, Bold = Current):

```
Key[0] = 0, Key[1] = 1, Key[2] = 3, Key[3] = INF.
```

So, we choose the smaller one for an unvisited vertex: Key[1] and set it as visited, Visited[1] = 1. Doing this, we are choosing the 0,1 edge to be in the MST, so we store it in MST's array. So, now our current vertex is 1, we refresh the keys again with the information vertex 1 provides:

```
MST: 0,1

Key[0] = 0, Key[1] = 1, Key[2] = 3, Key[3] = 3.
```

Now look that we have two minimum keys possible (2 and 3), so we choose one, and store the another one in the anotherPaths list. Now we refresh the Keys with the current vertex, in this case we choose 3, so now 1,3 is in the MST.

```
Another Paths: 0,2

MST: 0,1 - 1,3

Key[0] = 0, Key[1] = 1, Key[2] = 1, Key[3] = 3.
```

The loop always runs V-1 times because that's the number of edges that the MST has. So now it's time to find the last edge, which is 2,3.

```
Another Paths: 0,2

MST: 0,1 – 1,3 – 3,2

Key[0] = 0, Key[1] = 1, Key[2] = 1, Key[3] = 3.
```

So the output file MST is:

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

In order to print the terminal output, we sum all the final keys to find the MST cost:

0 + 1 + 1 + 3 = 5.

In the moment we read the graph, the operation to find the graph total cost was already done, so to find the total economized we just do Graph Cost minus MST Cost :

GraphCost: 1 + 3 + 3 + 1 = 8

MstCost: 5

Economized: 3

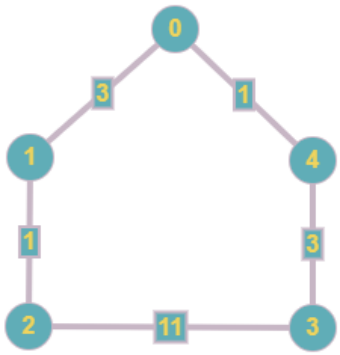
For uniqueness, we saw that every key had only one possibility to be set with the minimum value, so we skip the first condition, now our second condition is to see if anotherPaths list is empty, if it is the MST is unique:

Another Paths: 0,2. So the MST is not unique.

Terminal Output:

```
PS C:\Users\CLIENTE\Documents\Programação\C\ED\180064487_Projeto2 2.0> ./mst grafo7.txt saida7.txt
MST com custo: 5
Economia vs malha totalmente conexa: 3
MST nao eh unica
```

Now let's look case two (grafo8.txt – saida8.txt):



Starter Vertex = 0.

Start:

```
Another Paths:

MST:

Key[0] = 0, Key[1] = 3, Key[2] = INF, Key[3] = INF, Key[4] = 1
```

First interaction:

```
Another Paths:

MST: 0,1

Key[0] = 0, Key[1] = 3, Key[2] = 1, Key[3] = INF, Key[4] = 1
```

Second:

```
Another Paths: 1,2

MST: 0,1 – 0,4

Key[0] = 0, Key[1] = 3, Key[2] = 1, Key[3] = 3, Key[4] = 1
```

Third:

```
Another Paths: 1,2

MST: 0,1 – 0,4 – 4,3

Key[0] = 0, Key[1] = 3, Key[2] = 1, Key[3] = INF, Key[4] = 1
```

Fourth:

```
Another Paths: 1,2

MST: 0,1 – 0,4 – 4,3 – 1,2

Key[0] = 0, Key[1] = 3, Key[2] = 1, Key[3] = INF, Key[4] = 1
```

Output file MST:

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

Now, let's look at an interesting aspect of this case, our anotherPath list was modified, that happens because the edge that was there previously was chosen to be in the MST after, so we need to discard it. After all, it was not really another possible path that we just don't choose, it would be part of the MST eventually. How the anotherPath is adjusted now, we can make our final conclusions:

Terminal Output:

```
PS C:\Users\CLIENTE\Documents\Programação\C\ED\180064487_Projeto2 2.0> ./mst grafo8.txt saida8.txt
MST com custo: 8
Economia vs malha totalmente conexa: 11
MST eh unica
```

7. Conclusion

We conclude that the program does his job, he finds the MST and handle problem of the uniqueness in a detailed way. Even with some optimizations available, he runs in a quadratic time, which is not the ideal but it's not so bad as well. The test cases and they respective MSTs are available in the folder called **tests**, the pattern for the input and output is grafoY.txt saidaY.txt, with Y being the index of the test.