1INF06 Data Structures and Methodical Programming

Loaiza Vasquez, Manuel Alejandro
Oyarce Tocto, Elizabeth Patricia
Saras Rivera, André Edgardo

November 11, 2020

Pontificia Universidad Católica del Perú
Lima, Perú
manuel.loaiza@pucp.edu.pe
a20182778@pucp.edu.pe
andre.saras@pucp.edu.pe

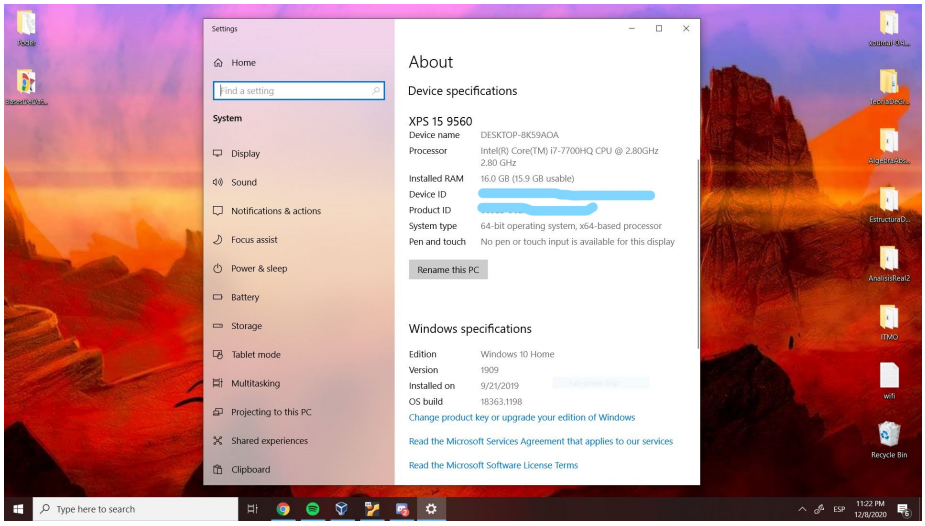
Fifth report of the course Data Structures and Methodical Programming taught at the Faculty of Science and Engineering at Pontificia Universidad Católica del Perú (PUCP) by Viktor Khlebnikov in the semester 2020-2.

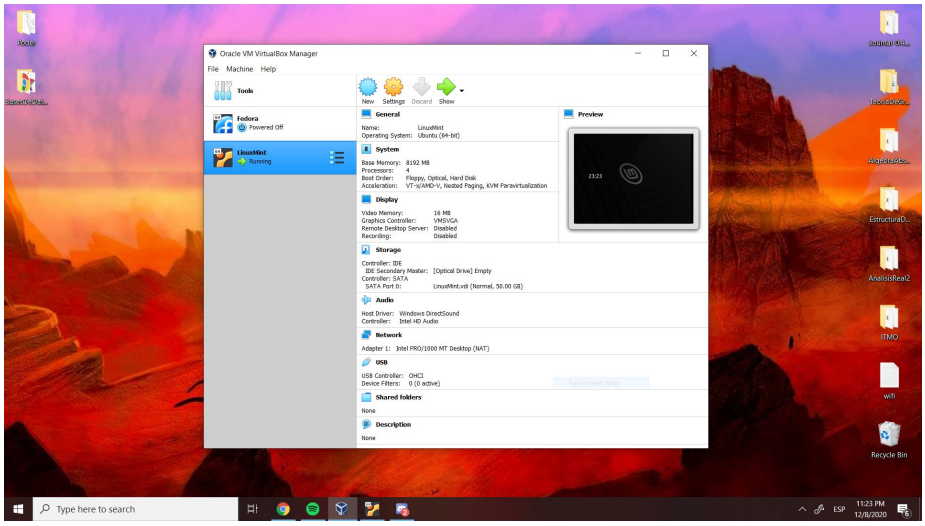
Contents

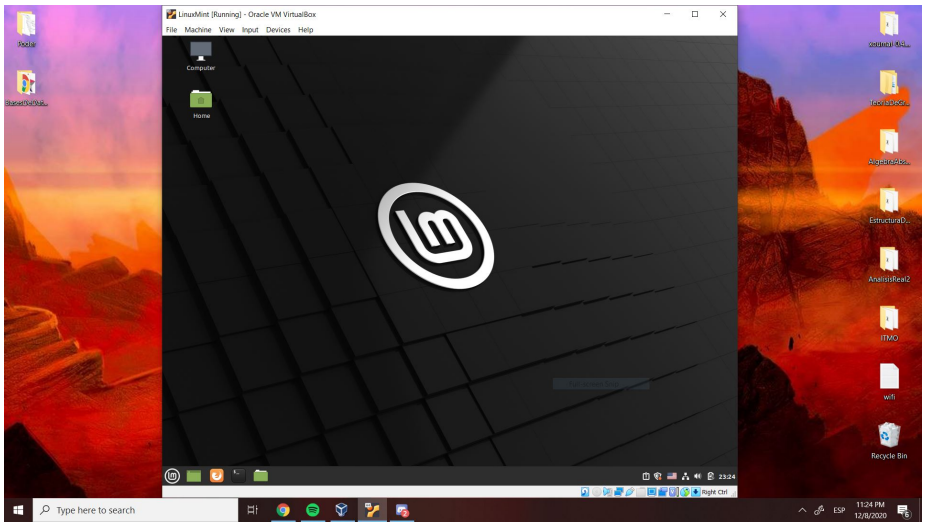
1 Loaiza Vasquez, Manuel Alejandro

This member has done 4.1, 4.2, 4.3.1, 4.3.3 and 4.3.4.

1.1 Resources and Oracle VM VirtualBox parameters



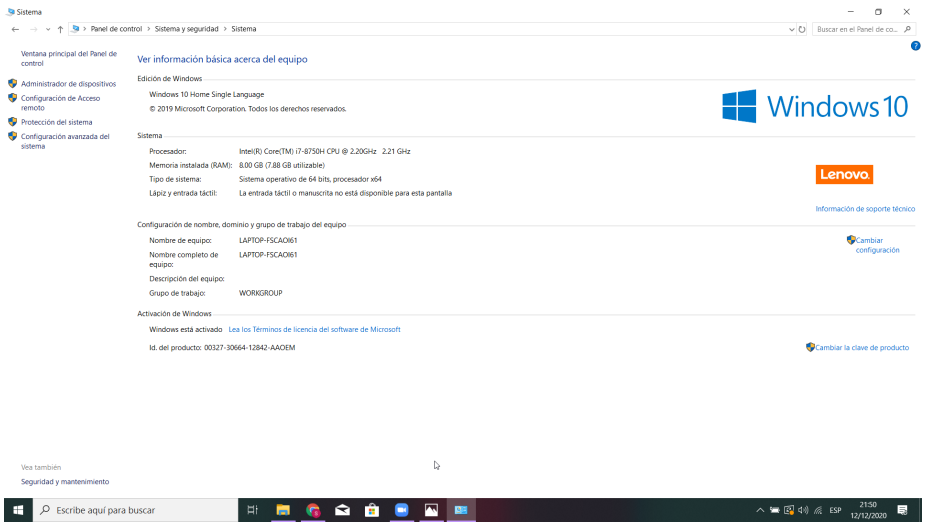


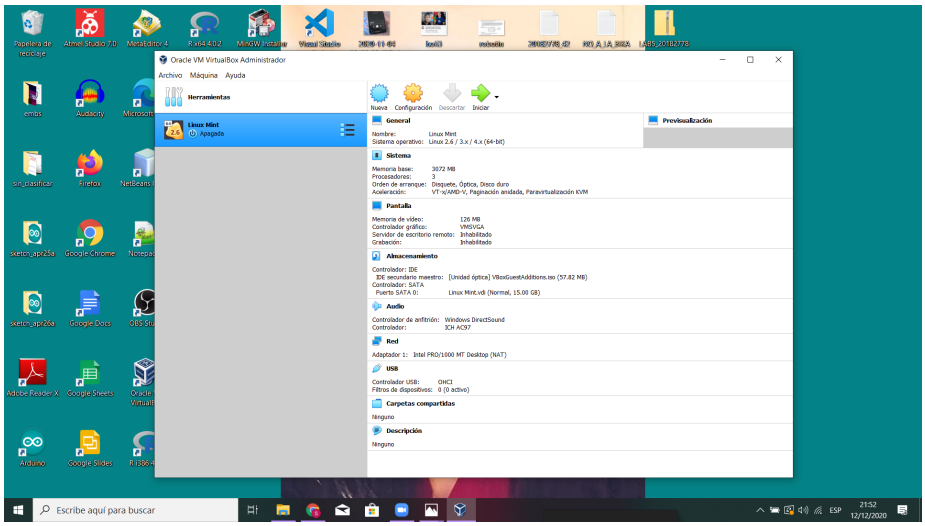


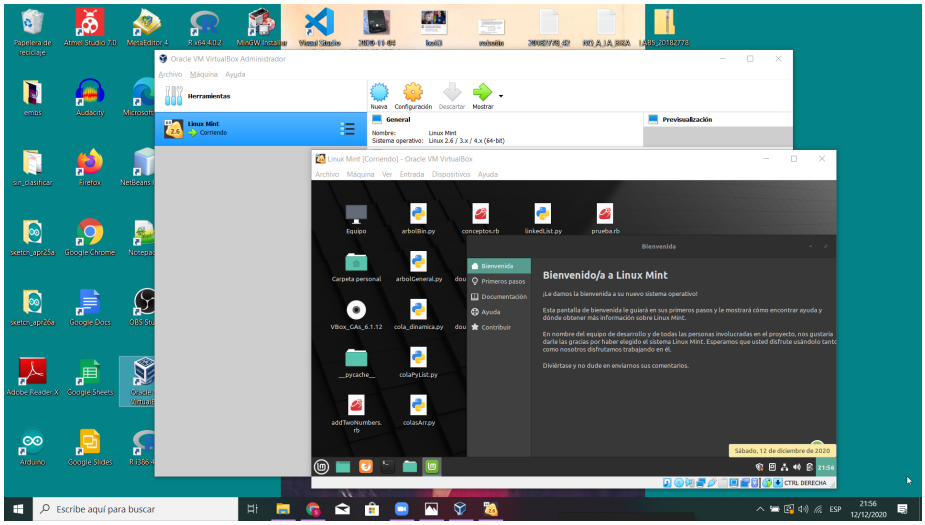
2 Oyarce Tocto, Elizabeth Patricia

This member has done 4.2, 4.3.1. and 4.3.2.

2.1 Resources and Oracle VM VirtualBox parameters



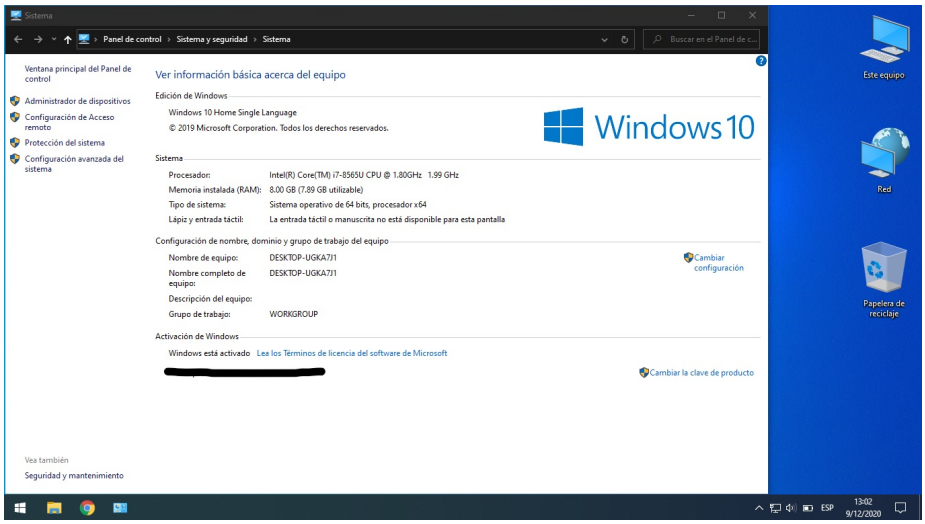


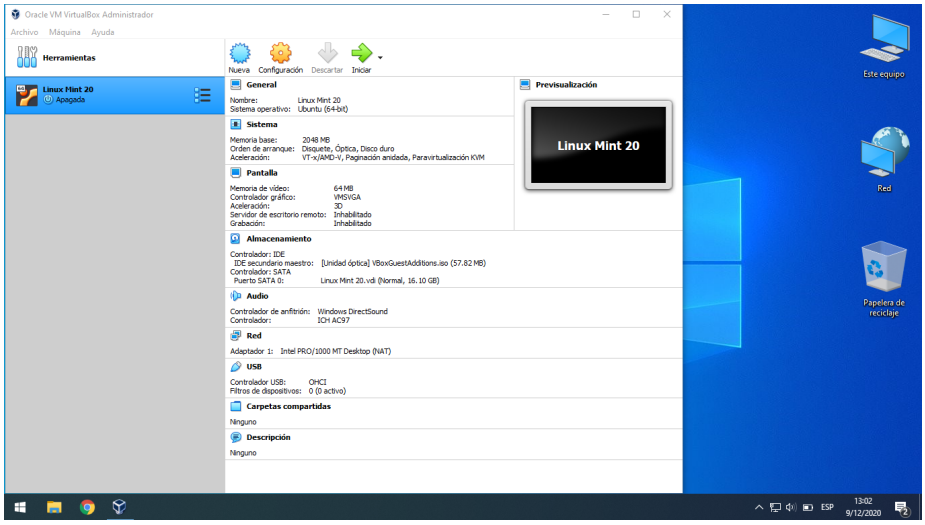


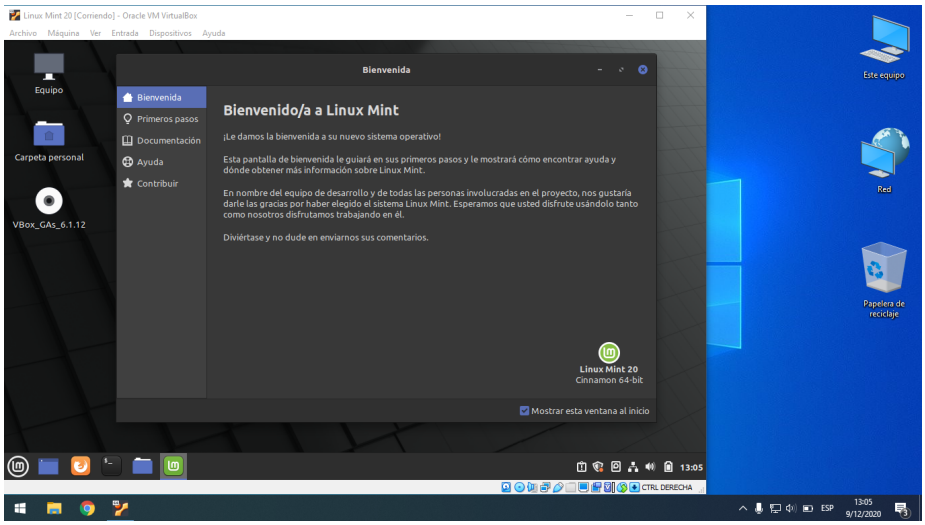
3 Saras Rivera, André Edgardo

This member has done 4.2, 4.3.1. and 4.3.2.

3.1 Resources and Oracle VM VirtualBox parameters







4 Single Source Shortest Path

4.1 Single Source Shortest Path Problem

Finding a shortest path between two nodes of a graph is an important problem that has may practical applications For example, a natural problem related to a road network is to calculate the shortest possible length of a route between two cities, given the lengths of the roads.

In an unweigthed graph, the length of a path equals the number of its edges and we can simply use a breadth-first search to find a shortest path as we did in our previous report. However, in this report we focus on weighted graphs where more sophisticated algorithms are needed for finding shortest paths.

Given a directed graph $G(V, E)$, with weight function $w : E \to R$. The weight $w(p)$ of a path $p = \langle v_0, v_1, \ldots, v_k \rangle$ is the sum of the weights of its edges

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

We define the shortest path weight $\delta(u, v)$ from u to v by

$$\delta(u, v) = \begin{cases} \min\{w(p) \mid \forall\, p = \langle u, \ldots, v \rangle\} & \text{if there is a path from } u \text{ to } v \\ \infty & \text{otherwise.} \end{cases}$$

A shortest path from vertex u to v is then defined as any path p with weight $w(p) = \delta(u, v)$.

4.2 Main Algorithm

Dijkstra's algorithm finds shortest paths from the starting node to all nodes of the graph. The algorithm requires that there are no negative weight edges in the graph.

Dijkstra's algorithm maintains distances to the nodes and reduces them during the search. At each step, Dijkstra's algorithm selects a node that has not been processed yet and whose distance is as small as possible. Then, the algorithm goes through all edges that start at the node and reduces the distances using them. Dijkstra's algorithm is efficient because it only processes each edge in the graph once, using the fact that there are no negative edges.

4.3 Implementation

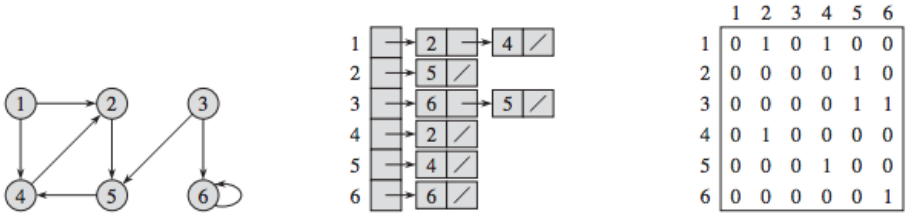
An efficient implementation of Dijkstra's algorithm requires that we can efficiently find the minimum distance to a node that has not been processed. An appropriate data structure for this is min-heap that contains the remaining nodes ordered by their distances. Using a min-heap, the next node to be processed can be retrieved in logarithmic time. Due to not being able to modify the value in the min-heap, we are going to add a new instance of a node to the min-heap always when its distance changes.

Our implementation of Dijkstra's algorithm calculates the minimum distances from a specific node to all other nodes of the graph. The graph is stored as adjacency lists so that `adj[`u`]` contains a pair (v, w) always when there is an edge from node u to v with weight w. The min-heap q contains pairs of the form (d, x) meaning that the current distance to node x is d. The array `distance` contains the distance to each node and the array `parent` the last node that has made each node to reduce its distance.

The implementation is as follows:

4.3.1 Graph

The structures we have to research and implement in this report are graphs. There are two common ways to represent a graph: using and adjacency matrix or using an adjacency list. Given a graph $G = (V, E)$, the adjacency matrix representation is easier because we are going to create an array of $|V|^2$ units of memory and a_{ij} will represent if an edge between the nodes i and j exists. However, we need a lot of memory to represent the graph and, also, there will be too much unused space. As a consequence, the adjacency list is the most efficient, in terms of space complexity, way to represent a graph because we are going to create nodes on each list that represent each existing edge if necessary.



As we can see, we are going to represent a graph like the one in (a) using an array of linked lists in (b). In Ruby, we can use `Hash`, which is the implementation of a hash table, instead of a linked list and the time complexity is not going to be affected.

The class `Graph` represents a graph with the specified nodes and edges. I have created a class `Edge`. This is a simplified version of the implementation from the previous report because we are not going to use all the methods and we are going to focus on the edges instead of the nodes.

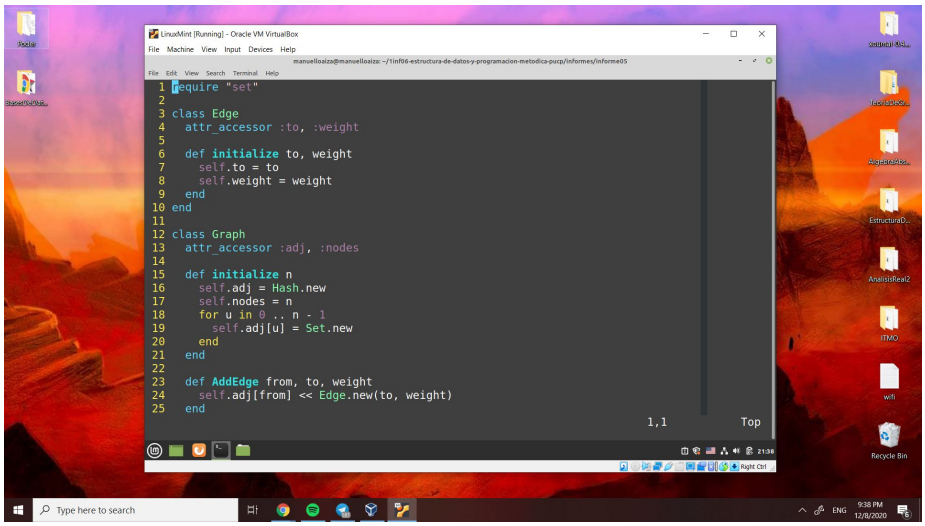
`Edge` definition has two attributes: `to` and `weight`, which represents the endpoint of a directed edge and its weight. If our graph is undirected, then we have to add the reversed edge too.

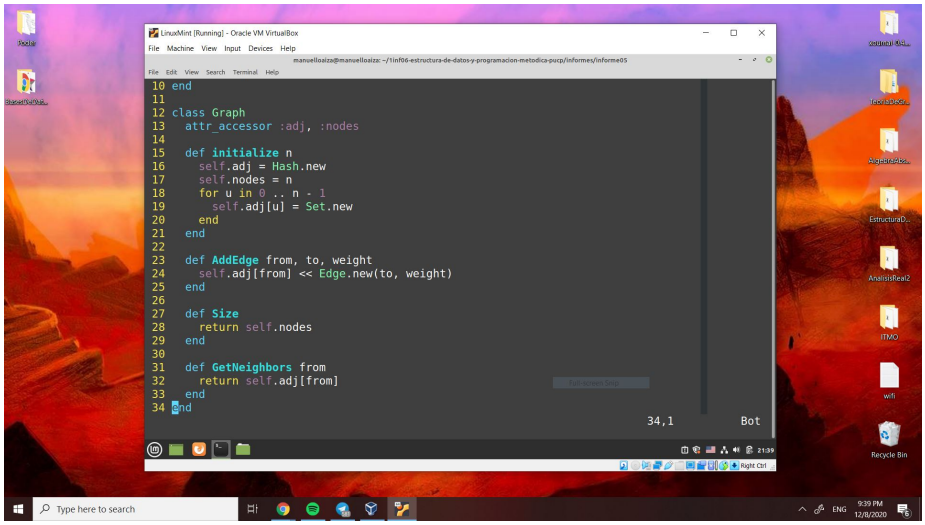
- Method: `initialize`
  Usage: `graph = Graph.new(n)`
  Creates a graph with n nodes.

- Method: `Size`
  Usage: `size = graph.Size`
  Returns the number of nodes in the graph.

- Method: `AddEdge`
  Usage: `graph.AddEdge(from, to, weight)`
  Adds an oriented weighted edge to the graph.

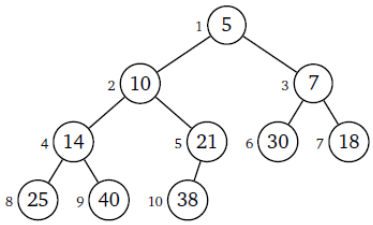
- Method: `GetNeighbors`
  Usage: `neighbors = graph.GetNeighbors(node_id)`
  Returns the set of edges that start at the specified node.





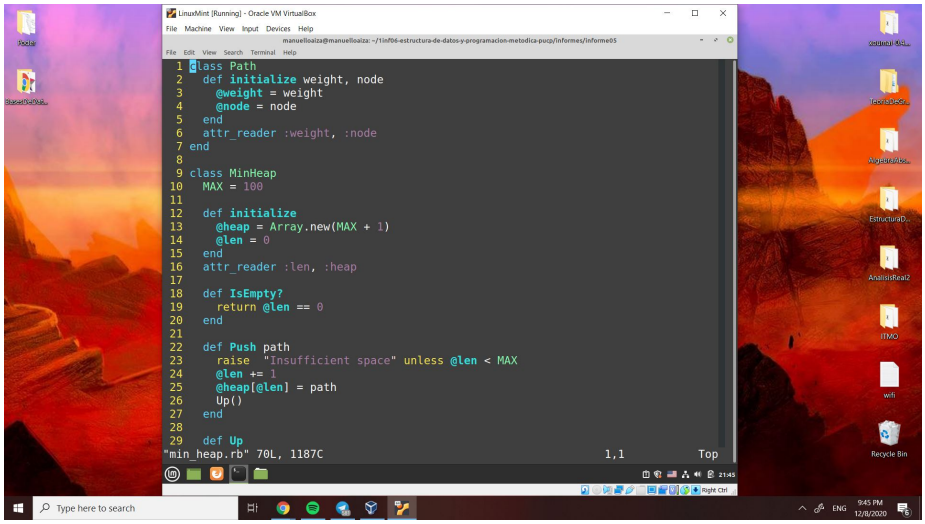
4.3.2 Min-Heap

A Min-Heap is a complete binary tree where the element in the root is less than all elements in the left and right subtrees, and both children satisfy this property too. It is usually represented by an arrangement where a node stored at position i has its left child at position $2i$ and its right child in position $2i + 1$



1	2	3	4	5	6	7	8	9	10
5	10	7	14	21	30	18	25	40	38

We need the class `Path` to be able to work. A path of length k from a vertex u to a vertex v in a graph $G = (V, E)$ is a sequence $\langle v_0, v_1, v_2 \ldots, v_k \rangle$ of distinct vertices such that $u = v_0$ and $v_k = v$, and $(v_{i-1}, v_i) \in E$ for $i = 1, 2, \ldots, k$. The length of the path is the number of edges in the path. There is always a 0-length path from u to u. In particular, since all the paths used in the algorithm start at the same initial vertex, it is only necessary to put where it ends and the weight of the path to define the class.



- Method: `initialize`
  Usage: `minHeap = MinHeap.new`
  Creates a min-heap using an array with initial length 0.

- Method: `IsEmpty?`
  Usage: `minHeap.IsEmpty?`
  Returns a boolean value true if it is empty and false otherwise.

- Method: `Push`
  Usage: `minHeap.Push(path)`
  Places the new path at the end of the arrangement and then by means of the *Up* function makes it float to its proper position.

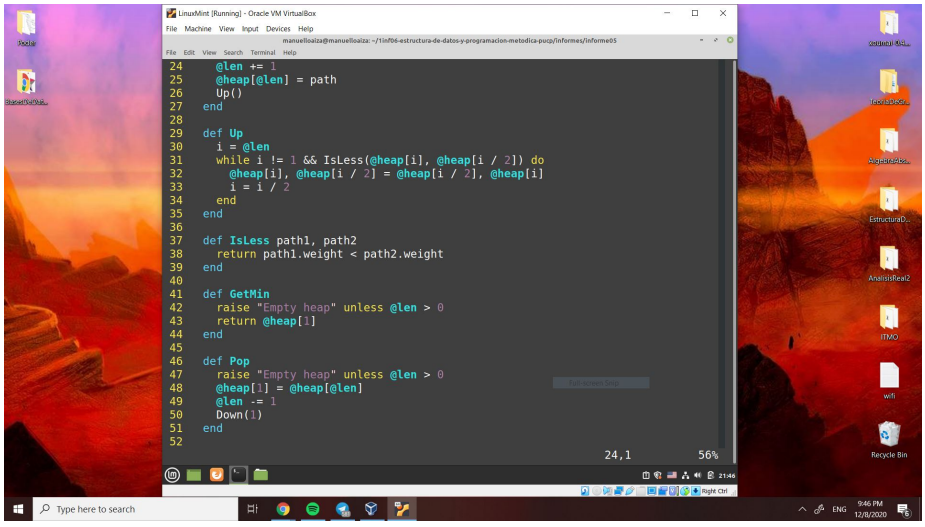
- Method: `Up`
  Usage: `Up()`
  Swaps the path with its parent until it is less than or equal, or the root is reached.

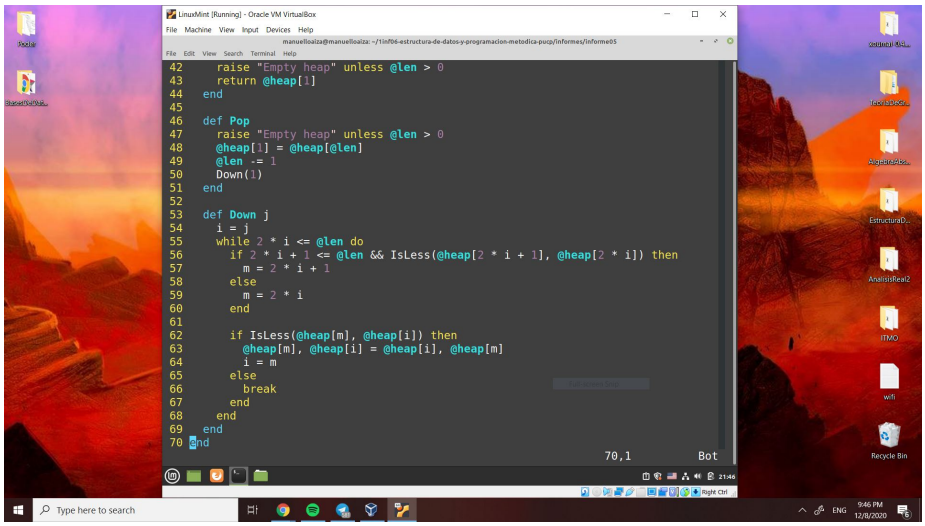
- Method: `IsLess`
  Usage: `var = IsLess(path1,path2)`
  Compare two paths by their weight attribute.

- Method: `GetMin`
  Usage: `min = minHeap.GetMin`
  Returns the minimum element of the arrangement.

- Method: `Pop`
  Usage: `minHeap.Pop`
  Swaps the first path with the last path and then by means of the *Down* function makes it fall to its proper position.

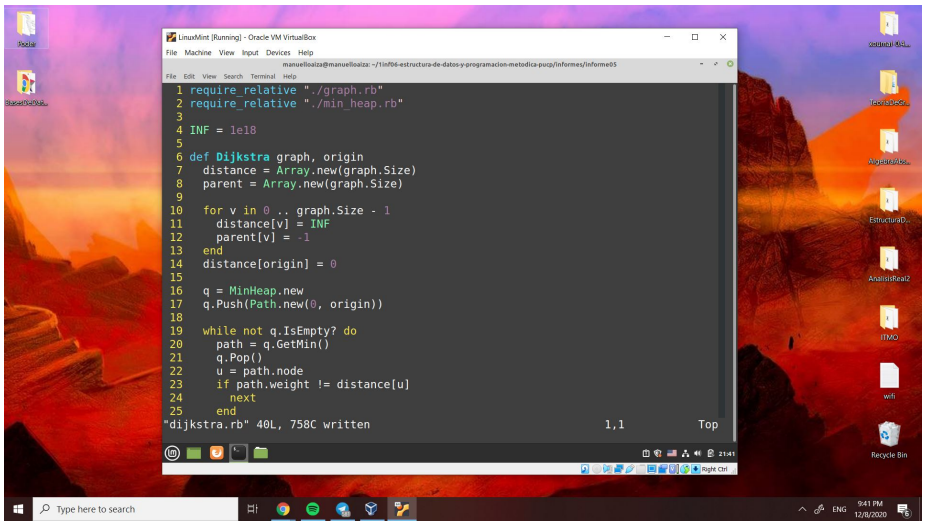
- Method: `Down`
  Usage: `Down(id)`
  Swaps the path with its minimum son until it is higher than or equal, or a leaf is reached.

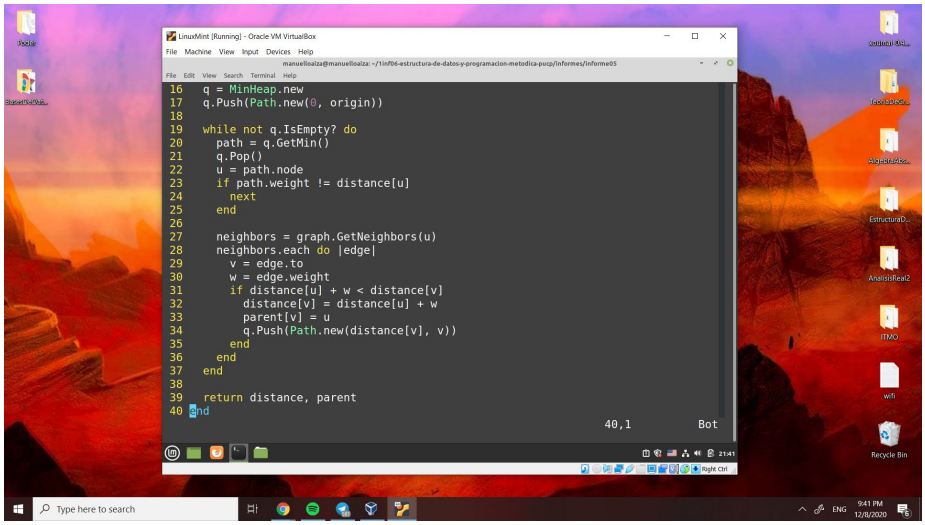




4.3.3 Dijkstra's Algorithm

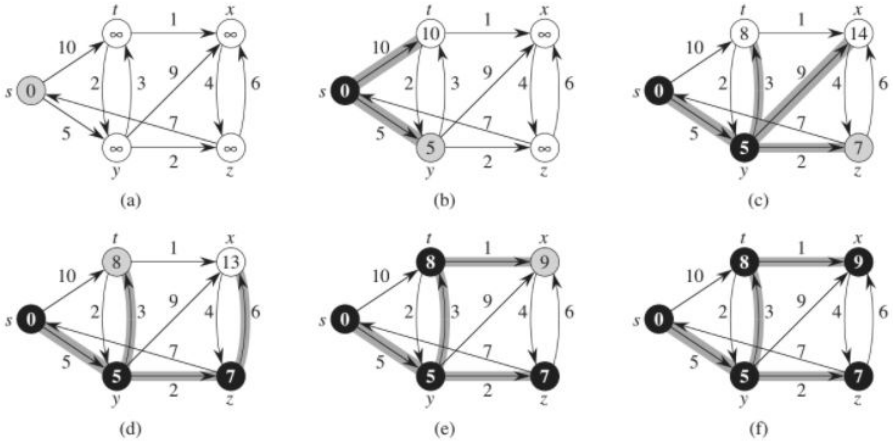
The time complexity of this implementation is $O((V + E) \log_2 E)$ because the algorithm goes through all nodes of the graph and adds for each edge at most one distance to the min-heap. The efficiency of Dijkstra's algorithm is based on the fact that the graph does not have negative edges. However, if the graph has a negative edge, the algorithm may give incorrect results.

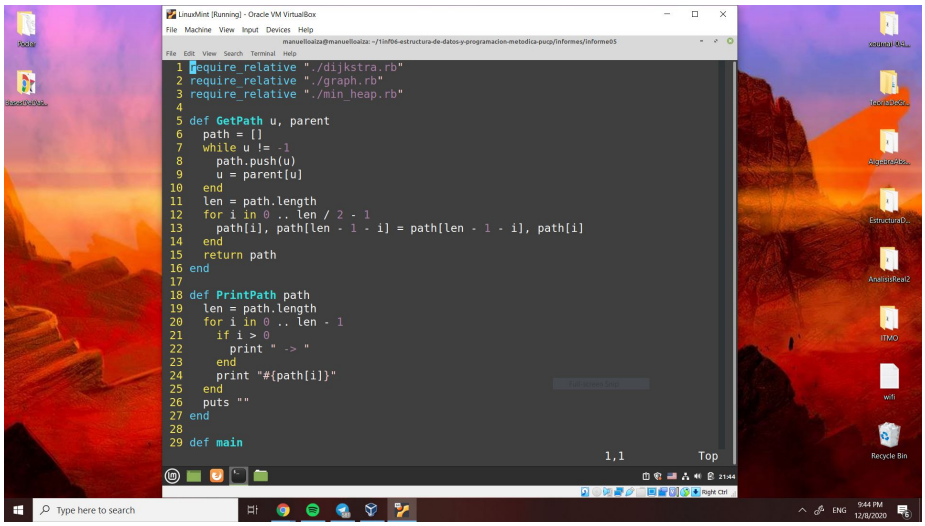


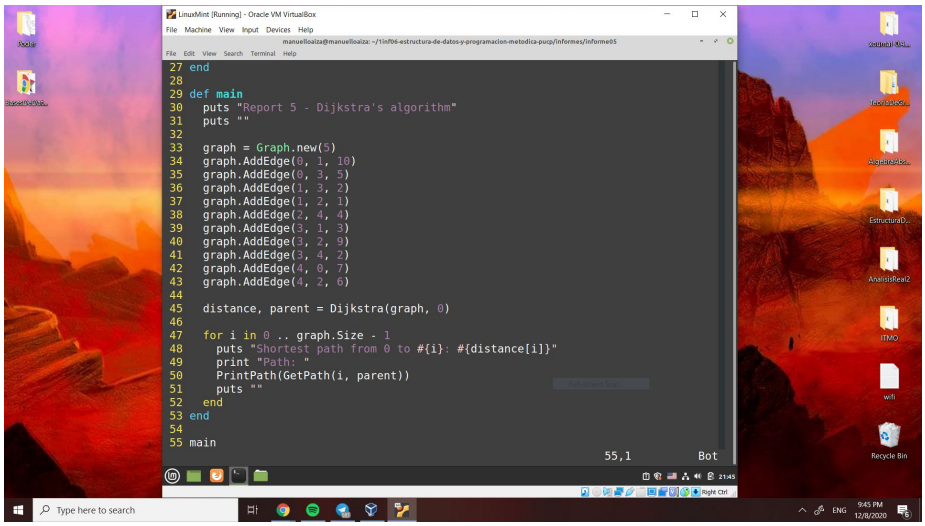


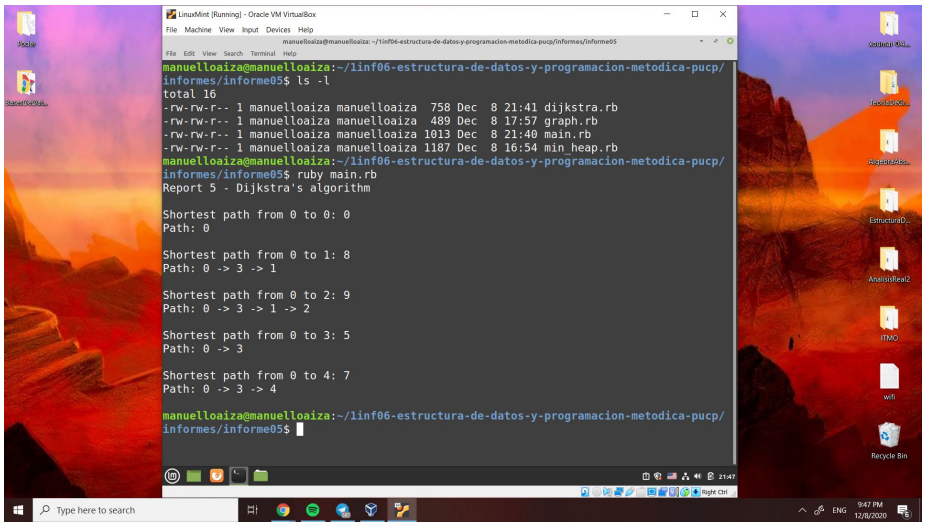
4.3.4 Testing

We are going to test our implementation with the following graph printing the weight and the path of the shortest path from s to each node. The nodes $0, 1, 2, 3, 4$ represent s, t, x, y, z, respectively.




LinuxMint [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
manuelloaiza@manuelloaiza: ~/1inf06-estructura-de-datos-y-programacion-metodica-pucp/informes/informe05
File Edit View Search Terminal Help
require_relative "./dijkstra.rb"
require_relative "./graph.rb"
require_relative "./min_heap.rb"
def GetPath u, parent
path = []
while u != -1
path.push(u)
u = parent[u]
end
len = path.length
for i in 0 .. len / 2 - 1
path[i], path[len - 1 - i] = path[len - 1 - i], path[i]
end
return path
end
def PrintPath path
len = path.length
for i in 0 .. len - 1
if i > 0
print " -> "
end
print "#{path[i]}"
end
puts ""
end
def main
1,1 Top
Type here to search
ENG 9:44 PM 12/8/2020


LinuxMint [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
manuelloaiza@manuelloaiza: ~/1inf06-estructura-de-datos-y-programacion-metodica-pucp/informes/informe05
File Edit View Search Terminal Help
end
def main
puts "Report 5 - Dijkstra's algorithm"
puts ""
graph = Graph.new(5)
graph.AddEdge(0, 1, 10)
graph.AddEdge(0, 3, 5)
graph.AddEdge(1, 3, 2)
graph.AddEdge(1, 2, 1)
graph.AddEdge(2, 4, 4)
graph.AddEdge(3, 1, 3)
graph.AddEdge(3, 2, 9)
graph.AddEdge(3, 4, 2)
graph.AddEdge(4, 0, 7)
graph.AddEdge(4, 2, 6)
distance, parent = Dijkstra(graph, 0)
for i in 0 .. graph.Size - 1
puts "Shortest path from 0 to #{i}: #{distance[i]}"
print "Path: "
PrintPath(GetPath(i, parent))
puts ""
end
end
main
55,1 Bot
Type here to search
ENG 9:45 PM 12/8/2020


LinuxMint [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
manuelloaiza@manuelloaiza: ~/1inf06-estructura-de-datos-y-programacion-metodica-pucp/informes/informe05
File Edit View Search Terminal Help
manuelloaiza@manuelloaiza:~/1inf06-estructura-de-datos-y-programacion-metodica-pucp/informes/informe05$ ls -l
total 16
-rw-rw-r-- 1 manuelloaiza manuelloaiza 758 Dec 8 21:41 dijkstra.rb
-rw-rw-r-- 1 manuelloaiza manuelloaiza 489 Dec 8 17:57 graph.rb
-rw-rw-r-- 1 manuelloaiza manuelloaiza 1013 Dec 8 21:40 main.rb
-rw-rw-r-- 1 manuelloaiza manuelloaiza 1187 Dec 8 16:54 min_heap.rb
manuelloaiza@manuelloaiza:~/1inf06-estructura-de-datos-y-programacion-metodica-pucp/informes/informe05$ ruby main.rb
Report 5 - Dijkstra's algorithm
Shortest path from 0 to 0: 0
Path: 0
Shortest path from 0 to 1: 8
Path: 0 -> 3 -> 1
Shortest path from 0 to 2: 9
Path: 0 -> 3 -> 1 -> 2
Shortest path from 0 to 3: 5
Path: 0 -> 3
Shortest path from 0 to 4: 7
Path: 0 -> 3 -> 4
manuelloaiza@manuelloaiza:~/1inf06-estructura-de-datos-y-programacion-metodica-pucp/informes/informe05$
Recycle Bin
Type here to search
ENG
9:47 PM
12/8/2020