

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

FACULTAD DE CIENCIAS E INGENIERÍA

ESPECIALIDAD DE INGENIERÍA INFORMÁTICA



Informe sobre el desarrollo en Python 3 de una aplicación que procesa el grafo de rutas de los domicilios de los integrantes a la PUCP empleando el algoritmo de Kruskal

Curso:

ESTRUCTURA DE DATOS Y PROGRAMACIÓN METÓDICA

Docente:

VIKTOR KHLEBNIKOV

San Miguel, Mayo del 2018

INTEGRANTES Y APORTES

Jhair Daniel Guzman Tito 20163275

- Investigación sobre el uso de OpenStreetMAp
- Investigación sobre el uso de la librería Matplotlib
- Implementación de la aplicación
- Conclusiones

Gilmer Wilder Cabrera García 20156075

- Introducción
- Investigación sobre el algoritmo de Kruskal y su implementación en Python
- Implementación de la aplicación
- Conclusiones

Andrea Lucia Reyes Burga 20160026

- Investigación sobre el uso de MatplotLib
- Investigación sobre el uso de OpenSteetMap
- Implementación de la aplicación
- Conclusiones

INTRODUCCIÓN

El presente informe trata sobre la creación de una aplicación que procesa el grafo de las rutas de los domicilios de cada uno de los integrantes a la PUCP mediante el uso del algoritmo de Kruskal. En primer lugar, se utilizó la herramienta OpenStreetMap para obtener información geográfica. Esto realizó con la finalidad de acceder a data sobre los nodos como el id, latitud y longitud para poder crear la representación de las aristas y sus respectivos pesos (longitud en kilómetros). En este caso se puede apreciar que se está modelando una situación de la vida real como es la representación de una red de calles.

Para hacer posible lo mencionado anteriormente fue necesario la creación de procedimientos en Python que permitan filtrar información útil para la creación del grafo que será procesado. Ello implicó realizar una investigación sobre el funcionamiento y utilidades de la herramienta OpenStreetMap.

En segundo lugar; se realizó la implementación del algoritmo de Kruskal en Python. De igual manera, fue necesario realizar una investigación sobre el funcionamiento y la utilidad que nos brinda este algoritmo. Básicamente, este algoritmo nos permite generar un árbol de coste mínimo a partir de un grafo. Es decir; nos permite crear un subgrafo, el cual es un árbol sin ciclos, que almacena las aristas con coste mínimo.

En tercer lugar; luego de la obtención del árbol de coste mínimo se procedió a la implementación de una aplicación la cual consiste en representar de manera gráfica el árbol de coste mínimo. Para esto se hizo uso de una librería de Python **matplotlib**, la cual permite generar gráficos de acuerdo a los requerimientos de programador.

ÍNDICE

1. Algoritmo kruskal
2. Información sobre Matplotlib
3. Información sobre Pytplot
4. Información sobre OpenStreetMap
5. Implementación
6. Aplicación
7. Conclusiones
8. Bibliografía

ALGORITMO DE KRUSKAL

El algoritmo de kruskal nos permite generar un árbol de coste mínimo a partir de un grafo con aristas valoradas. Un árbol de coste mínimo se puede entender como un subgrafo que no contiene ciclos. Este nos permite reducir la complejidad de un grafo con la finalidad de trabajar de manera más sencilla con el árbol generado. Cabe mencionar que el subgrafo que se genera es conexo. Es decir; todos sus no nodos se encuentran interconectados a través de aristas pero tienen como principal característica que las aristas poseen los costos mínimos correspondientes. En cuanto a los costos mínimos, esto se puede entender como los valores que tienes asociadas las aristas, por ejemplo, distancia, tiempo u otro valor que corresponda.

Lo mencionado anteriormente se puede apreciar en la siguiente imagen:

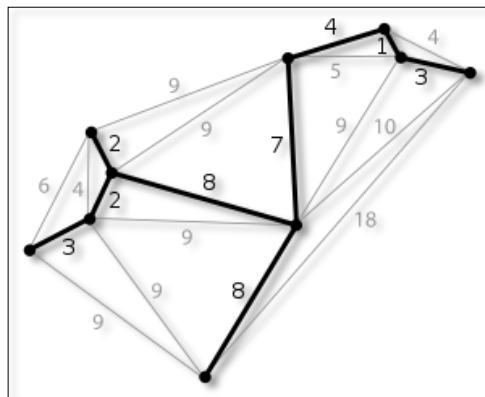


Imagen 1: Grafo y su árbol de coste mínimo

Imagen tomada de: eeNube Programación.

Explicación básica sobre el algoritmo:

Este algoritmo recibe como entrada un grafo que contiene todos los nodos y las aristas que conforman un grafo. A partir de ello; se ordena las aristas (u,v) de acuerdo a su coste (longitud) y posteriormente se va tomando aquellas que con menor coste y que no generen ciclos. Cabe mencionar que al finalizar todos los nodos quedan interconectados y por lo tanto forman parte del árbol de coste mínimo.

Aplicación típica:

Esto se puede evidenciar en el diseño de redes telefónicas entre las oficinas de una empresa. Sin embargo; para realizar la interconexión se establecen tarifas diferentes de acuerdo a las oficinas que se desean conectar mediante la línea telefónica. El objetivo es establecer las conexiones que se deben realizar para generar un mínimo coste total.

INFORMACIÓN SOBRE MATPLOTLIB

Es una librería que produce gráficos 2D que posee Python, ella produce figuras de calidad en una variedad de formatos e interactivos ambientes a través de la plataforma. Con esta librería se pueden realizar gráficos, histogramas, gráfico de barras, colocar imágenes, gráficos circulares, gráficos de líneas.



Proceso de Instalación:

Para instalar la librería es necesario contar con pip y para poder observar los gráficos es necesario descargar el paquete tkinter que es una interfaz gráfica.

Los comandos que se utilizaron para este proceso son los siguientes:

```
sudo python3 get-pip.py
sudo apt-get install python3-matplotlib
apt-get install python-tk
```

Uso de Matplotlib en la aplicación:

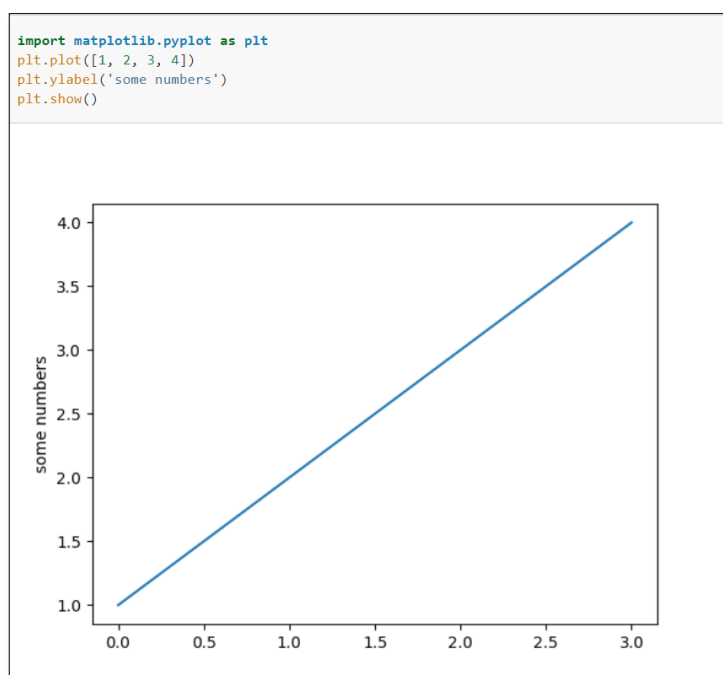
De las diversas herramientas de esta librería, para este informe utilizamos pyplot. Pyplot contiene funciones que ayudan a generar diferentes tipos de gráficos de manera rápida como se podrá observar en la siguiente imagen.

Function	Description
<code>acorr</code>	Plot the autocorrelation of x .
<code>angle_spectrum</code>	Plot the angle spectrum.
<code>annotate</code>	Annotate the point xy with text s .
<code>arrow</code>	Add an arrow to the axes.
<code>autoscale</code>	Autoscale the axis view to the data (toggle).
<code>axes</code>	Add an axes to the current figure and make it the current axes.
<code>axhline</code>	Add a horizontal line across the axis.
<code>axhspan</code>	Add a horizontal span (rectangle) across the axis.
<code>axis</code>	Convenience method to get or set axis properties.
<code>axvline</code>	Add a vertical line across the axes.
<code>axvspan</code>	Add a vertical span (rectangle) across the axes.
<code>bar</code>	Make a bar plot.
<code>barbs</code>	Plot a 2-D field of barbs.
<code>barh</code>	Make a horizontal bar plot.
<code>box</code>	Turn the axes box on or off.
<code>boxplot</code>	Make a box and whisker plot.
<code>broken_barh</code>	Plot a horizontal sequence of rectangles.
<code>cla</code>	Clear the current axes.
<code>clabel</code>	Label a contour plot.
<code>clf</code>	Clear the current figure.
<code>clim</code>	Set the color limits of the current image.
<code>close</code>	Close a figure window.

INFORMACIÓN SOBRE PYPLOT

Matplotlib.pyplot es una colección de diferentes funciones de estilo comando que hacen que matplotlib trabaje como si fuera MATLAB. Cada función de pyplot hace algún cambio a las figuras como, por ejemplo, crear líneas en un área del gráfico, decorar el gráfico con etiquetas, entre otras. Entonces, haciendo uso pyplot podemos crear gráficos en los que se ubican puntos según su valor x e y. Además se puede utilizar diferentes formatos en el gráfico.

En la siguiente imagen, se verá un pequeño ejemplo de cómo funciona.



El uso de pyplot en la aplicación:

Para el informe, se realizó un gráfico en el que se colocaba cada punto con su latitud como “x” y su longitud como “y”. Así mismo, se muestra como cada punto está unido en el gráfico mediante líneas. Esto se realizó usando la función plot() que une dos puntos en forma de líneas y se puede ver en la imagen de color amarillo. Luego, una vez realizado Kruskal, se realiza el mismo proceso que se uso para imprimir las líneas que unen a los puntos; pero usando las aristas que se obtenían de Kruskal. Por lo que se obtienen menos líneas y estas se pueden observar de color magenta. Así mismo editamos algunas etiquetas del gráfico con las funciones xlabel(), ylabel().

INFORMACIÓN SOBRE OPEN STREET MAP

Open Street Map es un proyecto que ayuda a crear mapas, donde los datos que se muestran son de uso libre para los usuarios y, uno puede colaborar con este proyecto si tienen alguna información que no se encuentra en el mapa.

Utilizar un archivo XML en Python:

Implementación de un árbol elemental API (Interfaz de programación de aplicaciones)

El módulo [xml.etree.ElementTree](#) implementa un simple y eficiente API para el análisis y creación de información XML.

Elementos de un árbol elemental

La forma más sencilla de representar la información de un XML es mediante un árbol. Un árbol elemental tiene dos clases: `ElementTree` que representa todo el documento XML como un árbol y `Element` que representa un solo nodo en este árbol. Las interacciones con todo el documento son hechas usualmente en un nivel de `ElementTree`. Y las interacciones con un solo elemento de XML y sus subelementos son hechos en el nivel de `Element`.

Importación de un documento XML:

De la siguiente forma se importa información de un documento XML en Python:

```
import xml.etree.ElementTree as ET
tree = ET.parse('country_data.xml')
root = tree.getroot()
```

Búsqueda de datos dentro del documento:

`Element` tiene algunos métodos muy útiles que ayudan a iterar recursivamente sobre todo el sub-árbol

Por ejemplo:

`Element.findall()`, en nuestro programa de Python utilizamos en varias partes este método. Este método encuentra solo elementos con una etiqueta, los cuales son hijos directos del elemento en el que nos encontramos.

Archivos XML:

XML es un formato universal para almacenar y manejar datos y documentos estructurados. Estos tipos de archivos trabajan con etiquetas "<etiqueta />", las cuales poseen información sobre sí mismas y también funcionan como contenedores para otras etiquetas, también conocidas como hijas (la etiqueta que las engloba sería el padre). Estas etiquetas llevan los nombres relacionados a la información que guardan.

Para poder trabajarlos dentro de Python, la estructura de un .xml puede ser tratada como un árbol general y con la ayuda de la librería xml.ElementTree.

Archivo .OSM:

Un archivo .osm se puede obtener desde la página web de OpenStreetMap. Los datos son almacenados en base a NODOS(node), CAMINOS(way) y RELACIONES(relation) de la siguiente manera:

La etiqueta <osm> almacena toda la información, y es la que se utiliza como raíz de todo el árbol general. La primera etiqueta hija es <bounds "atributos" >, la cual indica las latitudes máximas y mínimas, y las longitudes máxima y mínima de toda la muestra hecha que se exportó desde la página de OpenStreetMap.

A continuación de esta, se encuentran las etiquetas <node "atributos" /> los cuales almacenan en sus atributos su ID, la latitud y la longitud como coordenadas de este nodo, y otros datos más que informan sobre cómo y quién registró el nodo.

Seguido de los nodos, se encuentran las etiquetas <ways ... >, las cuales poseen etiquetas hijas que son referencias a los nodos que conforman un camino. Además, almacena etiquetas <tag ... /> que almacenan información sobre qué representa ese camino (una autopista, el perímetro de un parque, etc.), el nombre y estado de este, etc.

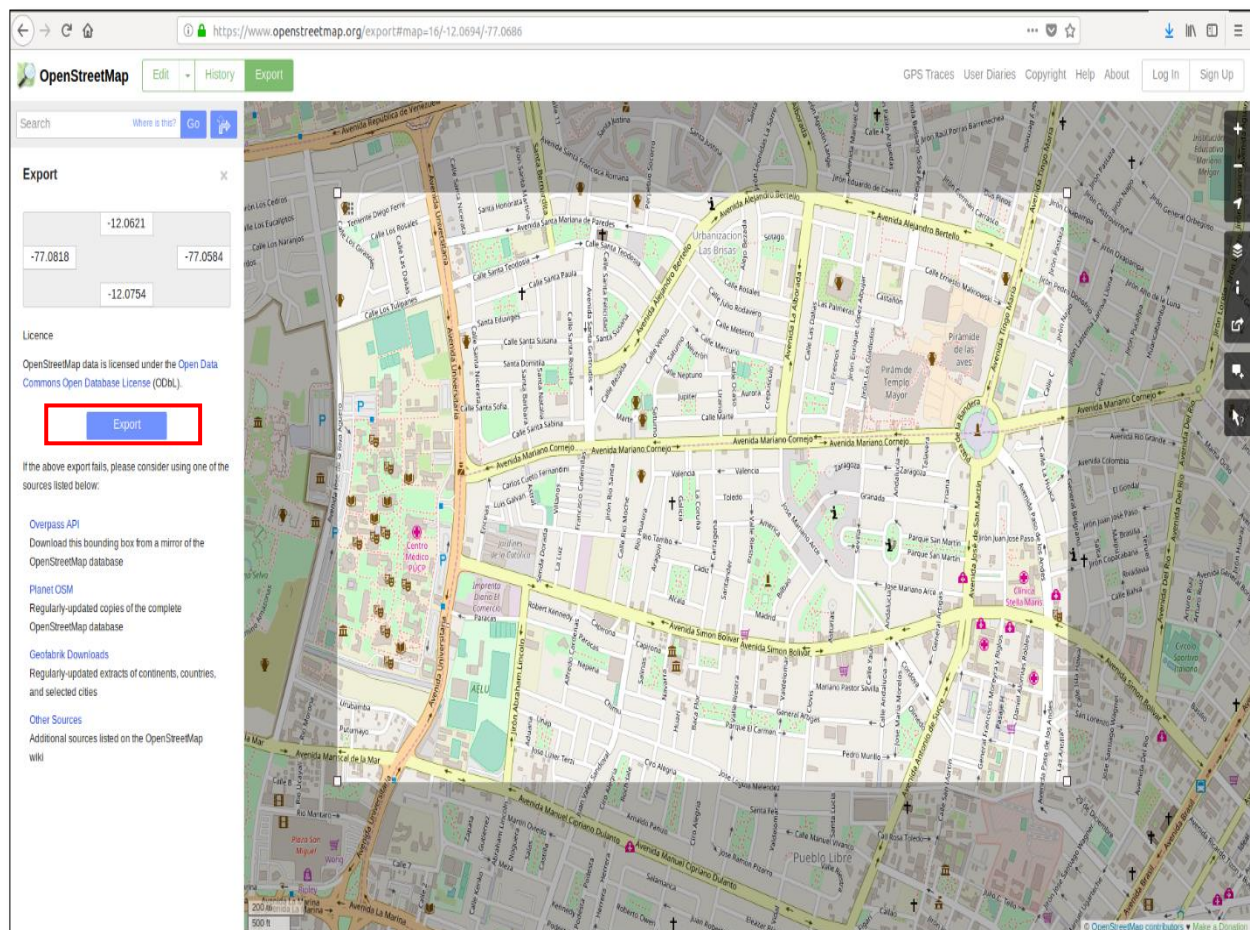
Por último, se encuentran las etiquetas <relation>, las cuales engloban varias etiquetas hijas <member ... />. Estas etiquetas "relation" ayudan a agrupar nodos y caminos que representan algo en particular como las rutas de un bus, las estaciones y paraderos de este, o como los detalles de caminos y edificios dentro de una universidad, entre otros.

```
<?xml ... ?>
<osm >
  <bounds ... >
    <node ... >
    <node ... >
    <node ... >
    <node ... >
    ...
  <way ... >
    <node ... >
    <node ... >
    <node ... >
    ...
    <tag ... >
    <tag ... >
  <way />
  ...
  <relation ... >
    <member ... >
    <member ... >
    <member ... >
    ...
  <relation />
  ...
</osm />
```

IMPLEMENTACIÓN

1. Obtención de información para la creación del grafo

Para obtener la información se usa OpenStreetMap. Desde su dirección web se puede descargar un archivo OSM, el cual porta los datos sobre las coordenadas de cada calle de región que se seleccione.



2. Depuración de la información obtenida

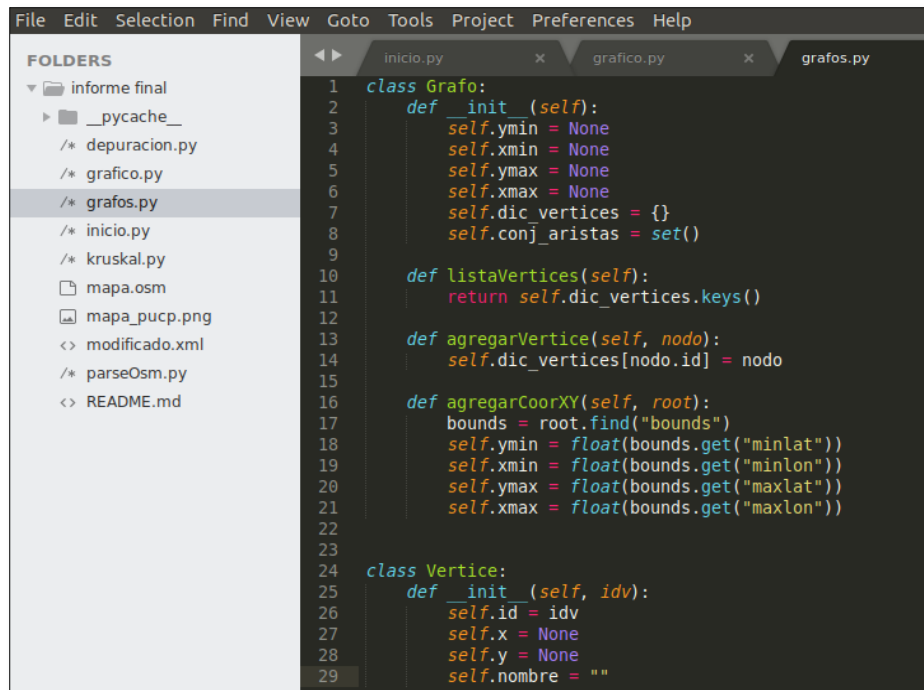
Luego de extraer el archivo OSM, se procesa de la misma manera como se parsea un archivo XML. Sin embargo, este archivo OSM contiene datos que no son útiles para el mapeo de autopistas. Por ello, se realiza una depuración haciendo uso del siguiente código:

```
1
2 import xml.etree.ElementTree as et
3 from grafos import Vertice
4
5
6 def depurar(archivo):
7     tree = et.parse(archivo)
8     root = tree.getroot()
9
10    eliminarRelaciones(root)
11    dict_nodos = {} # se lleva registro de los nodos que existen solo en los caminos
12    eliminarWays(dict_nodos, root)
13    eliminarNodos(dict_nodos, root)
14    tree.write('modificado.xml')
15
16
17
18
19 def eliminarRelaciones(root):
20     for relation in root.findall("relation"):
21         root.remove(relation)
22
23
24
25
26 def eliminarWays(dict_nodos, root):
27     # eliminación de caminos que no sean 'highway'
28     for way in root.findall("way"):
29         """obtenemos datos: autopista(boolean)
30         - si es autopista, nombre(string)
31         - el nombre de la calle"""
32         autopista, nombre = obtenerDatosWay(way)
33         "en caso no sea una autopista, se elimina y continua iterando"
34         if not autopista:
35             root.remove(way)
36             continue
37         else:
38             way.attrib.pop("changeset")
39             way.attrib.pop("timestamp")
40             way.attrib.pop("uid")
41             way.attrib.pop("user")
42             way.attrib.pop("version")
43             way.attrib.pop("visible")
44             # guardamos los nodos de los caminos
45             for nd in way.findall("nd"):
46                 dict_nodos[nd.get("id")] = 0
47
```

```
55
56
57
58 def eliminarNodos(dict_nodos, root):
59     # almacenamiento de todos los nodos en diccionario
60     dict_nodos = {}
61     for nodo in root.findall("node"):
62         dict_nodos[nodo.get("id")] = 0
63     # eliminación de nodos que no se encuentran en los caminos
64     # los que no se eliminan, se borran atributos que no se utilizan
65     for nodo in root.findall("node"):
66         if not nodo.get("id") in dict_nodos:
67             root.remove(nodo)
68         else:
69             nodo.attrib.pop("changeset")
70             nodo.attrib.pop("timestamp")
71             nodo.attrib.pop("uid")
72             nodo.attrib.pop("user")
73             nodo.attrib.pop("version")
74             nodo.attrib.pop("visible")
75
76
77 def obtenerDatosWay(way):
78     autopista = False
79     nombre = ""
80     for tag in way.findall("tag"):
81         # highway no puede ser "*", el cual significa ser una calle cerrada
82         if tag.get("k") == "highway" and tag.get("v") in [
83             'unclassified',
84             'residential',
85             'primary',
86             'secondary',
87             'tertiary',
88             'primary_link',
89             'living_street'
90         ]: autopista = True
91         elif tag.get("k") == "name": nombre = tag.get("v")
92         else: way.remove(tag)
93     return autopista, nombre
94
95
```

3. Implementación de un TAD grafo

Para poder almacenar la información debemos adaptar el TAD grafo de tal manera que pueda guardarse las coordenadas de cada nodo como vértice y la relación entre estos como aristas.

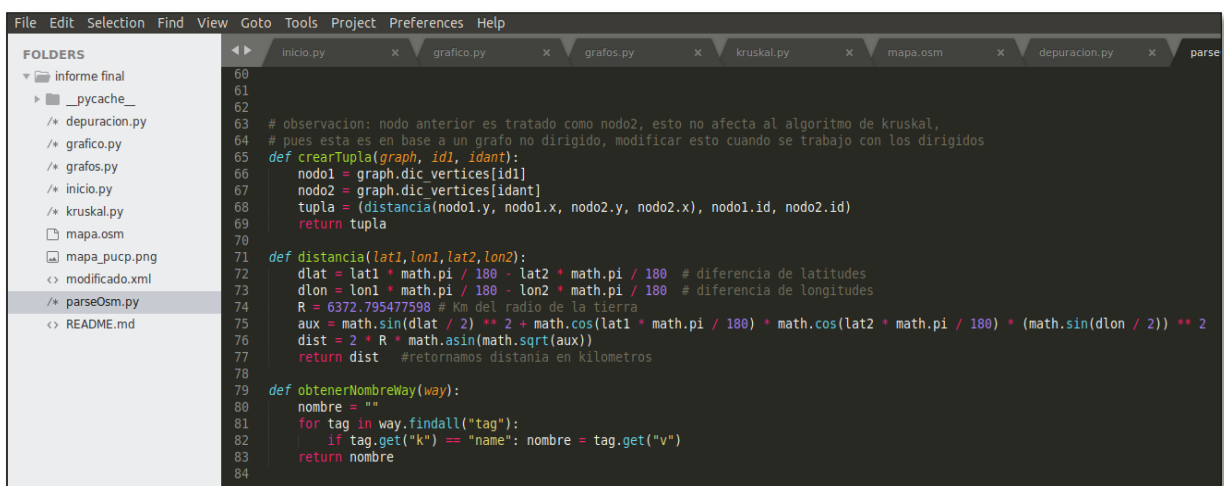


```
File Edit Selection Find View Goto Tools Project Preferences Help
FOLDERS
  informe final
  _pycache_
  /* depuracion.py
  /* grafico.py
  /* grafos.py
  /* inicio.py
  /* kruskal.py
  mapa.osm
  mapa_pucp.png
  <> modificado.xml
  /* parseOsm.py
  <> README.md

1 class Grafo:
2     def __init__(self):
3         self.ymin = None
4         self.xmin = None
5         self.ymax = None
6         self.xmax = None
7         self.dic_vertices = {}
8         self.conj_aristas = set()
9
10    def listaVertices(self):
11        return self.dic_vertices.keys()
12
13    def agregarVertice(self, nodo):
14        self.dic_vertices[nodo.id] = nodo
15
16    def agregarCoordXY(self, root):
17        bounds = root.find("bounds")
18        self.ymin = float(bounds.get("minlat"))
19        self.xmin = float(bounds.get("minlon"))
20        self.ymax = float(bounds.get("maxlat"))
21        self.xmax = float(bounds.get("maxlon"))
22
23
24    class Vertice:
25        def __init__(self, idv):
26            self.id = idv
27            self.x = None
28            self.y = None
29            self.nombre = ""
```

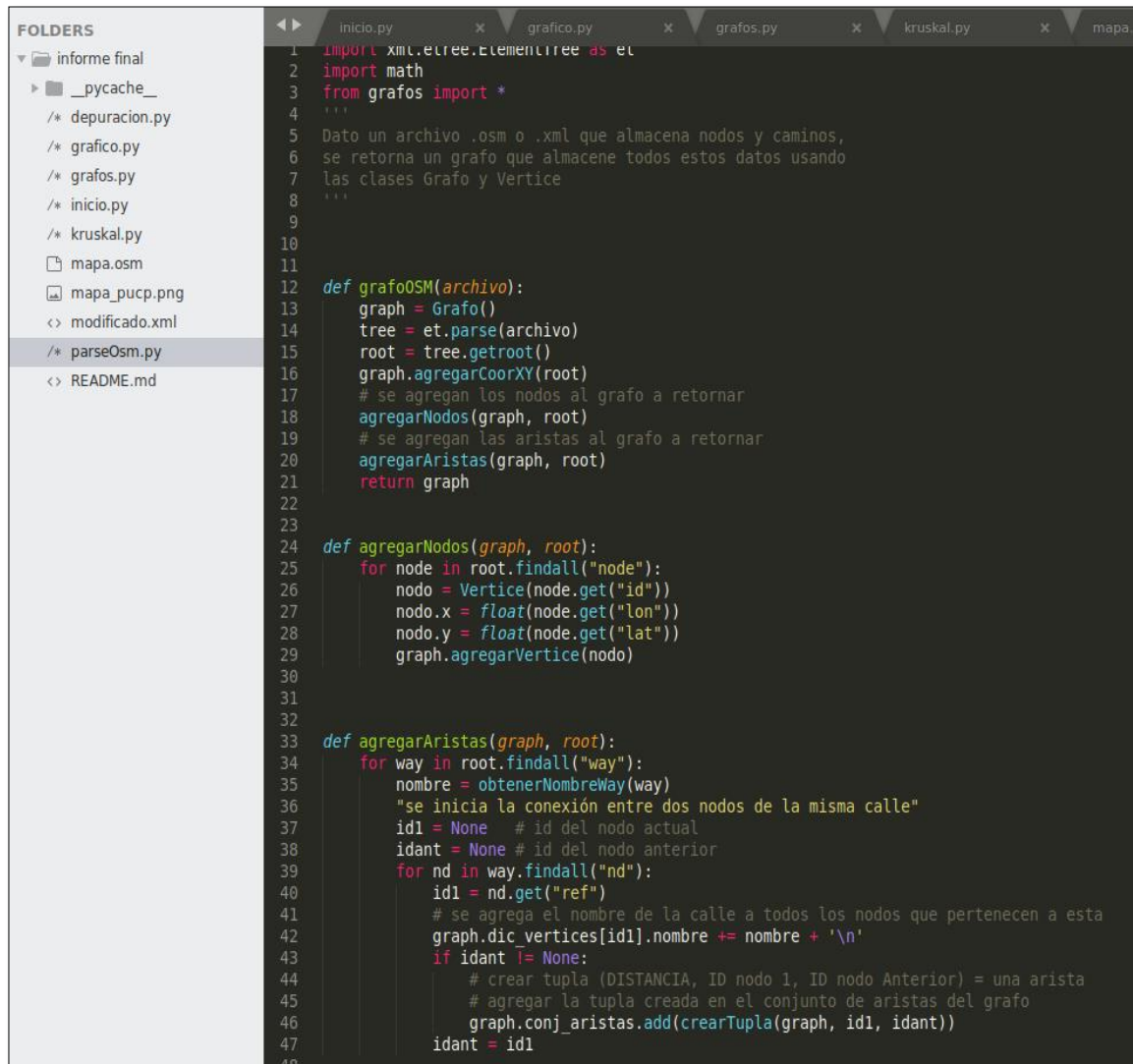
4. Almacenamiento de datos en el TAD grafo

Luego de realizar el proceso de depuración, se lee los datos de los nodos del archivo modificado.xml y estos se almacenan en el diccionario de vértices del grafo. De la misma manera, se recorre todos los caminos (way) y se almacenan en el conjunto de aristas del grafo en forma de tuplas (longitud, id nodo1, id nodo2). Observación: la longitud es calculada por el método distancia. A continuación, se muestra el código de los procedimientos para realizar lo mencionado anteriormente:



```
File Edit Selection Find View Goto Tools Project Preferences Help
FOLDERS
  informe final
  _pycache_
  /* depuracion.py
  /* grafico.py
  /* grafos.py
  /* inicio.py
  /* kruskal.py
  mapa.osm
  mapa_pucp.png
  <> modificado.xml
  /* parseOsm.py
  <> README.md

60
61
62
63 # observacion: nodo anterior es tratado como nodo2, esto no afecta al algoritmo de kruskal,
64 # pues esta es en base a un grafo no dirigido, modificar esto cuando se trabaje con los dirigidos
65 def crearTupla(graph, id1, idant):
66     nodo1 = graph.dic_vertices[id1]
67     nodo2 = graph.dic_vertices[idant]
68     tupla = (distancia(nodo1.y, nodo1.x, nodo2.y, nodo2.x), nodo1.id, nodo2.id)
69     return tupla
70
71 def distancia(lat1, lon1, lat2, lon2):
72     dlat = lat1 * math.pi / 180 - lat2 * math.pi / 180 # diferencia de latitudes
73     dlon = lon1 * math.pi / 180 - lon2 * math.pi / 180 # diferencia de longitudes
74     R = 6372.795477598 # Km del radio de la tierra
75     aux = math.sin(dlat / 2) ** 2 + math.cos(lat1 * math.pi / 180) * math.cos(lat2 * math.pi / 180) * (math.sin(dlon / 2) ** 2)
76     dist = 2 * R * math.asin(math.sqrt(aux))
77     return dist #retornamos distancia en kilometros
78
79 def obtenerNombreWay(way):
80     nombre = ""
81     for tag in way.findall("tag"):
82         if tag.get("k") == "name": nombre = tag.get("v")
83     return nombre
84
```



```
1 import xml.etree.ElementTree as et
2 import math
3 from grafos import *
4
5 Dato un archivo .osm o .xml que almacena nodos y caminos,
6 se retorna un grafo que almacene todos estos datos usando
7 las clases Grafo y Vertice
8
9
10
11
12 def grafoOSM(archivo):
13     graph = Grafo()
14     tree = et.parse(archivo)
15     root = tree.getroot()
16     graph.agregarCoordXY(root)
17     # se agregan los nodos al grafo a retornar
18     agregarNodos(graph, root)
19     # se agregan las aristas al grafo a retornar
20     agregarAristas(graph, root)
21     return graph
22
23
24 def agregarNodos(graph, root):
25     for node in root.findall("node"):
26         nodo = Vertice(node.get("id"))
27         nodo.x = float(node.get("lon"))
28         nodo.y = float(node.get("lat"))
29         graph.agregarVertice(nodo)
30
31
32
33 def agregarAristas(graph, root):
34     for way in root.findall("way"):
35         nombre = obtenerNombreWay(way)
36         "se inicia la conexión entre dos nodos de la misma calle"
37         id1 = None # id del nodo actual
38         idant = None # id del nodo anterior
39         for nd in way.findall("nd"):
40             id1 = nd.get("ref")
41             # se agrega el nombre de la calle a todos los nodos que pertenecen a esta
42             graph.dic_vertices[id1].nombre += nombre + '\n'
43             if idant != None:
44                 # crear tupla (DISTANCIA, ID nodo 1, ID nodo Anterior) = una arista
45                 # agregar la tupla creada en el conjunto de aristas del grafo
46                 graph.conj_aristas.add(crearTupla(graph, id1, idant))
47             idant = id1
48
```

5. Uso del algoritmo del algoritmo de Kruskal

Luego de haber almacenado la información en un grafo, este se utiliza para poder aplicar el algoritmo de Kruskal y obtener un nuevo conjunto de tuplas de aristas. Cabe mencionar que estas aristas son aquellas que tienen el coste mínimo y que en conjunto representan el árbol (subgrafo sin ciclos) de coste mínimo. El código implementado se muestra a continuación:


```
File Edit Selection Find View Goto Tools Project Preferences Help

FOLDERS
  informe final
  _pycache_
  /* depuracion.py
  /* grafico.py
  /* grafos.py
  /* inicio.py
  /* kruskal.py
  mapa.osm
  mapa_pucp.png
  modificado.xml
  /* parseOsm.py
  README.md

1 # Implementación del algoritmo Kruskal
2 from grafos import *
3 # Variables globales
4 base = dict()
5 ord = dict()
6
7
8 # Función para generar conjuntos
9 def make_set(v):
10     base[v] = v
11     ord[v] = 0
12
13
14 # Implementación de la función de búsqueda
15 # de manera recursiva
16 def find(v):
17     if base[v] != v:
18         base[v] = find(base[v])
19     return base[v]
20
21
22 # Implementación de la unión de conjuntos
23 def union(u, v):
24     v1 = find(u)
25     v2 = find(v)
26     if ord[v1] > ord[v2]:
27         base[v2] = v1
28     else:
29         base[v1] = v2
30         if ord[v1] == ord[v2]:
31             ord[v2] += 1
32
33 # Función principal del algoritmo Kruskal
34 def kruskal(graph):
35     # A = {conjunto vacío}
36     mst = set()
37
38     # Para todo vértice v en G.V
39     #for v in graph.vertices.keys():
40     for v in graph.listaVertices():
41         make_set(v)
42
43     # Ordena la lista G.E en forma no decedente por su peso w
44     # En este caso usamos el ordenador dentro de python
45     edges = list(graph.conj_aristas)
46     edges.sort()
47
48     # Para toda arista(u,v) en G.E
49     for e in edges:
50         weight, u, v = e
51         # Si encontrar-conjunto(u) != encontrar-conjunto(v)
52         if find(u) != find(v):
53             # A = A union (u,v)
54             union(u, v)
55             # Union(u,v)
56             mst.add(e)
57
58
```

```
File Edit Selection Find View Goto Tools Project Preferences Help

FOLDERS
  informe final
  _pycache_
  /* depuracion.py
  /* grafico.py
  /* grafos.py
  /* inicio.py
  /* kruskal.py
  mapa.osm
  mapa_pucp.png
  modificado.xml
  /* parseOsm.py
  README.md

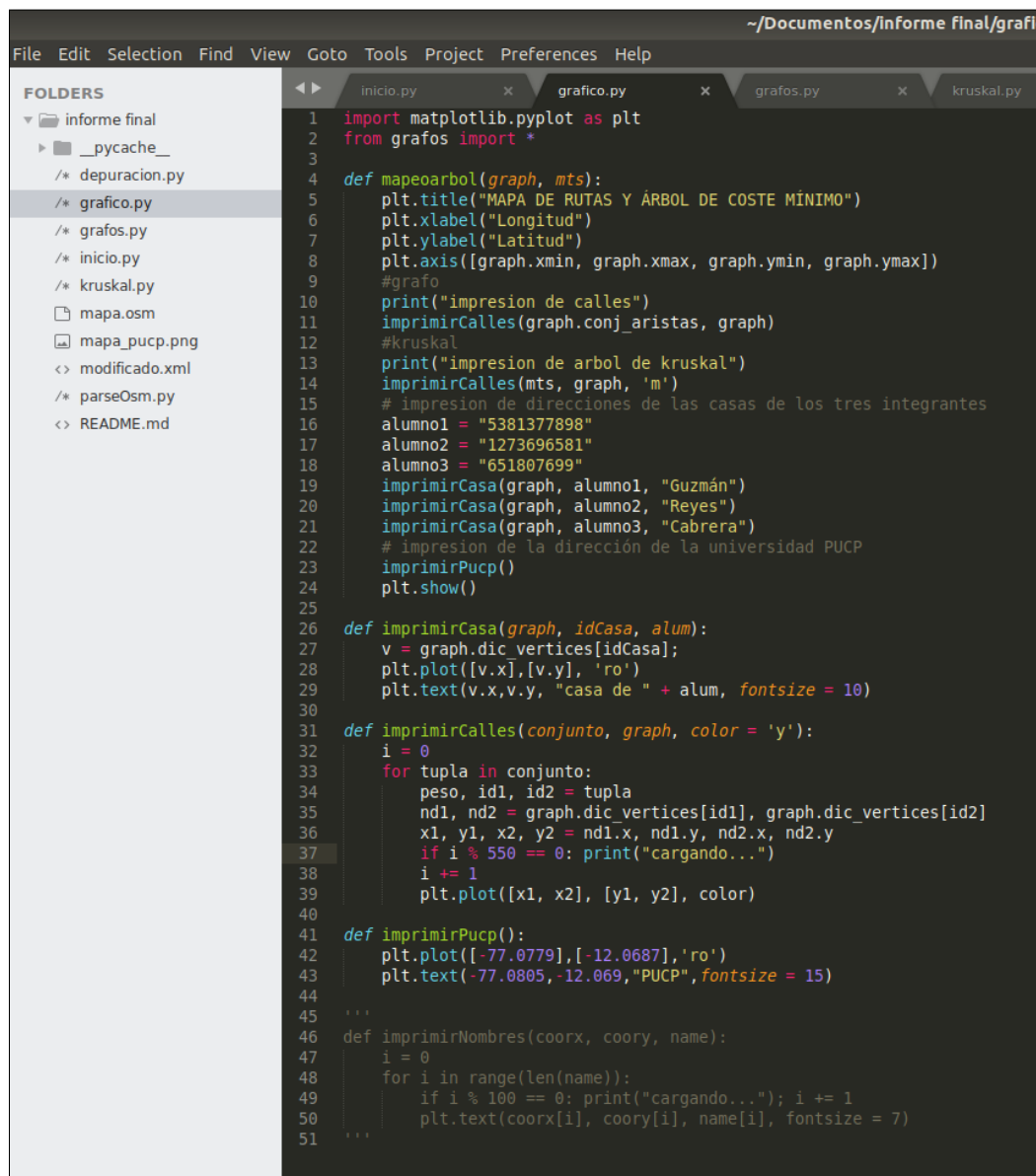
32
33 # Función principal del algoritmo Kruskal
34 def kruskal(graph):
35     # A = {conjunto vacío}
36     mst = set()
37
38     # Para todo vértice v en G.V
39     #for v in graph.vertices.keys():
40     for v in graph.listaVertices():
41         make_set(v)
42
43     # Ordena la lista G.E en forma no decedente por su peso w
44     # En este caso usamos el ordenador dentro de python
45     edges = list(graph.conj_aristas)
46     edges.sort()
47
48     # Para toda arista(u,v) en G.E
49     for e in edges:
50         weight, u, v = e
51         # Si encontrar-conjunto(u) != encontrar-conjunto(v)
52         if find(u) != find(v):
53             # A = A union (u,v)
54             union(u, v)
55             # Union(u,v)
56             mst.add(e)
57     return mst
58
```

APLICACIÓN

1. Uso de Matplotlib para representación de árbol de coste mínimo

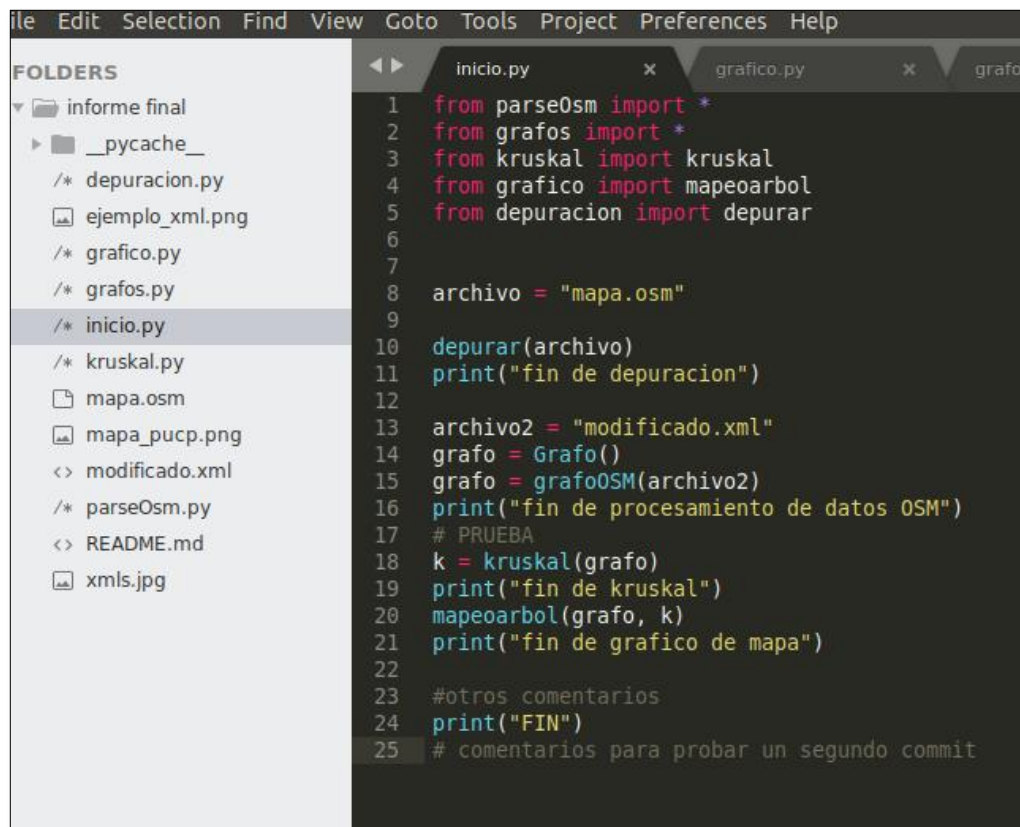
Luego de obtener el árbol de coste mínimo se decidió representarlo de manera gráfica mediante el uso de la librería de Python matplotlib.

Se crea la función `mapeoarb()` el cual maneja un plano cartesiano y lo almacena en `plt`. En este plano, se imprimen los nodos de acuerdo a sus coordenadas, las cuales ya han sido leídas en el parseo y almacenadas en el grafo. Además, imprime líneas entre dos coordenadas de nodos que representan una calle. Por último, se imprimen las posiciones de las casas de los integrantes del grupo con ayuda de los id de los nodos que son cercanos a estas, y un nodo en la entrada de la universidad. En código empleado se muestra a continuación:



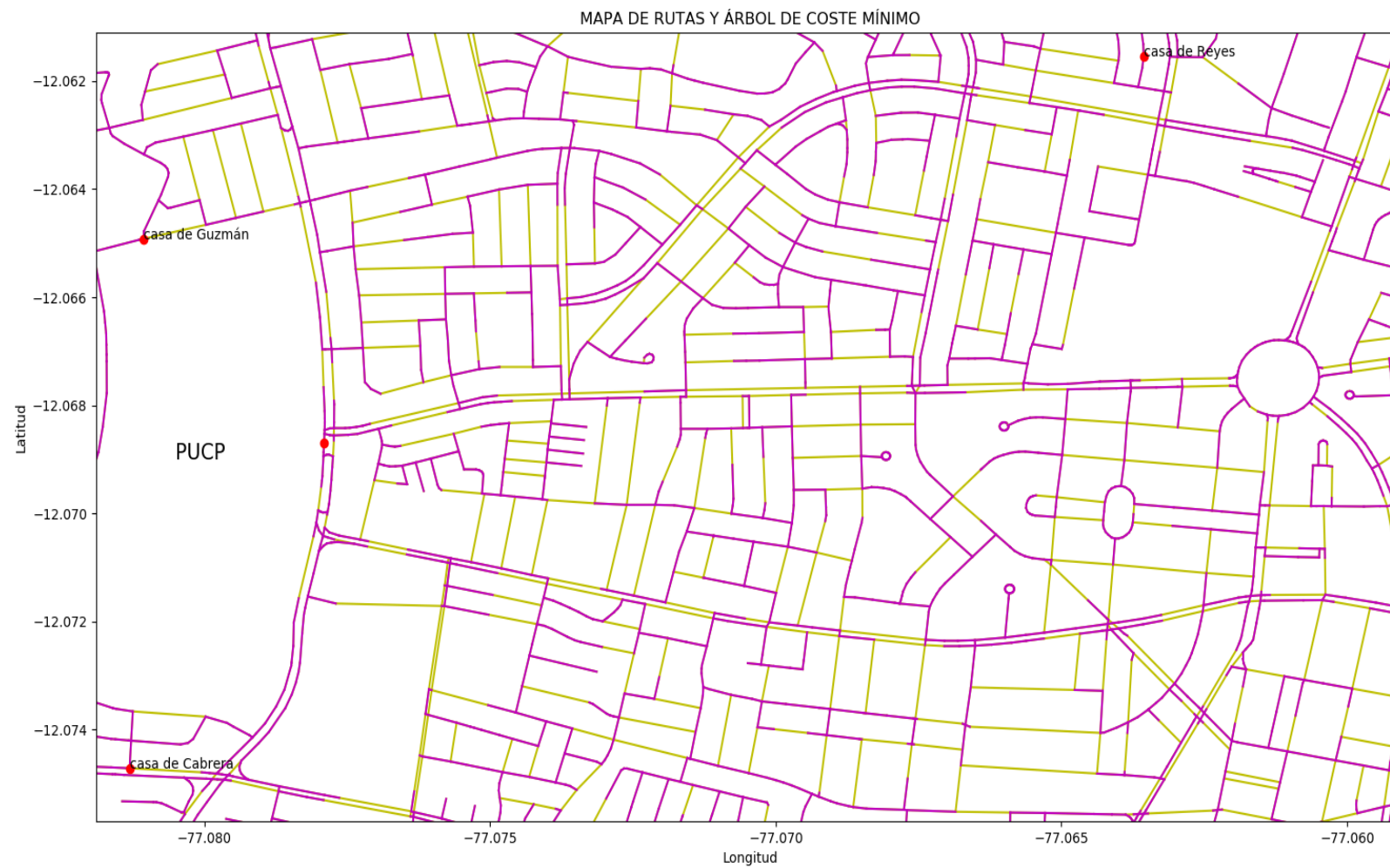
```
~/Documentos/informe final/grafi
File Edit Selection Find View Goto Tools Project Preferences Help
FOLDERS
  informe final
    __pycache__
    /* depuracion.py
    /* grafico.py
    /* grafos.py
    /* inicio.py
    /* kruskal.py
    mapa.osm
    mapa_pucp.png
    <> modificado.xml
    /* parseOsm.py
    <> README.md
grafico.py
1 import matplotlib.pyplot as plt
2 from grafos import *
3
4 def mapeoarb(graph, mts):
5     plt.title("MAPA DE RUTAS Y ÁRBOL DE COSTE MÍNIMO")
6     plt.xlabel("Longitud")
7     plt.ylabel("Latitud")
8     plt.axis([graph.xmin, graph.xmax, graph.ymin, graph.ymax])
9     #grafo
10    print("impresion de calles")
11    imprimirCalles(graph.conj_aristas, graph)
12    #kruskal
13    print("impresion de arbol de kruskal")
14    imprimirCalles(mts, graph, 'm')
15    # impresion de direcciones de las casas de los tres integrantes
16    alumno1 = "5381377898"
17    alumno2 = "1273696581"
18    alumno3 = "651807699"
19    imprimirCasa(graph, alumno1, "Guzmán")
20    imprimirCasa(graph, alumno2, "Reyes")
21    imprimirCasa(graph, alumno3, "Cabrera")
22    # impresion de la dirección de la universidad PUCP
23    imprimirPucp()
24    plt.show()
25
26 def imprimirCasa(graph, idCasa, alum):
27     v = graph.dic_vertices[idCasa];
28     plt.plot([v.x], [v.y], 'ro')
29     plt.text(v.x, v.y, "casa de " + alum, fontsize = 10)
30
31 def imprimirCalles(conjunto, graph, color = 'y'):
32     i = 0
33     for tupla in conjunto:
34         peso, id1, id2 = tupla
35         nd1, nd2 = graph.dic_vertices[id1], graph.dic_vertices[id2]
36         x1, y1, x2, y2 = nd1.x, nd1.y, nd2.x, nd2.y
37         if i % 550 == 0: print("cargando...")
38         i += 1
39         plt.plot([x1, x2], [y1, y2], color)
40
41 def imprimirPucp():
42     plt.plot([-77.0779], [-12.0687], 'ro')
43     plt.text(-77.0805, -12.069, "PUCP", fontsize = 15)
44
45 ...
46 def imprimirNombres(coorx, coory, name):
47     i = 0
48     for i in range(len(name)):
49         if i % 100 == 0: print("cargando..."); i += 1
50         plt.text(coorx[i], coory[i], name[i], fontsize = 7)
51 ...
```


Para finalizar, en el siguiente código se agrupa todas las implementaciones anteriores que permiten la ejecución del programa en general.



```
file Edit Selection Find View Goto Tools Project Preferences Help
FOLDERS
  informe final
    __pycache__
    /* depuracion.py
    ejemplo_xml.png
    /* grafico.py
    /* grafos.py
    /* inicio.py
    /* kruskal.py
    mapa.osm
    mapa_pucp.png
    <> modificado.xml
    /* parseOsm.py
    <> README.md
    xmls.jpg
inicio.py
1  from parseOsm import *
2  from grafos import *
3  from kruskal import kruskal
4  from grafico import mapeoarboll
5  from depuracion import depurar
6
7
8  archivo = "mapa.osm"
9
10 depurar(archivo)
11 print("fin de depuracion")
12
13 archivo2 = "modificado.xml"
14 grafo = Grafo()
15 grafo = grafoOSM(archivo2)
16 print("fin de procesamiento de datos OSM")
17 # PRUEBA
18 k = kruskal(grafo)
19 print("fin de kruskal")
20 mapeoarboll(grafo, k)
21 print("fin de grafico de mapa")
22
23 #otros comentarios
24 print("FIN")
25 # comentarios para probar un segundo commit
```

A continuación se muestra el resultado final, la cual es la representación gráfica del árbol de coste mínimo. En el que se ubican los domicilios de los integrantes del equipo y la universidad.



CONCLUSIONES

- Es uso de grafos para la representación de situaciones de la vida real como el modelamiento de una red de calles es de gran utilidad puesto que permite el manejo de información de manera más sencilla en beneficio del programador.
- Se concluyó que el algoritmo de Kruskal no es adecuado para para resolver el problema de encontrar la ruta más corta entre dos nodos. Sin embargo, este permite tomar decisiones de gran importancia en base al árbol de coste mínimo.
- Con la ayuda de matplotlib se pueden lograr diferentes tipos de gráficos los cuales nos pueden servir para representar diferentes tipos de diagramas y eso nos permitió explotar las librerías de Python.

BIBLIOGRAFÍA:

eeNube Programación.

Algoritmo de Kruskal. Consulta: 2 de mayo del 2018.

<http://eenube.com/index.php/ldp/algoritmos/38-algoritmo-kruskal-pseudocodigo-diagrama-de-flujos-e-implementacion-python>

Grafos

Algoritmo de kruskal. Consulta: 2 de mayo del 2018.

http://arodrigu.webs.upv.es/grafos/doku.php?id=algoritmo_kruskal

Matplotlib

Pyplot tutorial. Consulta: 3 de mayo de 2018.

<https://matplotlib.org/tutorials/introductory/pyplot.html#sphx-glr-tutorials-introductory-pyplot-py>

Matplotlib

Installing. Consulta: 3 de mayo de 2018.

<https://matplotlib.org/users/installing.html#linux>

Python

The elementTree XML API. Consulta 3 de mayo del 2018.

<https://docs.python.org/3.6/library/xml.etree.elementtree.html>

OPEN STREET MAP

<https://www.openstreetmap.org/#map=16/-12.0694/-77.0686>

IBM

¿Qué es XML? Consulta: 2 de junio de 2018

https://www.ibm.com/support/knowledgecenter/es/SSEPGG_8.2.0/com.ibm.db2.ii.doc/opt/c0007799.htm

WIKI

Key:highway. Consulta: 4 de junio de 2018

<https://wiki.openstreetmap.org/wiki/Key:highway>