1INF06 Data Structures and Methodical Programming

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

November 11, 2020

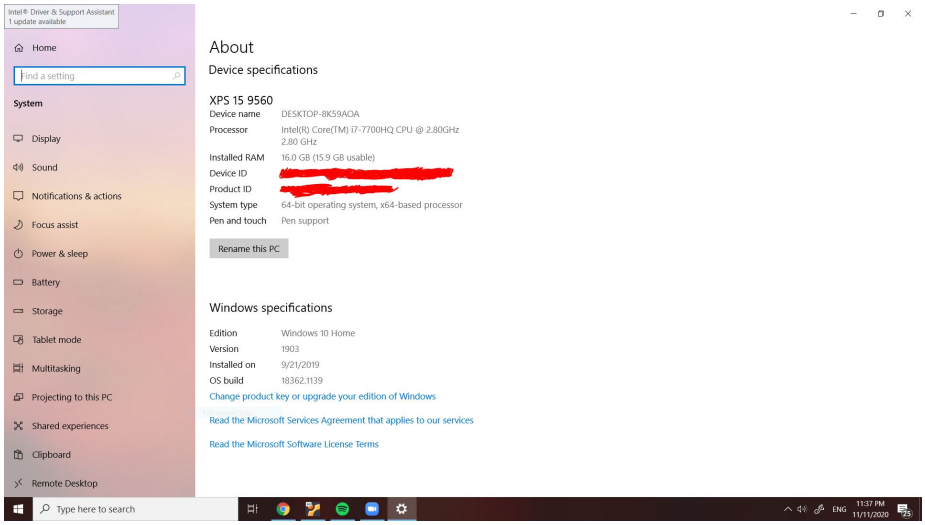
Pontificia Universidad Católica del Perú
Lima, Perú
manuel.loaiza@pucp.edu.pe
a20182778@pucp.edu.pe
andre.saras@pucp.edu.pe

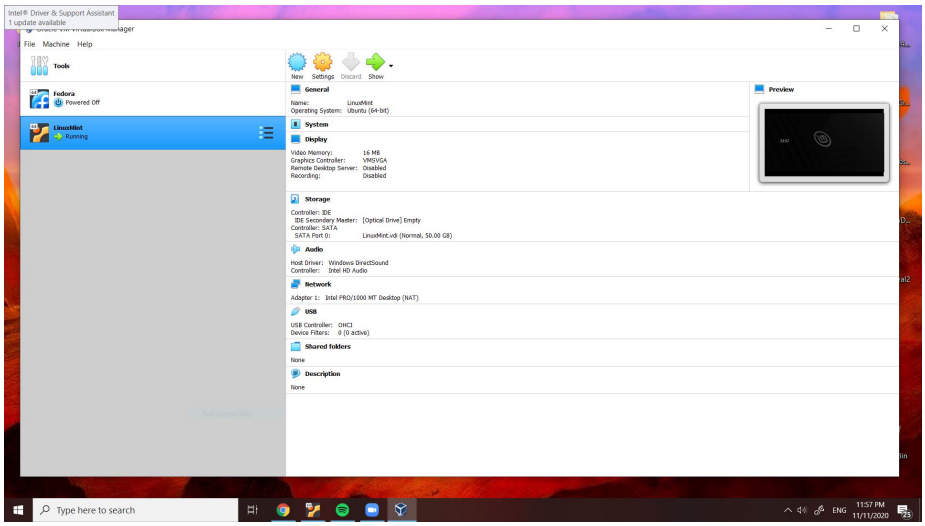
Fourth report of the course Data Structures and Methodical Programming taught at the Faculty of Science and Engineering at Pontificia Universidad Católica del Perú (PUCP) by Viktor Khlebnikov in the semester 2020-2.

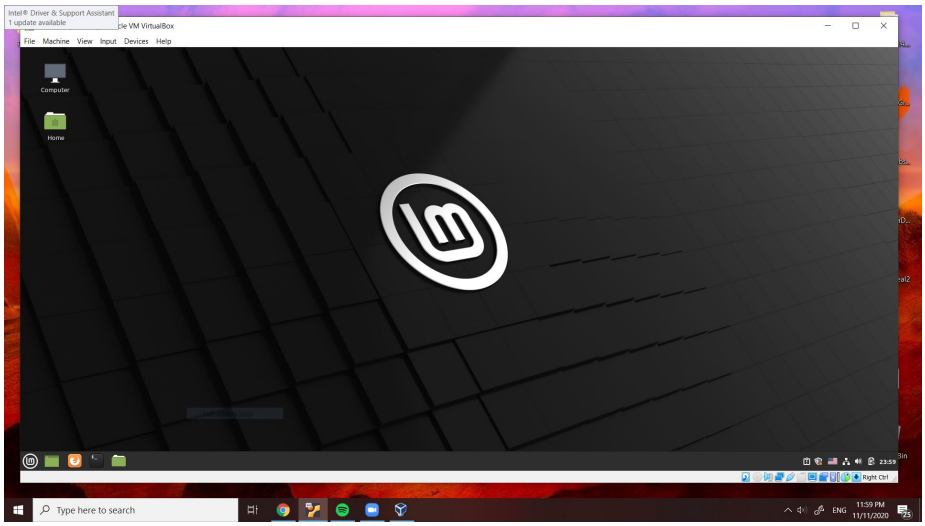
Contents

1 Loaiza Vasquez, Manuel Alejandro

1.1 Resources and Oracle VM VirtualBox parameters







1.2 Graph implementation using map

The class `Graph` represents a graph with the specified nodes and edges. I have created two more classes: `Node` and `Edges`.

`Node` definition has two attributes: `value` and `neighbors`. The first one is going to be an identifier for a node, for example, the student code of a PUCP student. The second one is a map (also known as dictionaries or `Hash` in Ruby) in which each element of the map stores a pointer to a `Node` instance and the key of that element is the identifier of the node pointed by the pointer.

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

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

- Method: `IsEmpty`
  Usage: `if graph.IsEmpty . . .`
  Returns `true` if the graph is empty.

- Method: `AddNode`
  Usage: `graph.AddNode(node_id)`
  Add a node to the graph.

- Method: `RemoveNode`
  Usage: `graph.RemoveNode(node_id)`
  Removes a node from the graph by its identifier. It also remove all the edges that contains at least one endpoint with these node.

- Method: `GetNode`
  Usage: `node_ptr = graph.GetNode(node_id)`
  Looks up a node in the map attached to the graph and returns a pointer to that node.

- Method: `NodeExists`
  Usage: `if graph.NodeExists(node_id) . . .`
  Return `true` if a node with the give identifier exists

- Method: `AddEdge`
  Usage: `graph.AddEdge(from, to)`
  Adds an edge to the graph. If a node doesn't exist in the graph, then it is going to be added.

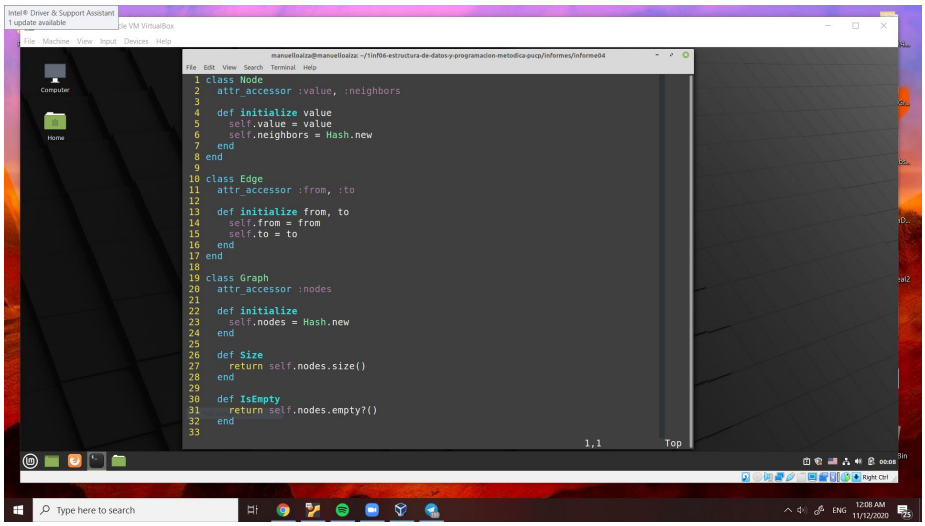
- Method: `RemoveEdge`
  Usage: `graph.RemoveEdge(from, to)`
  Removes an edge from the graph.

- Method: `IsConnected`
  Usage: `if graph.IsConnected(u, v) . . .`
  Return `true` if the graph contains an edge from `u` to `v`.

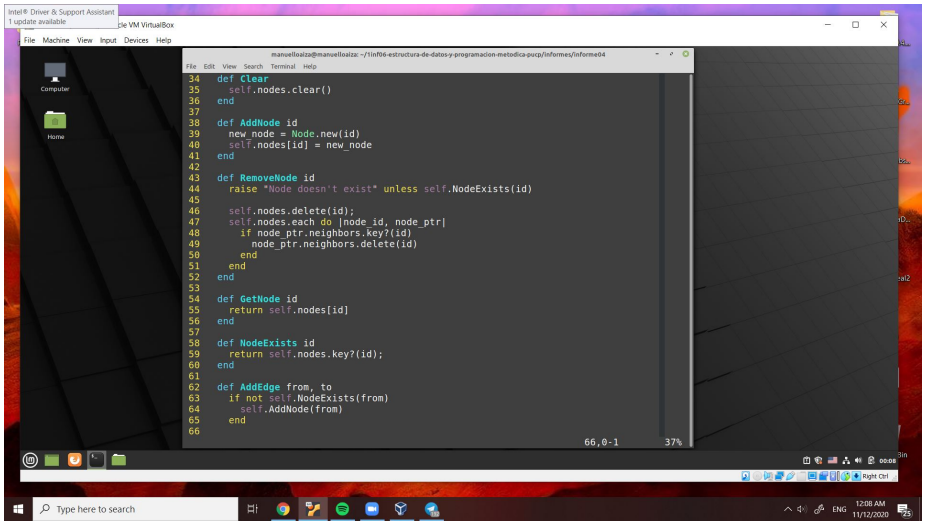
- Method: `GetNodeSet`
  Usage: `nodes = graph.GetNodeSet`
  Returns the set of all nodes in the graph.

- Method: `GetEdgeSet`
  Usage: `edges = graph.GetEdgeSet`
  Returns the set of all edges in the graph.

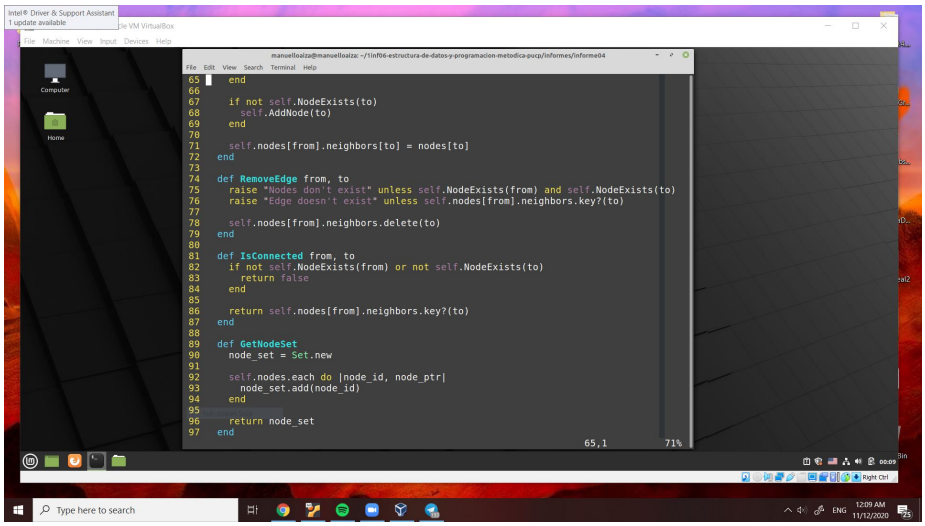
- Method: `GetNeighbors`
  Usage: `neighbors = graph.GetNeighbors(node_id)`
  Returns the set of nodes that are neighbors of the specified node.



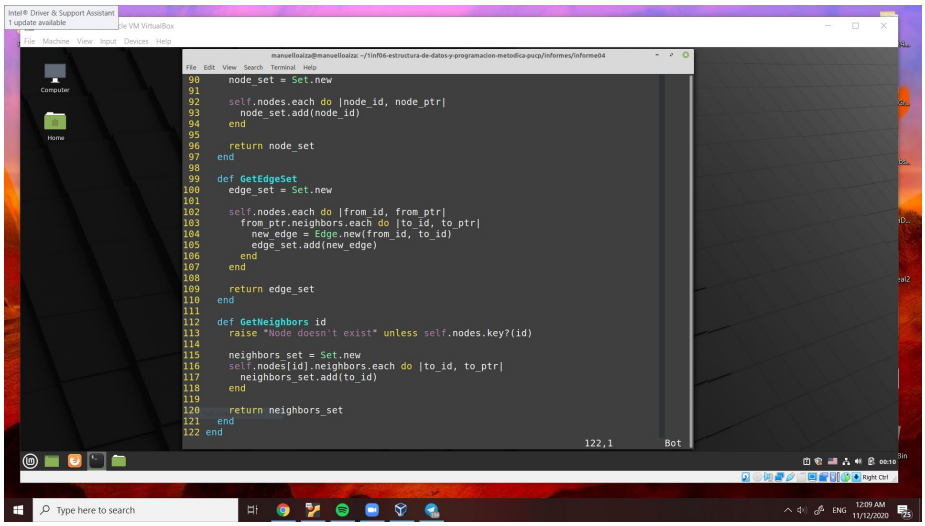
```
class Node
  attr_accessor :value, :neighbors

  def initialize value
    self.value = value
    self.neighbors = Hash.new
  end
end

class Edge
  attr_accessor :from, :to

  def initialize from, to
    self.from = from
    self.to = to
  end
end

class Graph
  attr_accessor :nodes

  def initialize
    self.nodes = Hash.new
  end

  def Size
    return self.nodes.size()
  end

  def IsEmpty
    return self.nodes.empty?()
  end
```



```
  def Clear
    self.nodes.clear()
  end

  def AddNode id
    new_node = Node.new(id)
    self.nodes[id] = new_node
  end

  def RemoveNode id
    raise "Node doesn't exist" unless self.NodeExists(id)

    self.nodes.delete(id);
    self.nodes.each do |node_id, node_ptr|
      if node_ptr.neighbors.key?(id)
        node_ptr.neighbors.delete(id)
      end
    end
  end

  def GetNode id
    return self.nodes[id]
  end

  def NodeExists id
    return self.nodes.key?(id);
  end

  def AddEdge from, to
    if not self.NodeExists(from)
      self.AddNode(from)
    end
```



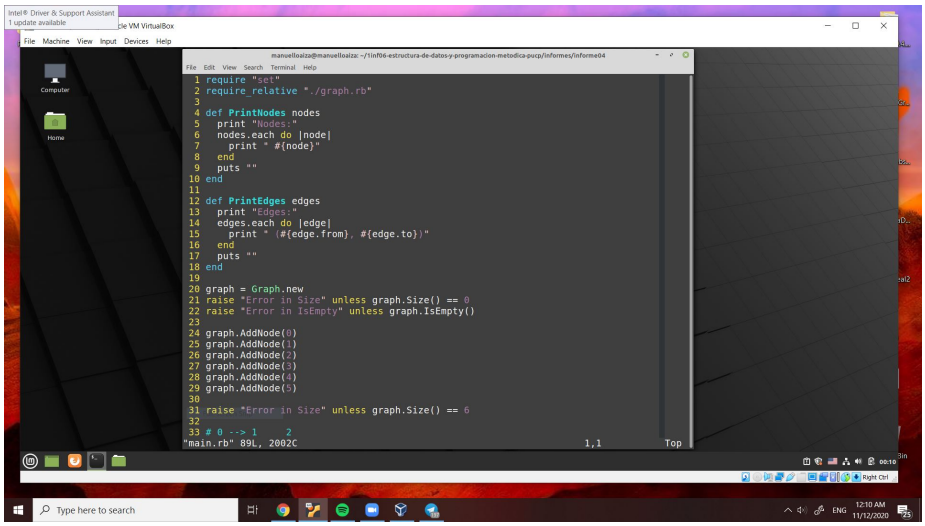
```
65    end
66
67    if not self.NodeExists(to)
68      self.AddNode(to)
69    end
70
71    self.nodes[from].neighbors[to] = nodes[to]
72  end
73
74  def RemoveEdge from, to
75    raise "Nodes don't exist" unless self.NodeExists(from) and self.NodeExists(to)
76    raise "Edge doesn't exist" unless self.nodes[from].neighbors.key?(to)
77
78    self.nodes[from].neighbors.delete(to)
79  end
80
81  def IsConnected from, to
82    if not self.NodeExists(from) or not self.NodeExists(to)
83      return false
84    end
85
86    return self.nodes[from].neighbors.key?(to)
87  end
88
89  def GetNodeSet
90    node_set = Set.new
91
92    self.nodes.each do |node_id, node_ptr|
93      node_set.add(node_id)
94    end
95
96    return node_set
97  end
```

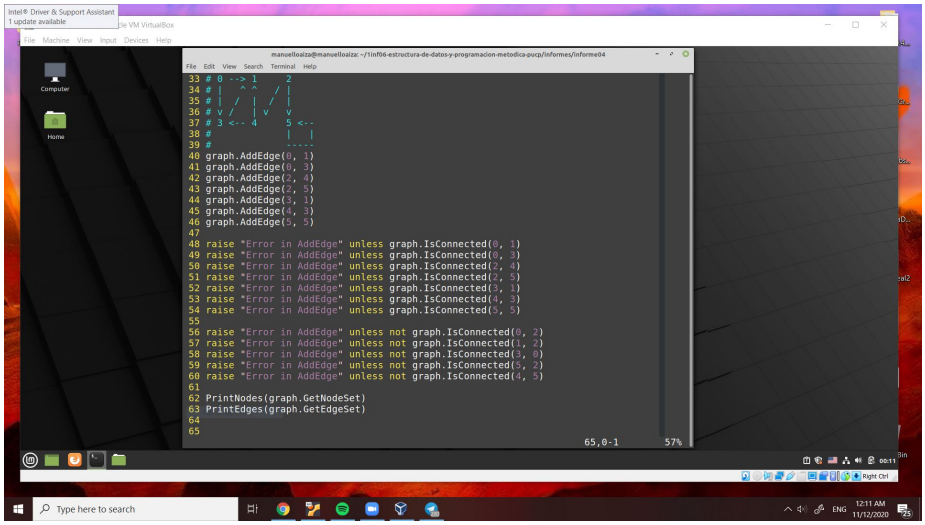


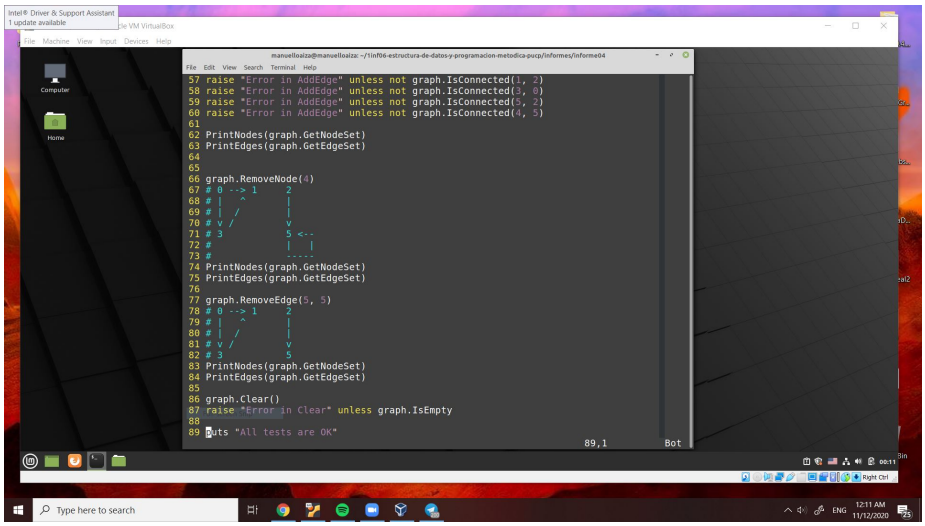
```
90    node_set = Set.new
91
92    self.nodes.each do |node_id, node_ptr|
93      node_set.add(node_id)
94    end
95
96    return node_set
97  end
98
99  def GetEdgeSet
100   edge_set = Set.new
101
102   self.nodes.each do |from_id, from_ptr|
103     from_ptr.neighbors.each do |to_id, to_ptr|
104       new_edge = Edge.new(from_id, to_id)
105       edge_set.add(new_edge)
106     end
107   end
108
109   return edge_set
110 end
111
112 def GetNeighbors id
113   raise "Node doesn't exist" unless self.nodes.key?(id)
114
115   neighbors_set = Set.new
116   self.nodes[id].neighbors.each do |to_id, to_ptr|
117     neighbors_set.add(to_id)
118   end
119
120   return neighbors_set
121 end
122 end
```

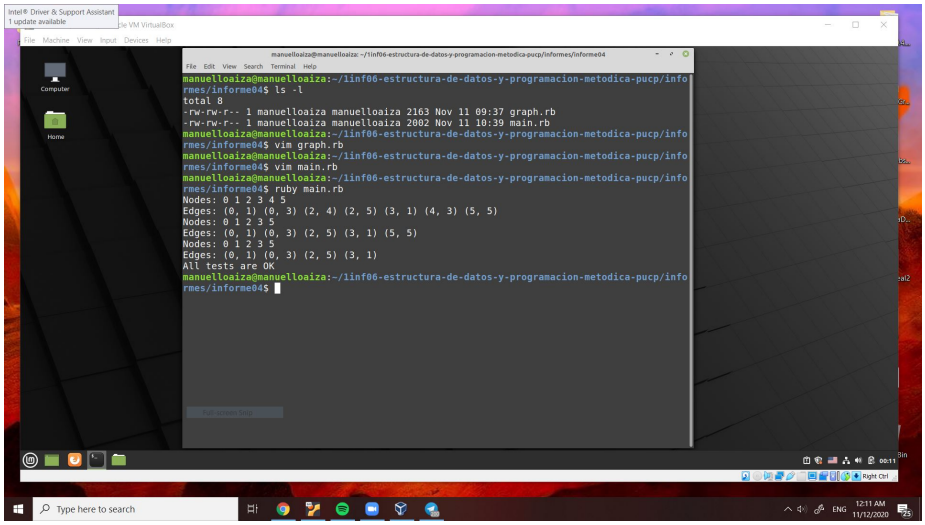
1.3 Ruby example program

The following program has the goal to test if the methods implemented in the class `Graph` are correct. The first header is for using the `Set` data structure in which I am going to store the partial results of my analysis to print them on screen to see how my graph has changed after deleting a vertex o an edge. The second header tells Ruby I want to bring in my `Graph` class.

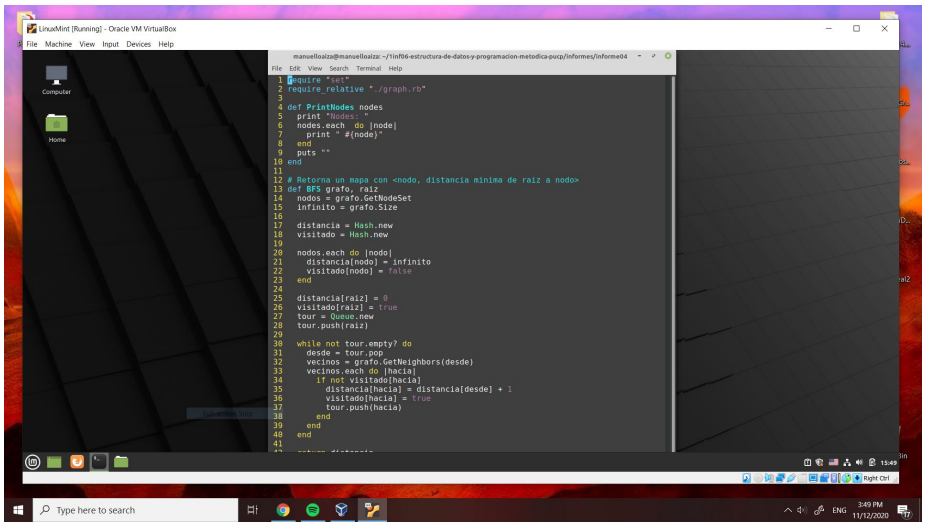

require "set"
require_relative "./graph.rb"
def PrintNodes nodes
print "Nodes:"
nodes.each do |node|
print " #{node}"
end
puts ""
end
def PrintEdges edges
print "Edges:"
edges.each do |edge|
print " (#{edge.from}, #{edge.to})"
end
puts ""
end
graph = Graph.new
raise "Error in Size" unless graph.Size() == 0
raise "Error in IsEmpty" unless graph.IsEmpty()
graph.AddNode(0)
graph.AddNode(1)
graph.AddNode(2)
graph.AddNode(3)
graph.AddNode(4)
graph.AddNode(5)
raise "Error in Size" unless graph.Size() == 6
0 --> 1 2
"main.rb" 89L, 2002C
1,1 Top

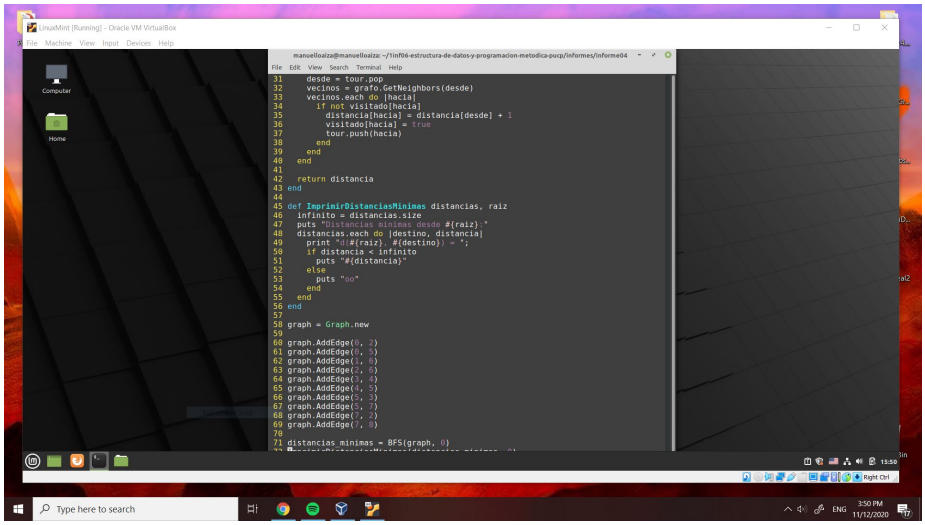

# 0 --> 1 2
# | ^ ^ / |
# | / | / |
# v / | v v
# 3 <-- 4 5 <--
# | |
# -----
graph.AddEdge(0, 1)
graph.AddEdge(0, 3)
graph.AddEdge(2, 4)
graph.AddEdge(2, 5)
graph.AddEdge(3, 1)
graph.AddEdge(4, 3)
graph.AddEdge(5, 5)
raise "Error in AddEdge" unless graph.IsConnected(0, 1)
raise "Error in AddEdge" unless graph.IsConnected(0, 3)
raise "Error in AddEdge" unless graph.IsConnected(2, 4)
raise "Error in AddEdge" unless graph.IsConnected(2, 5)
raise "Error in AddEdge" unless graph.IsConnected(3, 1)
raise "Error in AddEdge" unless graph.IsConnected(4, 3)
raise "Error in AddEdge" unless graph.IsConnected(5, 5)
raise "Error in AddEdge" unless not graph.IsConnected(0, 2)
raise "Error in AddEdge" unless not graph.IsConnected(1, 2)
raise "Error in AddEdge" unless not graph.IsConnected(3, 0)
raise "Error in AddEdge" unless not graph.IsConnected(5, 2)
raise "Error in AddEdge" unless not graph.IsConnected(4, 5)
PrintNodes(graph.GetNodeSet)
PrintEdges(graph.GetEdgeSet)
65,0-1 57%

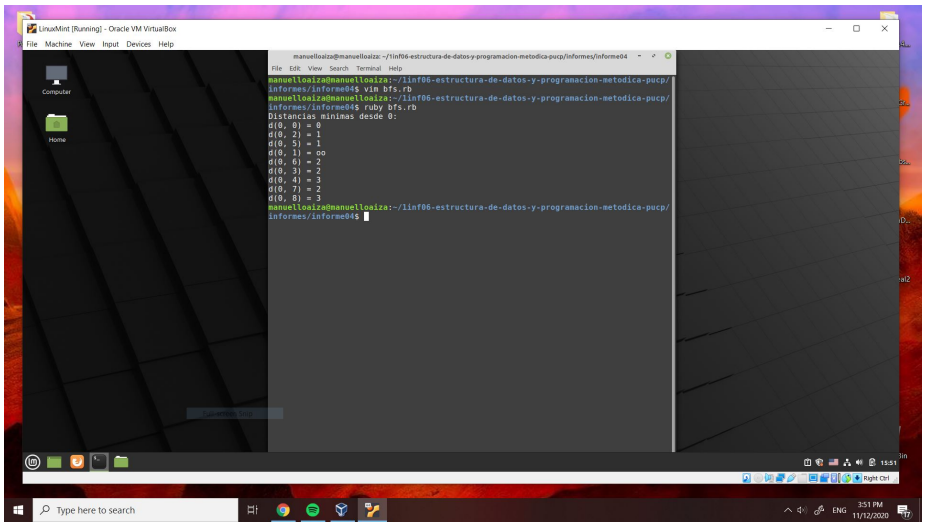




Finally, I am going to apply an algorithm called Breadth First Search to find the minimum distance in an unweighted graph from a specific vertex to the other ones in $O(V + E)$. If a node is not reachable from that vertex, then the minimum distance is going to be ∞. If we want to calculate the minimum distance from a specific vertex to all the other vertices in a weighted graph, we have to add weights to the edges and this algorithm is not going to work.


LinuxMint [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
manuelloaiza@manuelloaiza: ~/1in106-estructura-de-datos-y-programacion-metodica-pucp/informes/informe04
File Edit View Search Terminal Help
require "set"
require_relative "./graph.rb"
def PrintNodes nodes
print "Nodes: "
nodes.each do |node|
print " #{node}"
end
puts ""
end
Retorna un mapa con <nodo, distancia minima de raiz a nodo>
def BFS grafo, raiz
nodos = grafo.GetNodeSet
infinito = grafo.Size
distancia = Hash.new
visitado = Hash.new
nodos.each do |nodo|
distancia[nodo] = infinito
visitado[nodo] = false
end
distancia[raiz] = 0
visitado[raiz] = true
tour = Queue.new
tour.push(raiz)
while not tour.empty? do
desde = tour.pop
vecinos = grafo.GetNeighbors(desde)
vecinos.each do |hacia|
if not visitado[hacia]
distancia[hacia] = distancia[desde] + 1
visitado[hacia] = true
tour.push(hacia)
end
end
end
Computer
Home
Type here to search
ENG
3:49 PM
11/12/2020

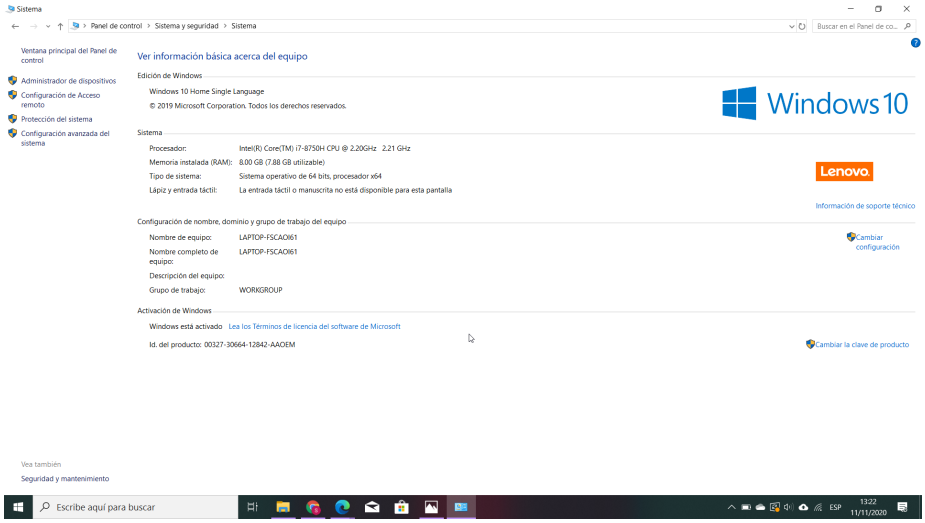

LinuxMint [Running] - Oracle VM VirtualBox
manuelloaiza@manuelloaiza: ~/1in106-estructura-de-datos-y-programacion-metodica-pucp/informes/informe04
File Edit View Search Terminal Help
desde = tour.pop
vecinos = grafo.GetNeighbors(desde)
vecinos.each do |hacia|
if not visitado[hacia]
distancia[hacia] = distancia[desde] + 1
visitado[hacia] = true
tour.push(hacia)
end
end
end
return distancia
end
def ImprimirDistanciasMinimas distancias, raiz
infinito = distancias.size
puts "Distancias minimas desde #{raiz}:"
distancias.each do |destino, distancia|
print "d(#{raiz}, #{destino}) = ";
if distancia < infinito
puts "#{distancia}"
else
puts "oo"
end
end
end
graph = Graph.new
graph.AddEdge(0, 2)
graph.AddEdge(0, 5)
graph.AddEdge(1, 6)
graph.AddEdge(2, 0)
graph.AddEdge(3, 4)
graph.AddEdge(4, 5)
graph.AddEdge(5, 3)
graph.AddEdge(5, 7)
graph.AddEdge(7, 2)
graph.AddEdge(7, 0)
distancias_minimas = BFS(graph, 0)
Computer
Home
Type here to search
ENG
3:50 PM
11/12/2020


LinuxMint [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Computer
Home
manuelloaiza@manuelloaiza: ~/1inf06-estructura-de-datos-y-programacion-metodica-pucp/informes/informe04
File Edit View Search Terminal Help
manuelloaiza@manuelloaiza:~/1inf06-estructura-de-datos-y-programacion-metodica-pucp/informes/informe04$ vim bfs.rb
manuelloaiza@manuelloaiza:~/1inf06-estructura-de-datos-y-programacion-metodica-pucp/informes/informe04$ ruby bfs.rb
Distancias mínimas desde 0:
d(0, 0) = 0
d(0, 2) = 1
d(0, 5) = 1
d(0, 1) = ∞
d(0, 6) = 2
d(0, 3) = 2
d(0, 4) = 3
d(0, 7) = 2
d(0, 8) = 3
manuelloaiza@manuelloaiza:~/1inf06-estructura-de-datos-y-programacion-metodica-pucp/informes/informe04$
Right Ctrl
Type here to search
ENG
3:51 PM
11/12/2020

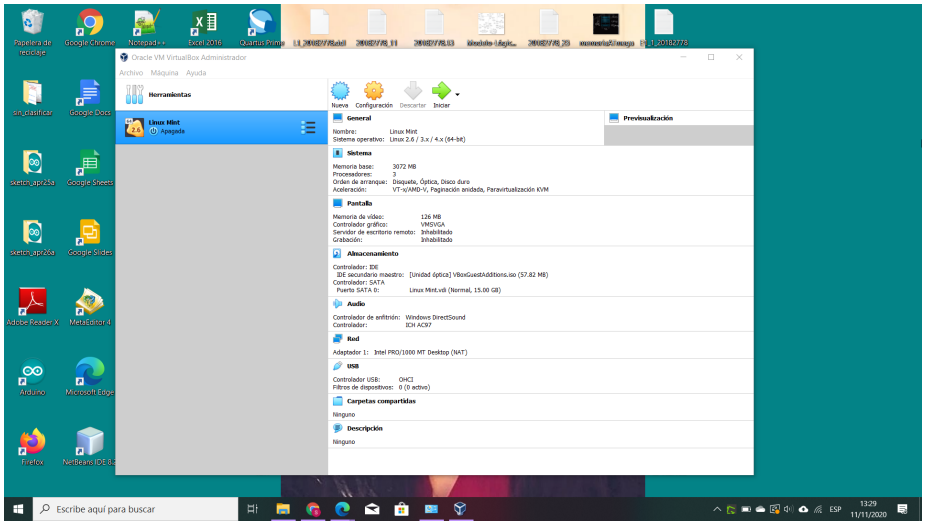
2 Oyarce Tocto, Elizabeth Patricia

2.1 Resources and Oracle VM VirtualBox parameters

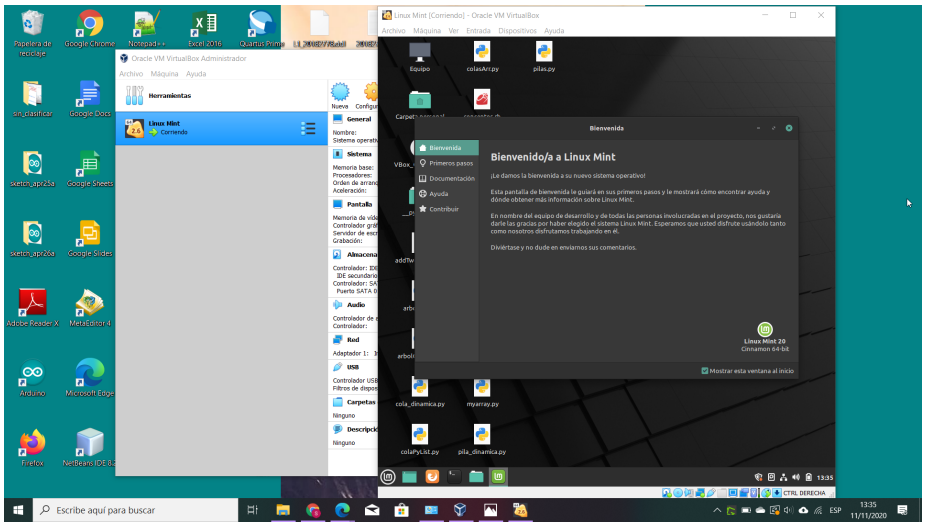
Computer resources



VirtualBox parameters



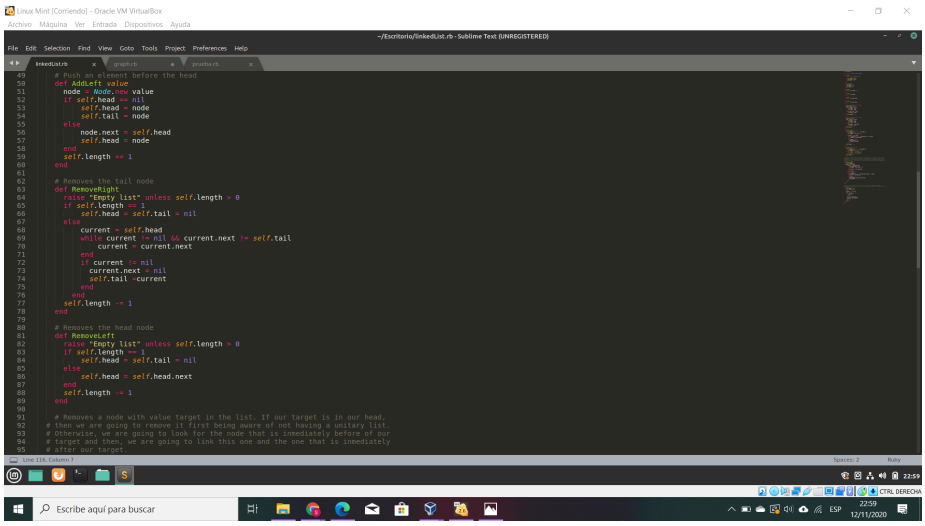
Linux Mint desktop

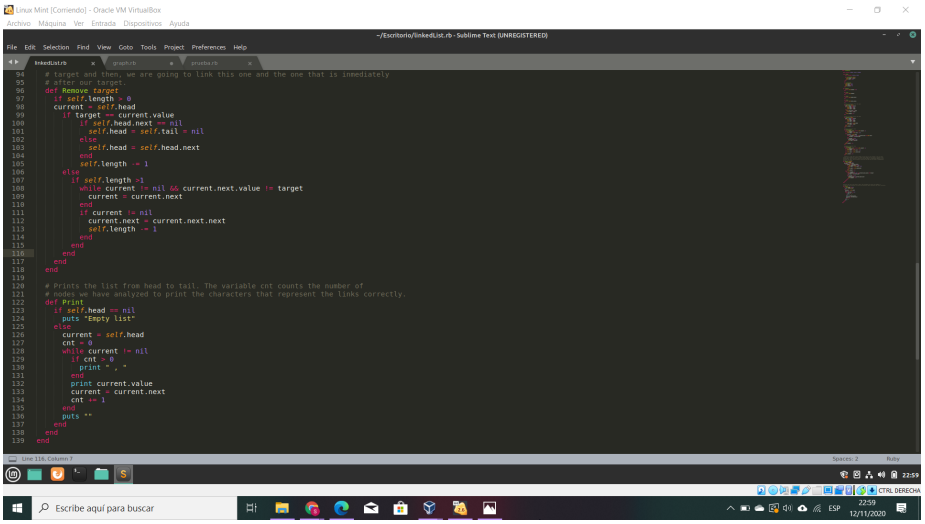


2.2 Graph implementation using linked lists

LinkedList program

The following images are of the linked list code which is seen to implement the graph. This TAD was explained in detail in the previous report.



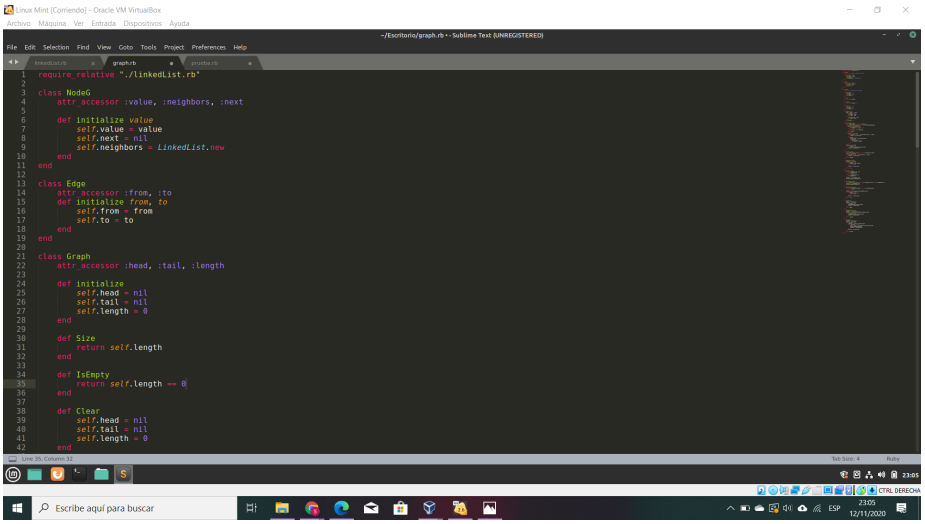


Graph implementation

To define the `Graph` class, we will use two helper classes: the `NodeG` class and the `Edge` class. In addition, the linked list implemented previously will be imported

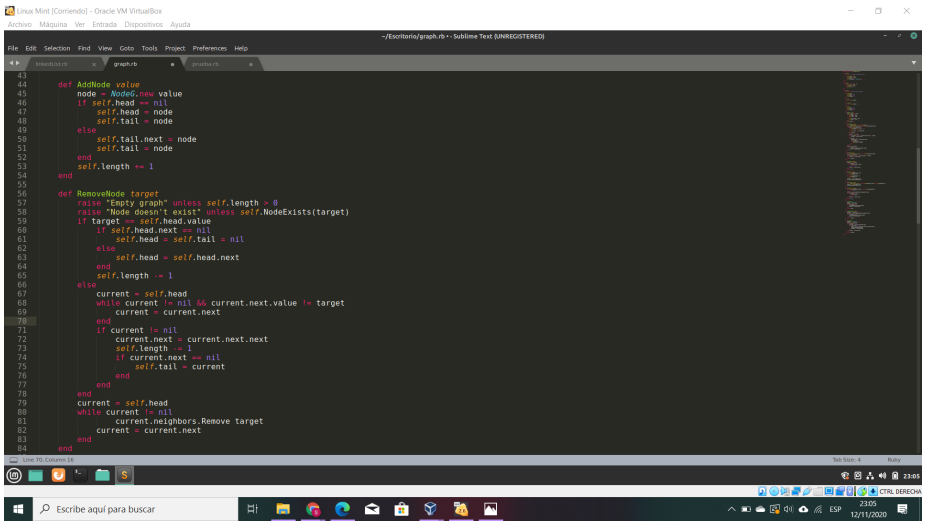
The `NodeG` definition has three attributes: `value`, `next` and `neighbors`. The first contains the value or identifier of one of the nodes of the graph. The second is a pointer that will be used to link it to the next node in the graph. The third is a linked list that contains all the neighbors that the node is related to.

The definition of `Edge` has two attributes: `from` and `to`, which represents the end points of a directed edge. If our graph is not directed then we have to add the inverted edge as well.



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

- Method: `IsEmpty`
  Returns `true` if the graph is empty, otherwise returns false.

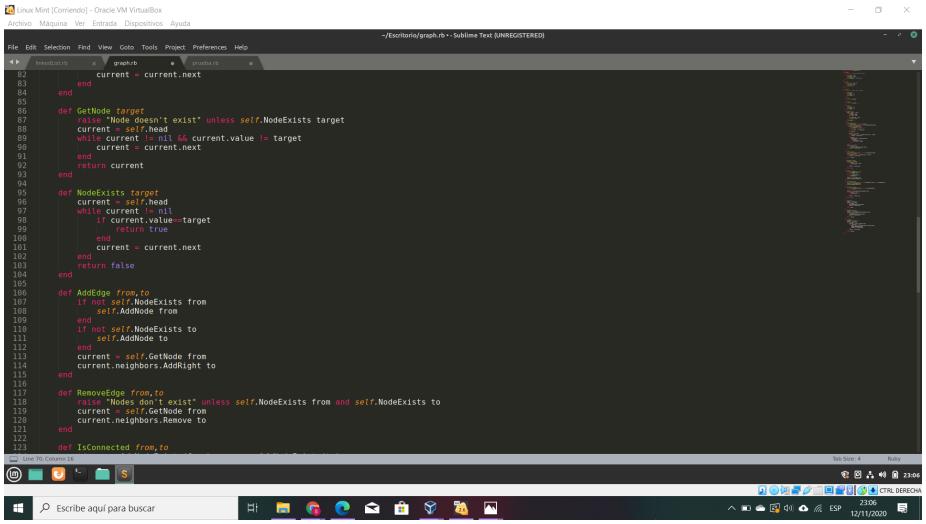


```
def AddNode value
    node = NodeG.new value
    if self.head == nil
        self.head = node
        self.tail = node
    else
        self.tail.next = node
        self.tail = node
    end
    self.length += 1
end

def RemoveNode target
    raise "Empty graph" unless self.length > 0
    raise "Node doesn't exist" unless self.NodeExists(target)
    if target == self.head.value
        if self.head.next == nil
            self.head = self.tail = nil
        else
            self.head = self.head.next
        end
        self.length -= 1
    else
        current = self.head
        while current != nil && current.next.value != target
            current = current.next
        end
        if current != nil
            current.next = current.next.next
            self.length -= 1
            if current.next == nil
                self.tail = current
            end
        end
    end
    current = self.head
    while current != nil
            current.neighbors.Remove target
        current = current.next
    end
end
```

- Method: `AddNode`
  Add a node to the graph.

- Method: `RemoveNode`
  Removes a node from the graph by its value. It also remove all the edges that contains at least one endpoint with these node.

- Method: `Clear`
  Deletes all the information from the graph, leave it empty.

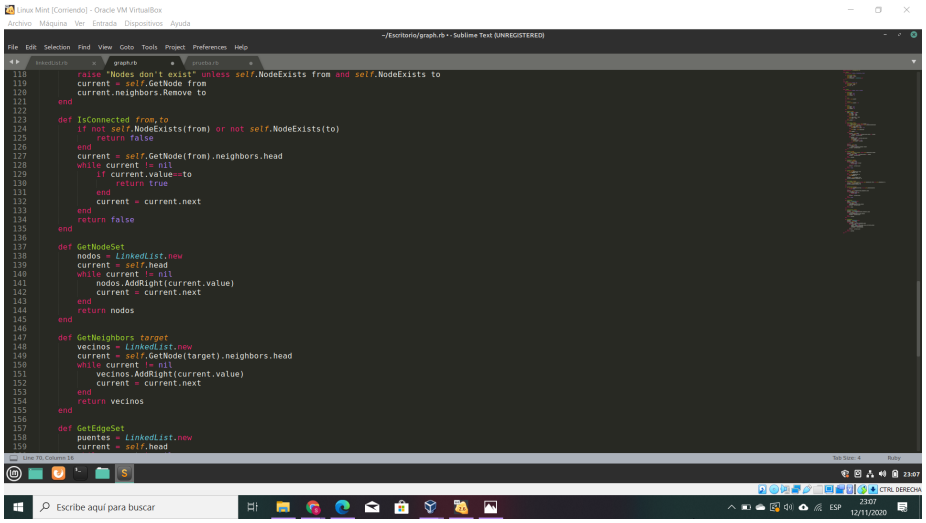


- Method: `GetNode`
  Looks up a node in the neighbors list to the graph and returns a pointer to that node.

- Method: `NodeExists`
  Return `true` if a node with the give value exists, otherwise returns false.

- Method: `AddEdge`
  Add a border to the graphic. If a node does not exist in the chart, it will be added previously.

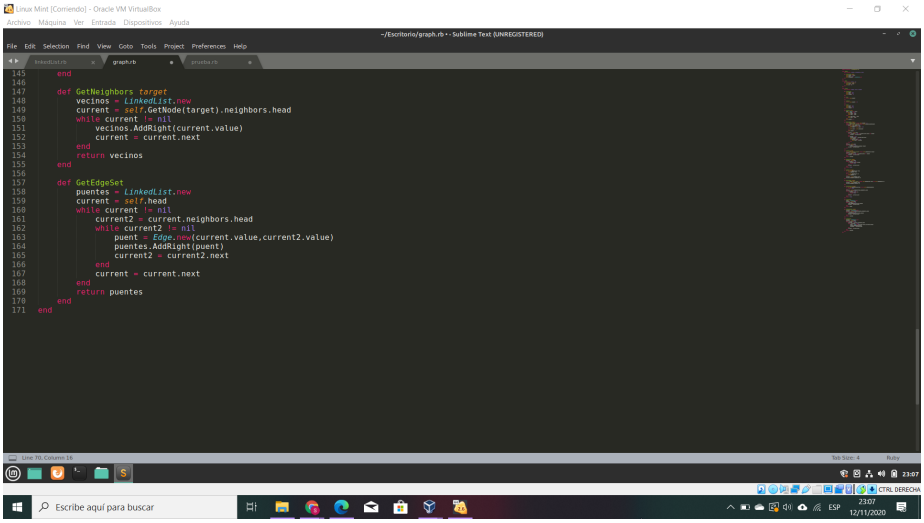
- Method: `RemoveEdge`
  Removes an edge from the graph.



- Method: `IsConnected`
  Returns `true` if the graph contains an edge from `u` to `v`, otherwise returns false.

- Method: `GetNodeSet`
  Returns the list of all nodes in the graph.

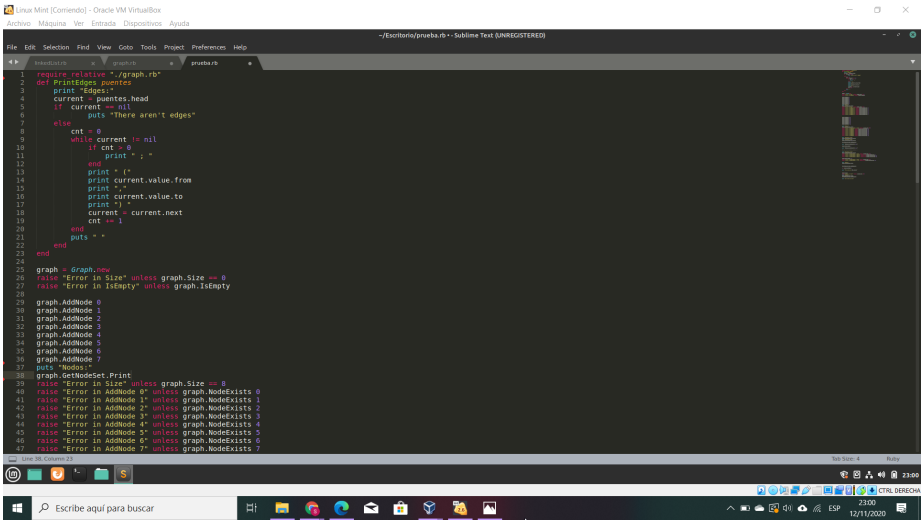
- Method: `GetNeighbors`
  Returns the list of nodes that are neighbors of the specified node.



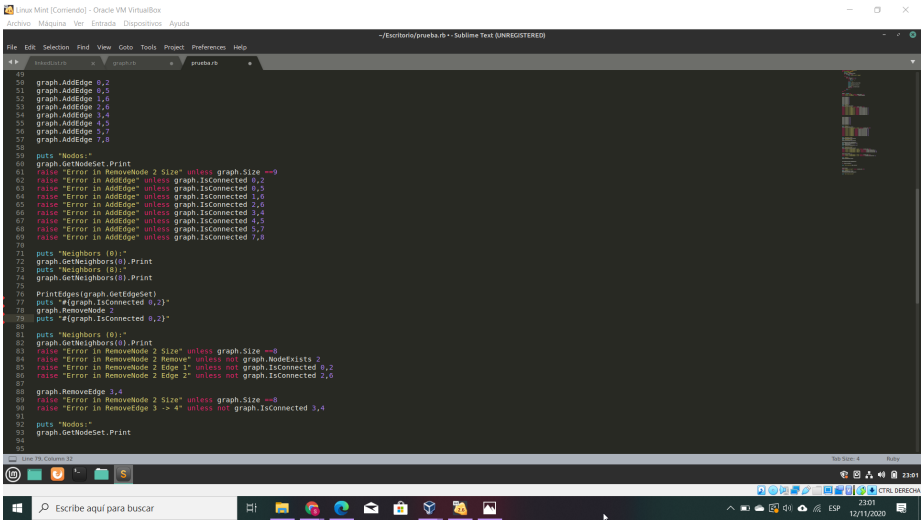
- Method: `GetEdgeSet`
  Returns the list of all edges (tuple form) in the graph.

2.3 Ruby example program

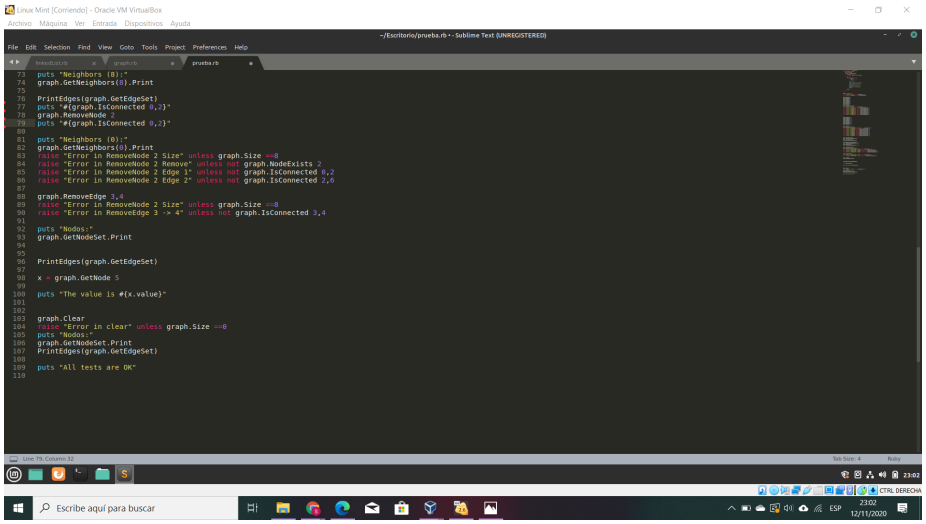
To test that the implementation of the graph is correct, asserts will be used (in the case of Ruby it is `raise`) throughout the entire program. First, a function has been created that allows me to print the list of edges that I get when calling the `GetNeighbors` method, and that when the graph does not have edges it still prints `There aren't edges`. We start the program by creating a new graph and check that it is empty with the `Size` and `IsEmpty` methods. Then we add 8 nodes and print the list of nodes that is obtained with the `GetNodeSet` method, we proceed to verify that these nodes exist in the graph by calling `Node Exists`.



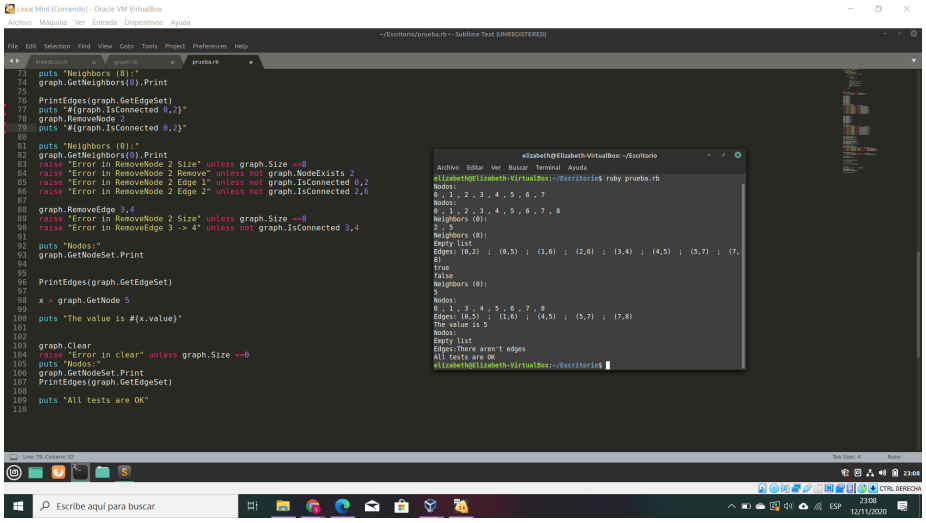
Next we add egdes even though those nodes do not exist previously, we print the nodes again and check the added edges with the `IsConnected` method. We will also call `GetNeighbors` to print the neighbors of some specific nodes, in addition we will print the existing edges. Now we will try `RemoveNode` to remove a node and we will make sure that the edges that were leaving or going to the node are also removed. Then we will remove an edge by calling `RemoveEdge` and we will check that those nodes are no longer connected.



Finally, print out the remaining edges before using the `Clear` method to leave the graph empty again.

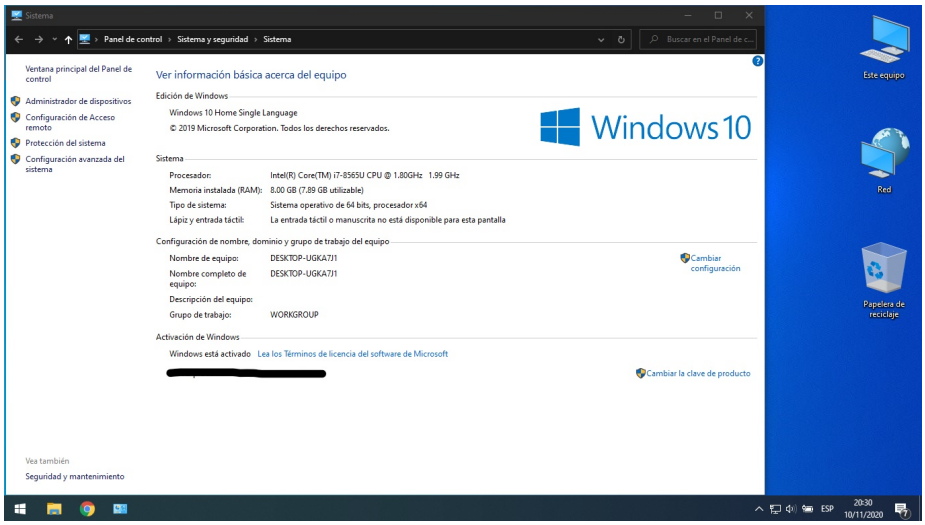


The following image is the execution of the example program.

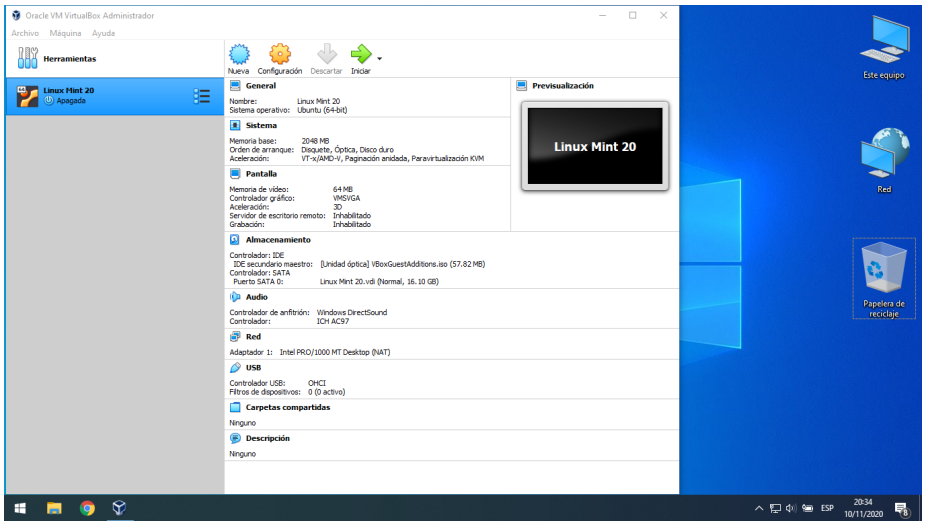


3 Saras Rivera, André Edgardo

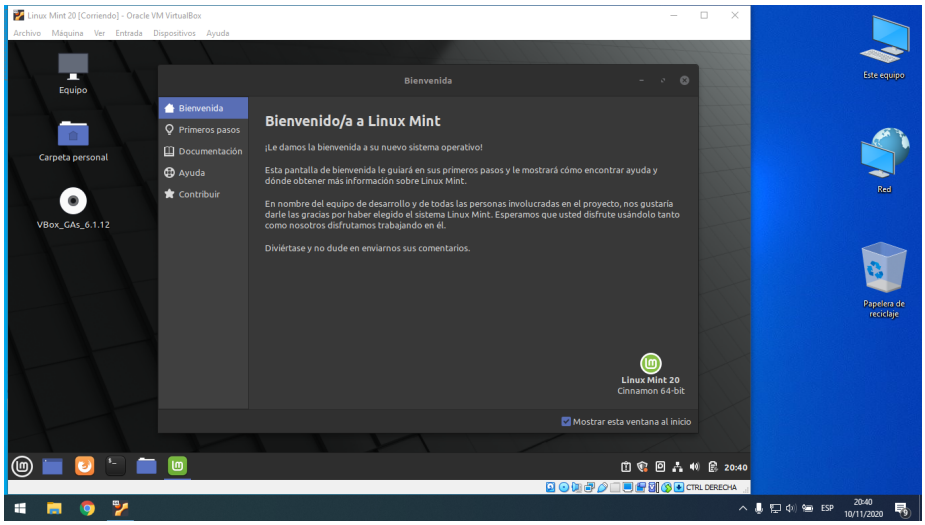
3.1 Resources and Oracle VM VirtualBox parameters



Computer resources.



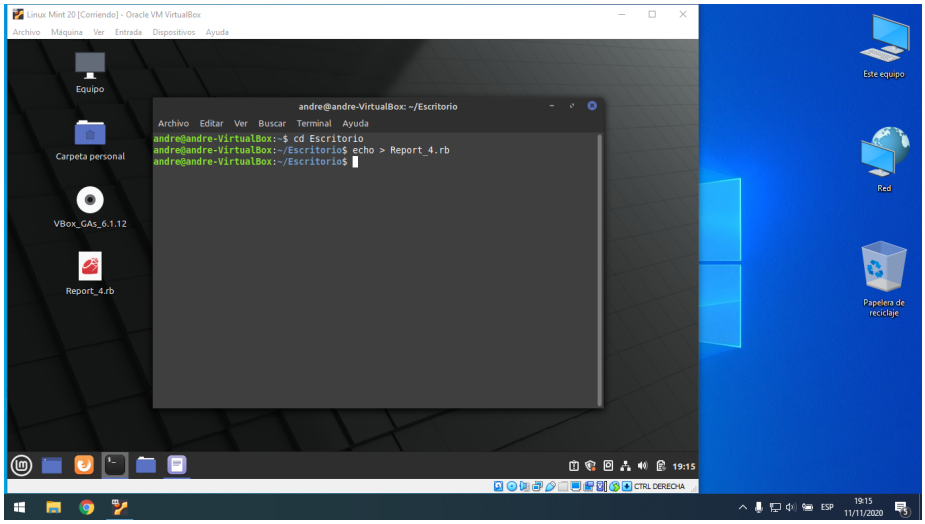
VirtualBox parameters.



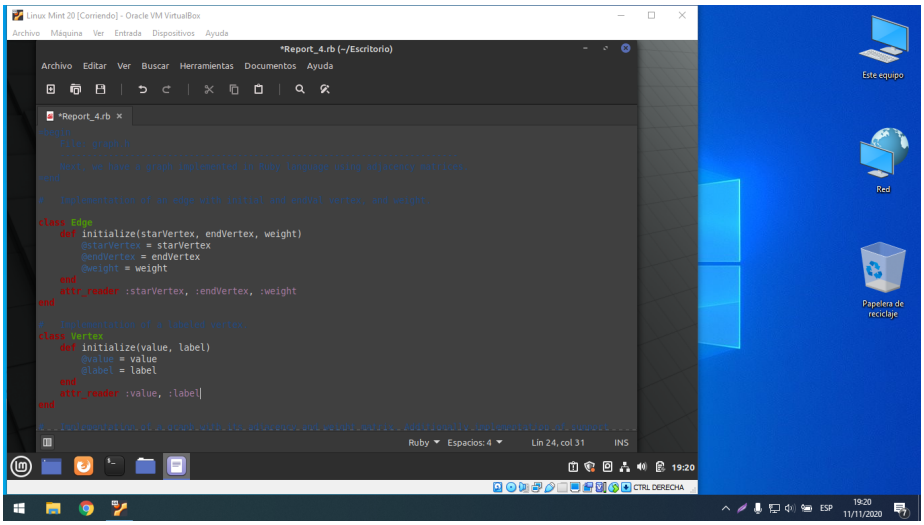
Linux Mint desktop.

3.2 Graph implementation using adjacency matrix

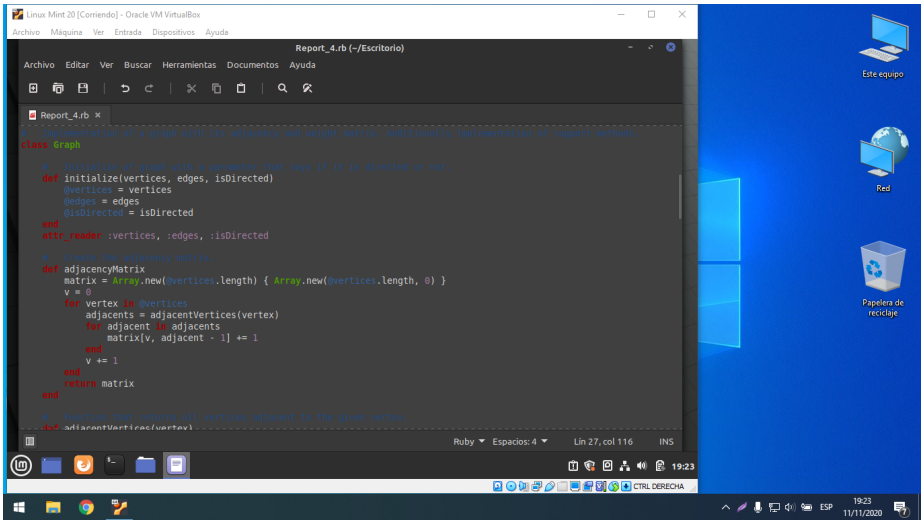
To start working create the file where the TAD will be encoded.



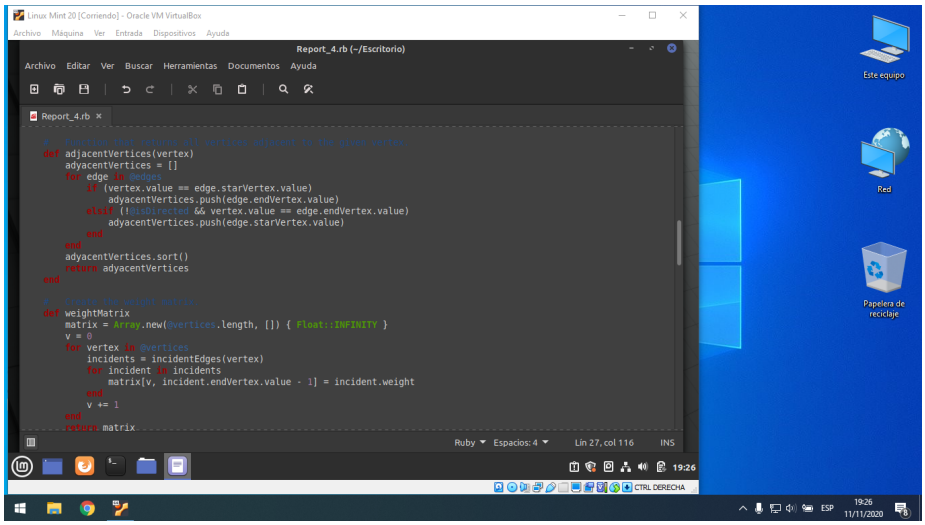
Within the file `.rb` the header is created and then the elements of the graph (edge and vertex) are implemented.

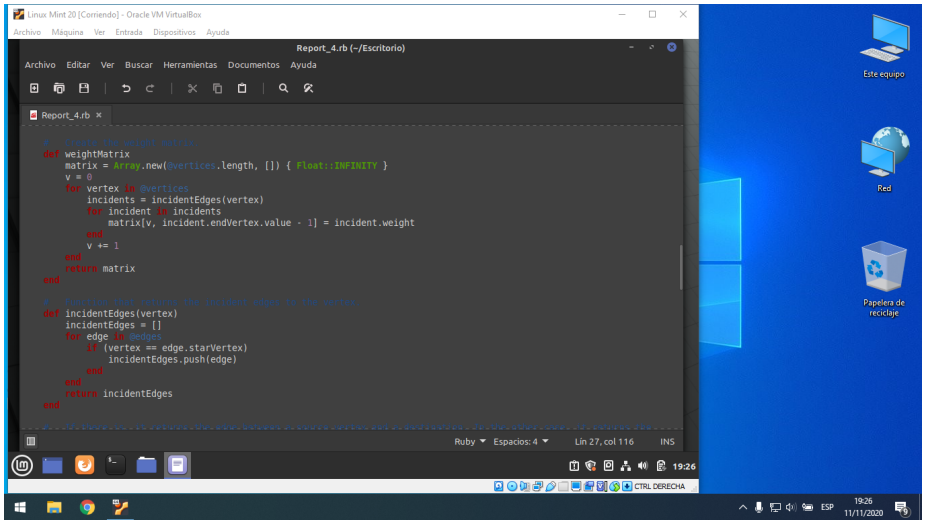


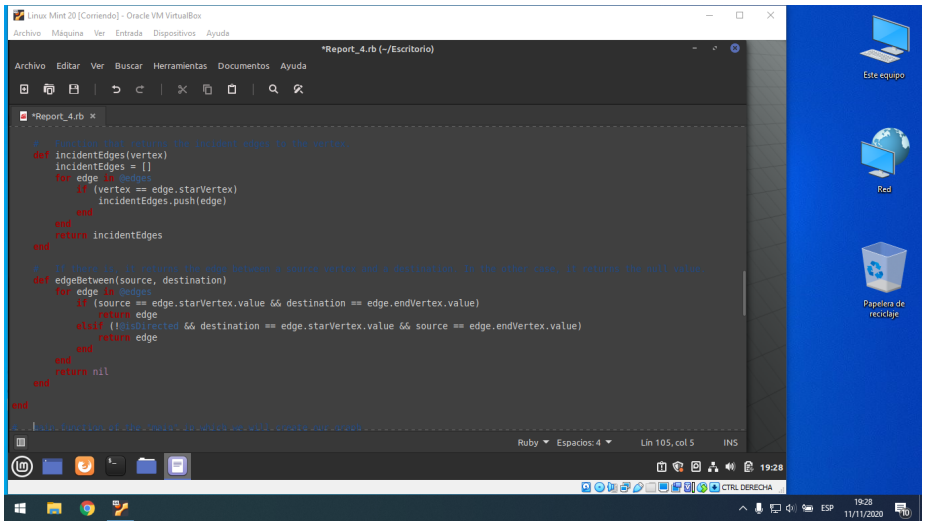
Within the implementation of the graph the main thing is to initialize and create the methods to form the adjacency matrix and the weight matrix (which uses the incidences), consequently we will create the support methods.

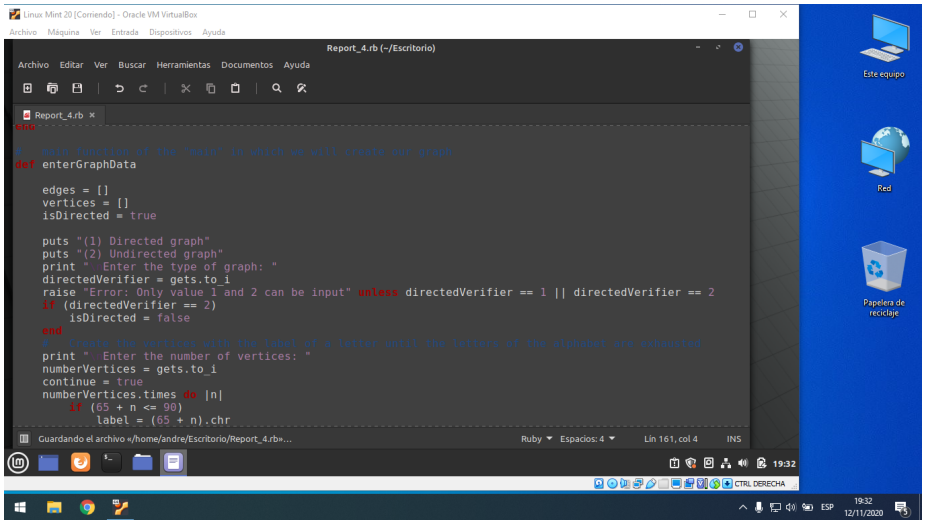


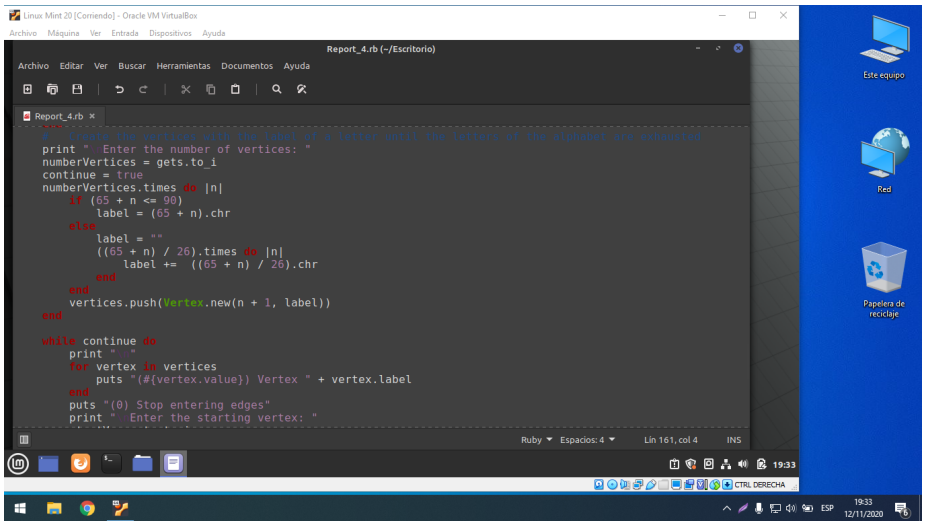
Now, the function that will be in the main will be created and it will ask for the data and fill the adjacency matrix. It is important to mention that the vertices will have a label that is a letter, if the number of letters is finished it will continue with vertices without a label, but you can place any other.

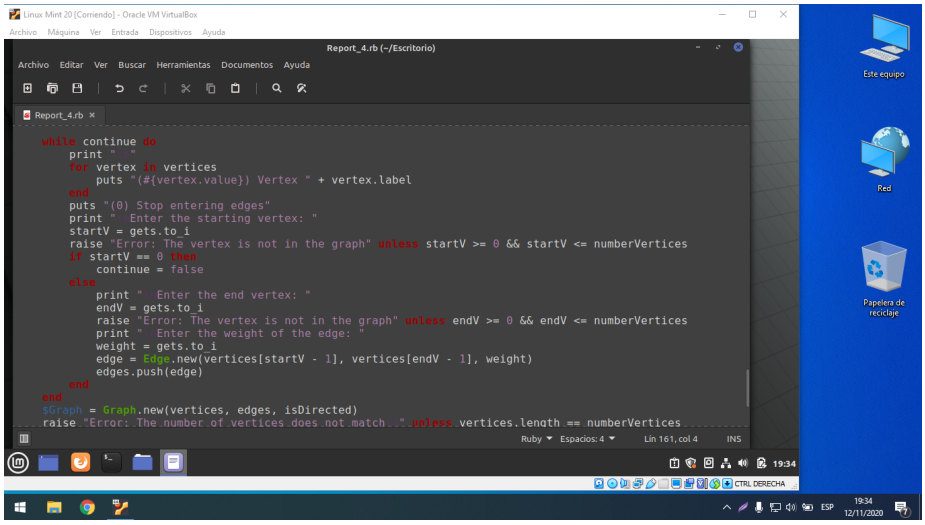

Linux Mint 20 [Corriendo] - Oracle VM VirtualBox
Archivo Máquina Ver Entrada Dispositivos Ayuda
Report_4.rb (~/Escritorio)
Archivo Editar Ver Buscar Herramientas Documentos Ayuda
Report_4.rb
Function that returns all vertices adyacent to the given vertex.
def adjacentVertices(vertex)
adyacentVertices = []
for edge in @edges
if (vertex.value == edge.starVertex.value)
adyacentVertices.push(edge.endVertex.value)
elsif (!@isDirected && vertex.value == edge.endVertex.value)
adyacentVertices.push(edge.starVertex.value)
end
end
adyacentVertices.sort()
return adyacentVertices
end
Create the weight matrix
def weightMatrix
matrix = Array.new(@vertices.length, []) { Float::INFINITY }
v = 0
for vertex in @vertices
incidents = incidentEdges(vertex)
for incident in incidents
matrix[v, incident.endVertex.value - 1] = incident.weight
end
v += 1
end
return matrix
Ruby Espacios: 4 Lín 27, col 116 INS
19:26
CTRL DERECHA
Este equipo
Red
Papelera de reciclaje
ESP
19:26
11/11/2020


Linux Mint 20 [Corriendo] - Oracle VM VirtualBox
Archivo Máquina Ver Entrada Dispositivos Ayuda
Report_4.rb (~/Escritorio)
Archivo Editar Ver Buscar Herramientas Documentos Ayuda
Report_4.rb
Create the weight matrix
def weightMatrix
matrix = Array.new(@vertices.length, []) { Float::INFINITY }
v = 0
for vertex in @vertices
incidents = incidentEdges(vertex)
for incident in incidents
matrix[v, incident.endVertex.value - 1] = incident.weight
end
v += 1
end
return matrix
end
Function that returns the incident edges to the vertex.
def incidentEdges(vertex)
incidentEdges = []
for edge in @edges
if (vertex == edge.starVertex)
incidentEdges.push(edge)
end
end
return incidentEdges
end
Ruby Espacios: 4 Lín 27, col 116 INS
19:26
CTRL DERECHA
Este equipo
Red
Papelera de reciclaje
ESP
19:26
11/11/2020

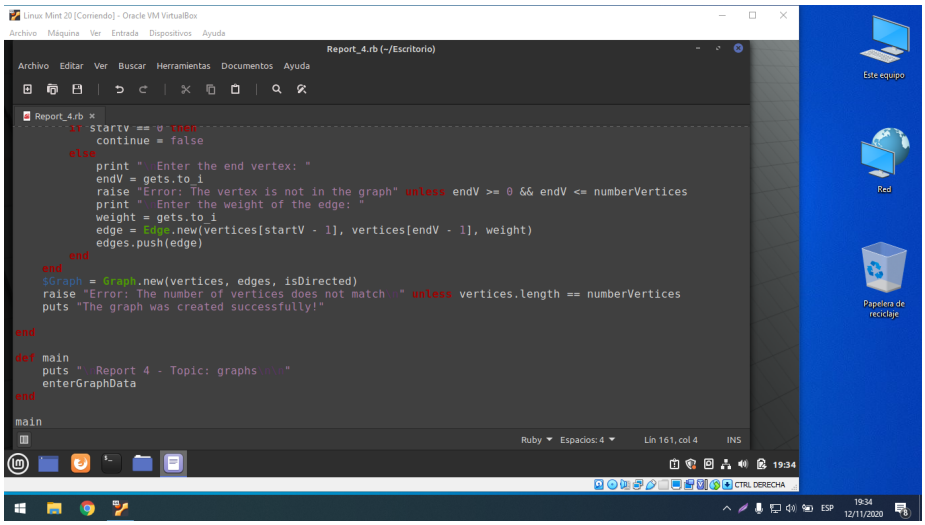

Linux Mint 20 [Corriendo] - Oracle VM VirtualBox
*Report_4.rb (~/Escritorio)
Archivo Editar Ver Buscar Herramientas Documentos Ayuda
*Report_4.rb
Function that returns the incident edges to the vertex
def incidentEdges(vertex)
incidentEdges = []
for edge in edges
if (vertex == edge.starVertex)
incidentEdges.push(edge)
end
end
return incidentEdges
end
If there is, it returns the edge between a source vertex and a destination. In the other case, it returns the null value.
def edgeBetween(source, destination)
for edge in edges
if (source == edge.starVertex.value && destination == edge.endVertex.value)
return edge
elsif (!@isDirected && destination == edge.starVertex.value && source == edge.endVertex.value)
return edge
end
end
return nil
end
end
Ruby Espacios: 4 Lín 105, col 5 INS
Este equipo
Red
Papelera de reciclaje
CTRL DERECHA
ESP
19:28
11/11/2020


Linux Mint 20 [Corriendo] - Oracle VM VirtualBox
Report_4.rb (~/Escritorio)
Archivo Editar Ver Buscar Herramientas Documentos Ayuda
Report_4.rb
end
Main function of the "main" in which we will create our graph
def enterGraphData
edges = []
vertices = []
isDirected = true
puts "(1) Directed graph"
puts "(2) Undirected graph"
print " Enter the type of graph: "
directedVerifier = gets.to_i
raise "Error: Only value 1 and 2 can be input" unless directedVerifier == 1 || directedVerifier == 2
if (directedVerifier == 2)
isDirected = false
end
Create the vertices with the label of a letter until the letters of the alphabet are exhausted
print " Enter the number of vertices: "
numberVertices = gets.to_i
continue = true
numberVertices.times do |n|
if (65 + n <= 90)
label = (65 + n).chr
Guardando el archivo «/home/andre/Escritorio/Report_4.rb»...
Ruby Espacios: 4 Lín 161, col 4 INS
Este equipo
Red
Papelera de reciclaje
CTRL DERECHA
ESP
19:32
12/11/2020



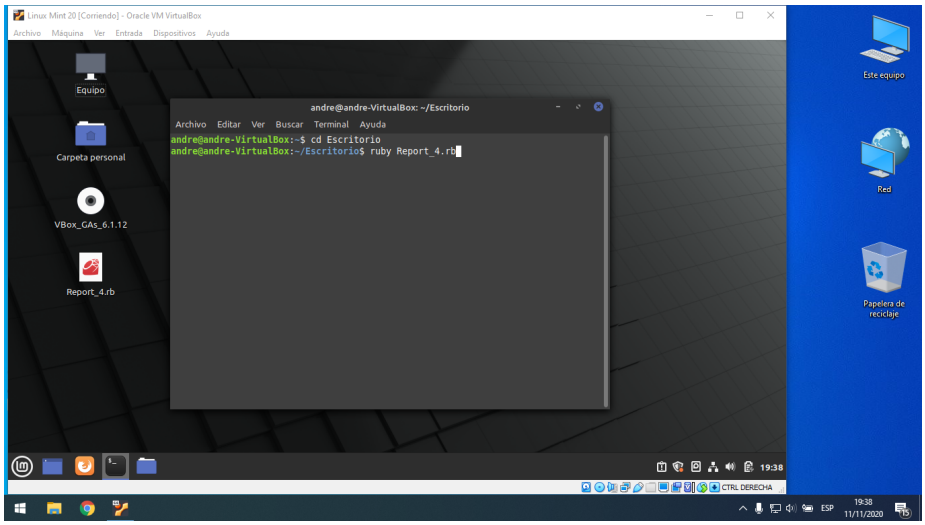


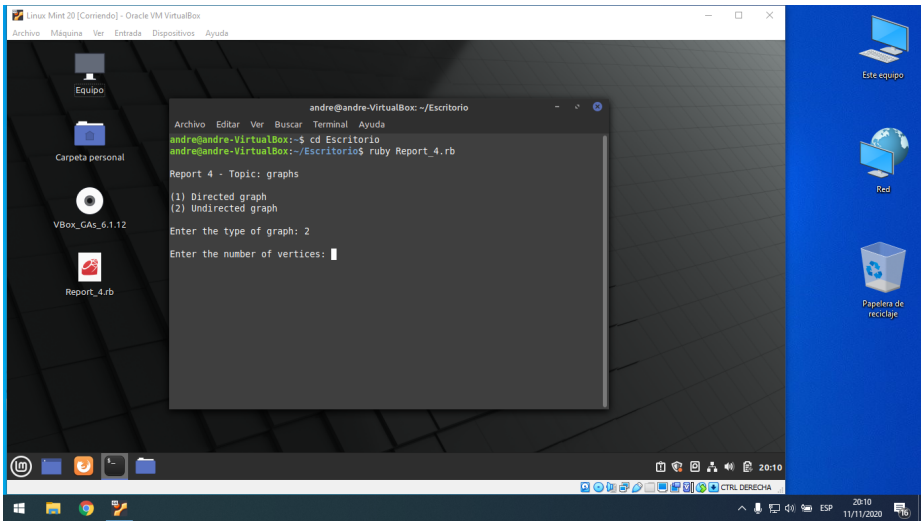
Finally, the main function is created.



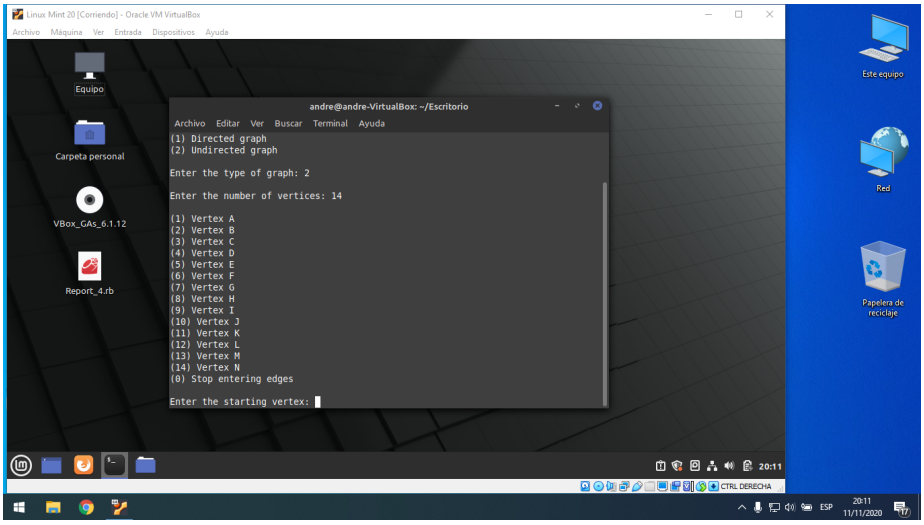
3.3 Ruby example program

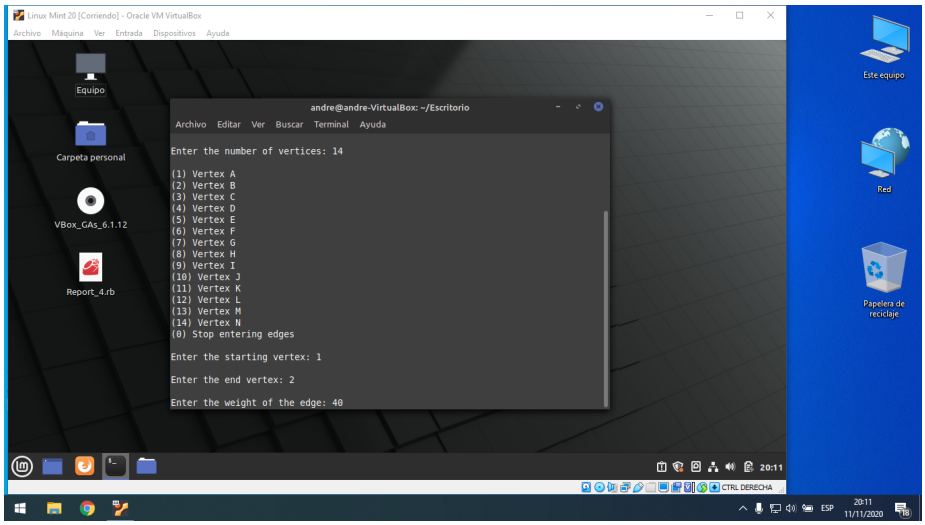
Let's test the program. First of all, we have to indicate whether it is a directed graph or not



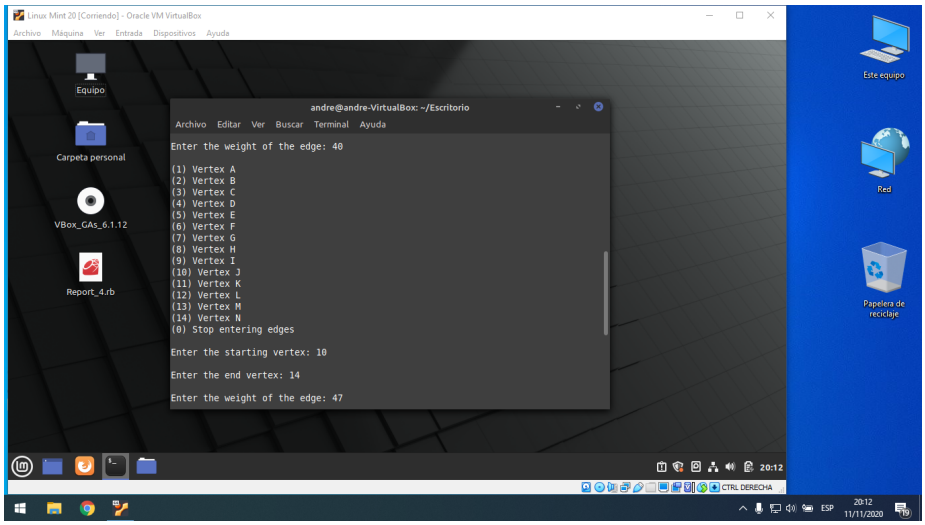


When entering the number of vertices, these are created automatically with their respective label.

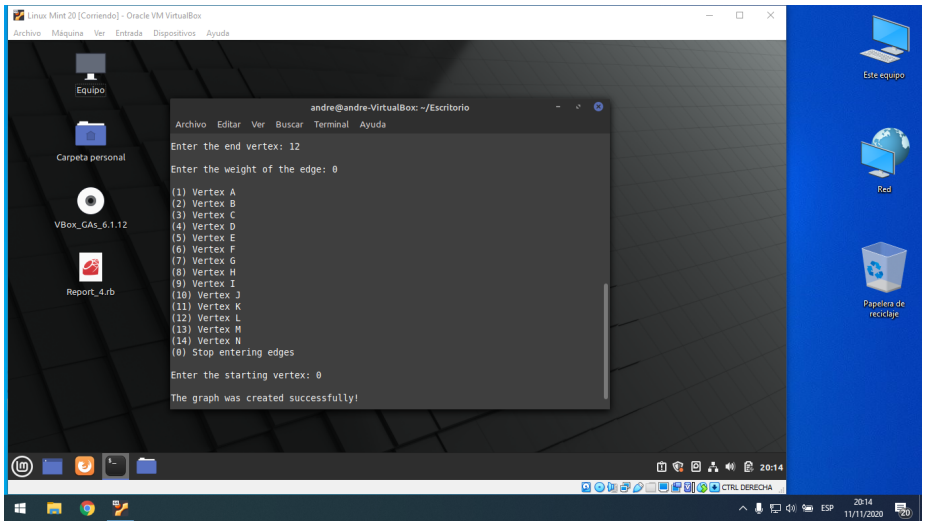




Then, we enter the edges one by one, placing their beginning, end and weight vertex (it can have 0 or negative weight without any problem).



To finish the process, we enter the value of 0 as the initial node. If everything has been correct, then the message of successful creation of the graph will appear on screen.


Linux Mint 20 [Corriendo] - Oracle VM VirtualBox
Archivo Máquina Ver Entrada Dispositivos Ayuda
Equipo
Carpeta personal
VBox_GAs_6.1.12
Report_4.rb
andre@andre-VirtualBox: ~/Escritorio
Archivo Editar Ver Buscar Terminal Ayuda
Enter the end vertex: 12
Enter the weight of the edge: 0
(1) Vertex A
(2) Vertex B
(3) Vertex C
(4) Vertex D
(5) Vertex E
(6) Vertex F
(7) Vertex G
(8) Vertex H
(9) Vertex I
(10) Vertex J
(11) Vertex K
(12) Vertex L
(13) Vertex M
(14) Vertex N
(0) Stop entering edges
Enter the starting vertex: 0
The graph was created successfully!
20:14
CTRL DERECHA
Este equipo
Red
Papelera de reciclaje
ESP
20:14
11/11/2020