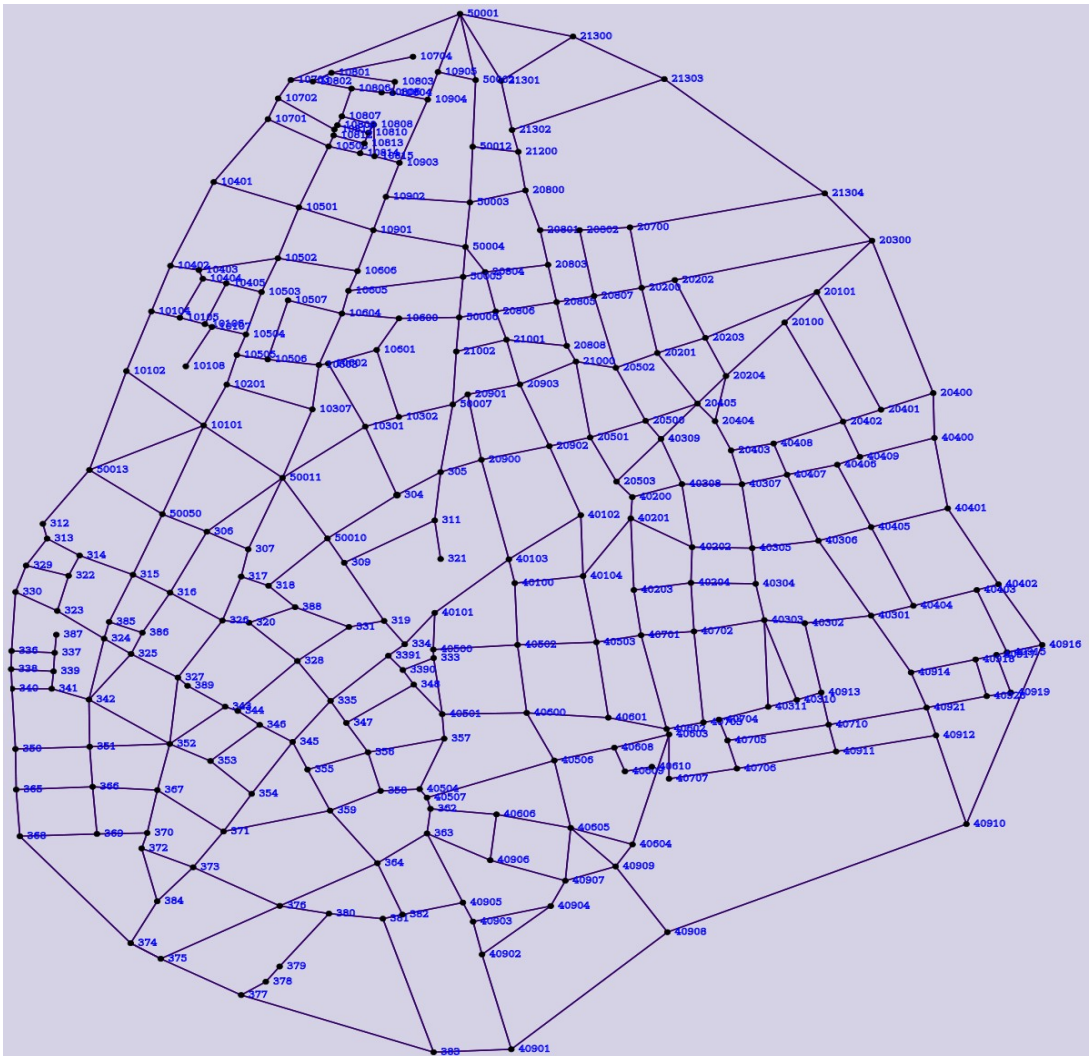
Práctica 1.
Inteligencia artificial.

Jesús Manuel Muñoz Mazuecos G2 - 06270974R

Índice.

1. Introducción al problema.

El objetivo del problema es determinar la ruta óptima entre dos puntos de la localidad de ciudad real. La ruta se determina para minimizar el espacio recorrido o para maximizar la aparición de comercios y negocios de restauración. Para representar el espacio de soluciones del problema se ha implementado un grafo no dirigido en el que cada nodo representa un cruce de calles. Cada nodo tiene un identificador único y unas coordenadas de localización asociadas, más adelante se detallará el por qué de estas coordenadas. Cada arco que conecta un par de nodos tiene asociado un peso. Este peso viene determinado por la distancia en metros del par de nodos que conecta, el número de comercios y el número de bares. Para determinar el camino a seguir entre dos nodos nos hemos apoyado en métodos de búsqueda que devuelven una solución al problema. Una solución viene determinada por una secuencia de nodos y una valoración de esta solución.



1.1 Soporte para la resolución del problema.
Una vez que está planteado el problema debemos elegir los recursos que vamos a utilizar para la resolución del mismo. La primera cuestión que surge viene relacionada con el almacenamiento del grafo y persistencia de información. Para almacenar información sobre el grafo de manera permanente utilizamos dos ficheros con extensión *.csv.* Un archivo se utiliza para almacenar las aristas del grafo mientras que en el otro se almacenan los nodos con sus correspondientes coordenadas de localización. Posteriormente surge la necesidad del uso de métodos que devuelvan una solución del problema, para ello se han pensado en métodos de búsqueda no informada, un método de búsqueda informada y otro método de búsqueda local. Los métodos que se van a utilizar son los siguientes.

- Búsqueda en anchura.
- Búsqueda en profundidad.
- Búsqueda mediante profundidad acotada.
- Búsqueda por costo uniforme.
- Búsqueda mediante A*.
- Búsqueda local por enfriamiento simulado.

Una vez que hemos decidido los métodos que vamos a utilizar debemos seleccionar un lenguaje de programación para la implementación de una solución en un computador. El lenguaje de implementación que se va a usar es *python.* Se trata de un lenguaje interpretado que soporta la programación orientada a objetos y además proporciona librerías para la gestión de grafos, ficheros csv y métodos eficientes para la inserción en listas ordenadas.
La solución también implica el diseño de una interfaz gráfica de usuario, para realizar este cometido nos hemos apoyado el la librería *wx* de python. Esta librería proporciona los métodos y clases necesarias para crear controles de usuario que permitan interactuar con el programa.

2. Solución del problema.

En esta sección se van a detallar las estrategias de diseño aplicadas para llegar a la solución. Para aspectos relacionados con el diseño se va a introducir cuando sea necesario pequeños fragmentos de código para clarificar determinados aspectos. Los pasos a seguir para llegar a la solución son los siguientes.

- Implementación del grafo.
- Definición de estado.
- Función de selección.
- Función de sucesión.
- Función de inserción.
- Algoritmos de búsqueda.

2.1 Implementación del grafo.

Para implementar el grafo nos hemos apoyado en la clase *pygraph* de python. La clase *pygraph* proporciona métodos para generar vértices y aristas con atributos. Python proporciona un constructor para crear un grafo y métodos para insertar nodos y aristas. En nuestra solución utilizamos un método que a medida que va leyendo filas del fichero *csv* se va generando el grafo.

2.2 Definición de estado.

En esta sección se va a definir el concepto de estado dentro del ámbito del problema. Mientras que uno de los métodos de búsqueda está generando una solución se tiene que evaluar los estados que son accesibles en un determinado momento. De este modo un estado está definido por una serie de parámetros que identifican el estado de la solución. Cada método de búsqueda se basa en la función estado para construir la solución. A medida que uno de los métodos de búsqueda va construyendo una ruta se evalúan los estados que están a nuestro alcance, y en este caso la función estado nos devuelve el impacto inmediato si se produce un movimiento a un determinado nodo. Cada método evaluará los parámetros de la función estado que considere necesarios para construir la mejor solución posible según el criterio que aplique dicho método. La función estado está compuesta por los siguientes parámetros.

- Identificador del nodo.
- Profundidad acumulada.
- Coste asociado acumulado.
- Distancia euclídea al nodo destino.

2.3 Función de selección.

Una vez que tenemos el concepto de estado definido necesitamos mecanismos que determinen a qué nodo movernos una vez que hemos determinado los posibles estados en un determinado instante. Para ello vamos a utilizar una lista de inserción ordenada en donde vamos a ir insertando los estados que va generando el programa. Para almacenar un estado nos hemos definido una clase nodo tiene como atributos cada uno de los parámetros de la función estado y un valor que será el criterio de ordenación. Por lo tanto si vamos insertando elementos de tipo nodo en nuestra lista debemos establecer un criterio de ordenación de la misma. Dependiendo del método de búsqueda que estemos utilizando se establecerá un criterio u otro para la ordenación de la lista. De esta manera el primer elemento de la lista será el mejor estado accesible en un determinado momento. A continuación se detalla una tabla en donde se especifica el criterio de ordenación que usa cada método de búsqueda para la selección de candidatos.

Algoritmo	Modo	Criterio de ordenación
Búsqueda en anchura	0	Profundidad
Búsqueda en profundidad	1	-1 * Profundidad.
B. en profundidad acotada	2	-1 * Profundidad.
B. en profundidad iterativa	3	-1 * Profundidad.
B. mediante costo uniforme	4	Función costo.
Algoritmo A*	5	Función costo + distancia eucl.

2.3.1 Inserción en la lista.

Siempre que realicemos una búsqueda hemos de tener en cuenta que el número de elementos que se van a generar va a ser muy grande y una inserción ineficiente en la lista puede dar lugar a demoras de tiempo innecesarias. Para evitar esperas innecesarias en el proceso de generación nos hemos apoyado en la clase *bisect* de python que nos da el soporte necesario para el mantenimiento de una lista ordenada sin necesidad de reordenar la lista después de cada inserción, este hecho nos permitirá reducir el tiempo de ejecución. Debemos tener en cuenta que la lista irá creciendo con el tiempo y cada vez será más costoso ordenarla. En este caso nos encontramos con nodos que a la vez tienen sucesores, estos nodos contienen un tipo de dato real que será utilizado como nuestro criterio de ordenación. Cada elemento se va ir introduciendo en una posición de la lista que dependerá del valor del tipo de dato real, siempre manteniendo las referencias a sus predecesores aunque estos estén en diferentes posiciones de la lista. Por ejemplo si el nodo con el valor 2 genera tres sucesores de valores 1, 3 y 4 respectivamente. La lista quedará: 1-2-3-4 entonces el siguiente para generar sucesores será el nodo con el valor 1. Como ya se ha mencionado antes aunque la lista queda ordenada se mantienen las referencias a sus predecesores, así que en el caso de que nos queramos mover por la lista podemos saber los predecesores de cada nodo.

Otro punto a tener en cuenta es cómo se mantiene ordenada la lista. El concepto es que partimos de una lista inicialmente ordenada de un sólo elemento. A partir de aquí nos apoyamos en el módulo bisect de python, el cual localiza la siguiente posición de inserción de un elemento en una lista ordenada. Una vez que sabemos la posición de inserción lo insertamos en la lista:

lista[posición] = nuevo

De manera que el coste para insertar un dato es el coste de búsqueda de la posición mas el coste de inserción en una posición dada. Cabe destacar que a la hora de insertar en la posición no se machaca el elemento si no que se desplaza la lista (se actualizan las referencias del elemento anterior y siguiente sin tener que recorrer la lista). Otro tema es que bisect se basa en una lista ordenada de claves para determinar la posición de inserción (ésta lista contiene referencias al valor del dato real de cada nodo). A continuación se muestra una estadística de generación en inserción en la que se ve claramente el efecto de una inserción en la lista eficiente.

- Elementos insertados: 500000
- Tiempo medio de inserción: $1.9868\,e^{-6}$ *segundos*

A medida que va creciendo la lista el coste de generación aumenta ya que las referencias a los sucesores se deben mantener en memoria. Sin embargo el tiempo medio de inserción no se dispara gracias a la función del módulo "Array bisection algorithm" de python. El siguiente ejemplo ilustra el método a seguir si se quiere utilizar el módulo bisect.

- pos = bisect_left(keys, nodo.valor)
- lista.insert(pos, nodo)
- insort(keys, nodo.valor)

2.4 Función de sucesión.

En nuestro problema cada vez que se selecciona un nodo debemos generar sus correspondientes sucesores. A medida que vamos avanzando en nuestro camino para llegar a la solución vamos visitando una serie de nodos. Por lo tanto se ha adoptado la elección de eliminar los nodos visitados de cualquier lista de sucesores que se genere a lo largo del camino. A continuación se muestra el fragmento de código correspondiente a la función que genera sucesores. Cabe destacar que el temple simulado no tiene en cuenta los nodos visitados en el momento que genera un sucesor aleatorio.

```
vecinos = grafo.neighbors(n)

# eliminar los visitados

for vertex in vecinos:
    if visitados.count(vertex) != 0:
       vecinos.remove(vertex)

return vecinos
```

2.5 Función de inserción.

A continuación se detalla la función de inserción. Este método recibe el grafo de la ciudad, el padre del nodo que vamos a crear, el nombre del vértice que vamos a insertar y el modo en el que estamos buscando para indicar al constructor de la clase nodo que valor se debe asignar a la función de valoración. De esta manera cada vez que nos encontramos con un identificador de nodo correspondiente a un sucesor podemos determinar el estado si nos movemos al nodo hijo.

```
def insertar(g, padre, vertice, modo):

    #calulo del costo

    costoArista = calcularCosto(g, padre.state.IDVertex, vertice)
    costo = padre.state.costo + costoArista

    #calculamos la distancia euclidea...

    distancia = euclidea(origen, vertice)

    # se crea un tipo nodo para insertar en la lista

    nod=nodo(modo, padre,vertice,padre.state.profundidad + 1,costo,distancia)

    # INSERCIÓN EN LA LISTA
    tIniIns = time.time()
    pos = bisect_left(keys, nod.valor)
    lista.insert(pos, nod)
    insort(keys, nd.valor)
    SumTime = + (time.time() - tIniIns)
    # FIN INSERCIÓN
```

2.6 Algoritmos de búsqueda.

En esta sección se va a detallar la estructura de los algoritmos de búsqueda que se han utilizado para encontrar la solución a un problema. Como ya se ha mencionado antes, cada método se basa en un parámetro de la función estado para establecer el criterio de ordenación en la lista de nodos accesibles.

Algoritmo	Modo	Criterio de ordenación
Búsqueda en anchura	0	Profundidad
Búsqueda en profundidad	1	-1 * Profundidad.
B. en profundidad acotada	2	-1 * Profundidad.
B. en profundidad iterativa	3	-1 * Profundidad.
B. mediante costo uniforme	4	Función costo.
Algoritmo A*	5	Función costo + distancia eucl.

2.6.1 Estructura del algoritmo de búsqueda.

La estructura del algoritmo de búsqueda es independiente del método de búsqueda que se utilice. Lo único que difiere es el criterio con el que se selecciona un nodo elegido. Cada vez que se generan sucesores se crea un tipo nodo y se inserta en la lista, el algoritmo ordena la lista según el criterio que se especifique. A continuación se detalla la estructura del algoritmo de búsqueda.

- Generar raíz.
- Insertar raíz en la lista.
- Mientras la lista no sea vacía:
  - Extraer el primero.
  - Eliminar el primero
  - Si el elegido es igual al destino
    - Devolver solución
  - Sino
    - Generar sucesores.
    - Insertar sucesores en la lista.

2.6.2 Generación de nodos.

Cada vez que se genera un nodo tenemos que crear un objeto de la clase nodo que contiene información sobre el estado. El constructor de la clase nodo posee un campo *valor* que se trata de un atributo de tipo real que se utiliza para ordenar la lista. Para el algoritmo A* la heurística utilizada es la suma de la función coste más la distancia euclídea.

- Atributos del tipo nodo.
  - Criterio: es un tipo entero que identifica el método de búsqueda que estamos utilizando.
  - Father: es una referencia al padre del nodo que hemos generado.
  - IDVertex: Identificador del vértice.
  - Profundidad: tipo entero que se corresponde con la profundidad del nodo.
  - Costo: Coste acumulado hasta el momento.
  - Euclídea: Distancia euclídea del nodo creado hasta el destino.

A continuación se va a detallar el código correspondiente al constructor de la clase nodo en donde se puede apreciar como se establece el criterio de ordenación.

```
def __init__(self, criterio = 0, father = None, IDVertex = '', profundidad = 0,
        costo = 0.0, euclidea = 0.0):

        if criterio == 0:                                       # anchura
            self.valor = float(profundidad)
        elif criterio == 1 or criterio == 2 or criterio == 3: # profundidades
            self.valor = -1.0 * float(profundidad)
        elif criterio == 4:                                     # costo uniforme
            self.valor = costo
        elif criterio == 5:                                     # A*
            self.valor = costo + euclidea
        else:
            print "Criterio no válido"
```

2.6.3 Cálculo del coste.

La solución implementada dispone de un método para asignar el coste a un nodo. El método toma como parámetros el grafo en donde se está realizando la búsqueda, el padre de nodo elegido y el nodo al que le queremos asignar el coste. Cabe destacar que tenemos tres valores que conforman el coste:

- Distancia en metros entre el par de nodos.
- Número de bares del tramo.
- Número de tiendas del tramo.
- Distancia euclídea del nodo elegido al destino (Sólo para A*).

Debemos tener un especial cuidado en la ponderación de estos valores para obtener un solución óptima en los casos de costo uniforme y búsqueda informada por A*. Una ponderación inadecuada puede dar lugar a una sobre-estimación de posibles estados. Por ejemplo un nodo que tiene una distancia euclídea con respecto al destino menor que otro no quiere decir que sea menos costoso elegir este camino. Por eso se ha tenido en cuenta una fórmula para estimar el costo acumulado de un nodo. Si estamos realizando una búsqueda informada por el método de A* al coste le damos una ponderación de 0.8 mientras que la distancia euclídea la ponderamos con un factor de 0.2. Por otra parte si queremos minimizar la distancia recorrida, aplicamos la siguiente fórmula:

$$Coste = 0.8 * (distancia + 1000)$$

$$Distancia\ \ euclídea = 0.2 * euclídea$$

De esta manera variaciones sustanciales en la distancia tendrán más importancia que variaciones de la propia distancia euclídea respecto al destino.

2.6.4 Búsqueda local mediante enfriamiento simulado.

La búsqueda local mediante temple simulado toma como argumentos un grafo, un nodo origen y otro destino para devolver un camino óptimo basándose en el algoritmo A* para generar las soluciones vecinas. El algoritmo toma como solución inicial un camino y una valoración inicial (bastante mala) del camino entre el origen y el destino. A partir de este punto el algoritmo selecciona un nodo vecino al azar para devolver una solución vecina. Si la valoración de la solución vecina es mejor que la valoración de la solución actual se selecciona este nodo como actual y se actualiza la solución. En caso contrario si la valoración del nodo candidato es peor que la valoración de la solución actual. Se selecciona dependiendo de un factor de probabilidad que depende de la temperatura actual del algoritmo. A temperaturas altas suele ser bastante probable que se seleccione un nodo con peor valoración. En cada iteración la temperatura se decrementa por un factor *alpha*, el algoritmo termina una vez que se ha alcanzado un valor de temperatura muy bajo. Por otra parte, el número de iteraciones del bucle interno se incrementa a medida que sube la temperatura, a temperaturas bajas tenemos un número de iteraciones bajo mientras que a temperaturas altas el número de iteraciones es bastante alto. Cabe destacar que para almacenar las soluciones intermedias tenemos un objeto de la clase solución que almacena un camino y una valoración de este camino. Una vez que aceptamos una solución candidata, cogemos nuestro camino desde el nodo inicial hasta el nodo actual y concatenamos a este subcamino la solución vecina.
En el anexo de este documento se adjunta el código fuente correspondiente a este algoritmo.

3. Pruebas.

A lo largo de esta sección se van a ver los diferentes resultados que obtenemos aplicando cada uno de los algoritmos desarrollados en la práctica 1 para una serie de problemas complejidad sencilla, media y avanzada. Complejidad del problema depende del anillo en que están contenidos los nodos origen y destino.

3.1 Especificación de los problemas.

Para el problema de complejidad sencilla se han escogido dos nodos que se encuentran dentro del primer anillo, el problema de complejidad media halla el camino entre dos nodos del segundo anillo y el problema de complejidad alta escoge dos nodos contenidos en el tercer anillo. Básicamente la complejidad depende de la distancia euclídea entre los nodos origen y destino.

Complejidad.	Nodo origen.	Nodo destino.
Sencilla.	40902	10807
Media.	359	10604
Avanzada.	10504	10401

3.2 Soluciones por la búsqueda en anchura.

A continuación se va a aplicar el recorrido en anchura para cada uno de los problemas propuestos.

3.2.1 Problema de complejidad sencilla.

Recorrido en anchura para el problema de complejidad sencilla.

- Origen: 10504.
- Destino: 10401.
- Coste: 3469.6.
- Camino: ['10504', '10503', '10502', '10501', '10401'].
- Elementos generados: 49.
- Tiempo medio de inserción: 1.6543 e^{-7} segundos.
- Tiempo de ejecución: 0.042 segundos.

3.2.2 Problema de complejidad media.

Recorrido en anchura para el problema de complejidad media.

- Origen: 359.
- Destino: 10604.
- Coste: 9712.0.
- Camino: ['359', '371', '367', '352', '327', '326', '316', '306', '50011', '10307', '10603', '10604']
- Elementos generados: 391.
- Tiempo medio de inserción: 2.561 e^{-8} segundos.
- Tiempo de ejecución: 0.436 segundos.

3.2.3 Problema de complejidad avanzada.

Recorrido en anchura para el problema de complejidad avanzada.

- Origen: 40902.
- Destino: 10807.
- Coste: 16300.80.
- Camino: ['40902', '40901', '40908', '40910', '40916', '40402', '40401', '40400', '20400', '20300', '21304', '21303', '21300', '50001', '10703', '10802', '10806', '10807'].
- Elementos generados: 572.
- Tiempo medio de inserción: 1.208 e^{-8} segundos
- Tiempo de ejecución: 0.719 segundos.

3.3. Soluciones para la búsqueda en profundidad.

A continuación se va a aplicar el recorrido en profundidad para cada uno de los problemas propuestos.

3.3.1 Problema de complejidad sencilla.

Recorrido para el problema de complejidad sencilla.

- Origen: 10504.
- Destino: 10401.
- Coste: 108950.4.
- Camino: ['10504', '10505', '10506', '10507', '10604', '10605', '50005', '20804', '20806', '21001', '20808', '21000', '20502', '20807', '20200', '20700', '20200', '20202', '20300', '20101', '20401', '20402', '40408', '40407', '40406', '40409', '40400', '40401', '40402', '40916', '40910', '40912', '40921', '40710', '40913', '40302', '40303', '40311', '40704', '40703', '40602', '40701', '40602', '40603', '40608', '40506', '40605', '40909', '40604', '40605', '40907', '40906', '40606', '362', '363', '40905', '40903', '40902', '40901', '383', '377', '375', '376', '373', '372', '373', '384', '374', '368', '369', '370', '367', '352', '342', '325', '327', '389', '343', '344', '346', '345', '354', '371', '359', '358', '40504', '357', '40501', '333', '40500', '40502', '40503', '40104', '40201', '40203', '40204', '40702', '40303', '40304', '40305', '40307', '40308', '40309', '20503', '20501', '20902', '20900', '20901', '50007', '10302', '10301', '304', '50010', '309', '319', '331', '388', '318', '317', '307', '50011', '10101', '10102', '10104', '10402', '10401']
- Elementos generados: 485.
- Tiempo medio de inserción: 1.868 e^{-7} segundos.
- Tiempo de ejecución: 0.464 segundos.

3.3.2 Problema de complejidad media.

Recorrido para el problema de complejidad media.

- Origen: 359.
- Destino: 10604.
- Coste: 72520.8
- Camino: ['359', '364', '382', '40905', '40903', '40904', '40907', '40909', '40604', '40603', '40707', '40706', '40911', '40710', '40921', '40710', '40913', '40302', '40303', '40304', '40305', '40306', '40407', '40408', '20403', '20404', '20405', '40309', '40308', '40307', '40407', '40406', '40409', '40400', '20400', '20401', '20402', '20100', '20101', '20300', '20202', '20200', '20700', '20802', '20801', '20803', '20805', '20808', '21000', '20502', '20807', '20805', '20808', '21000', '20501', '20902', '40102', '40104', '40201', '40203', '40701', '40602', '40601', '40600', '40506', '40507', '40504', '357', '356', '347', '348', '3390', '333', '40500', '40101', '334', '319', '309', '311', '305', '304', '10301', '10602', '10603', '10604']
- Elementos generados: 223.
- Tiempo medio de inserción: 4.062 e^{-8} segundos.
- Tiempo de ejecución: 0.392 segundos.

3.3.3 Problema de complejidad avanzada.

Recorrido para el problema de complejidad avanzada.

- Origen: 40902.
- Destino: 10807.
- Coste: 107835.2
- Camino: ['40902', '40904', '40907', '40909', '40604', '40603', '40707', '40706', '40911', '40710', '40921', '40710', '40913', '40302', '40303', '40304', '40305', '40306', '40407', '40408', '20403', '20404', '20405', '40309', '40308', '40307', '40407', '40406', '40409', '40400', '20400', '20401', '20402', '20100', '20101', '20300', '20202', '20200', '20700', '20802', '20801', '20803', '20805', '20808', '21000', '20502', '20807', '20805', '20808', '21000', '20501', '20902', '40102', '40104', '40201', '40203', '40701', '40602', '40601', '40600', '40506', '40507', '362', '363', '40905', '363', '364', '376', '373', '372', '373', '384', '374', '368', '369', '370', '367', '352', '342', '325', '327', '389', '343', '344', '346', '345', '335', '3391', '334', '319', '309', '311', '305', '304', '10301', '10602', '10603', '10604', '10605', '50005', '50006', '20806', '20804', '50004', '50003', '50012', '21200', '21302', '21303', '21300', '50001', '10905', '10904', '10903', '10815', '10808', '10810', '10813', '10814', '10508', '10701', '10702', '10703', '10802', '10806', '10807']
- Elementos generados: 367.
- Tiempo medio de inserción: 6.756 e^{-8}
- Tiempo de ejecución: 0.419 segundos.

3.4 Soluciones para la búsqueda en profundidad acotada.

A continuación se va a aplicar el recorrido en profundidad acotada para cada uno de los problemas propuestos. Se han aplicado límites de profundidad de 60.

3.4.1 Problema de complejidad sencilla.

Recorrido para el problema de complejidad sencilla.

- Origen: 10504.
- Destino: 10401.
- Coste: 39858.40.
- Camino: ['10504', '10505', '10506', '10507', '10604', '10605', '50005', '20804', '20806', '21001', '20808', '21000', '20502', '20807', '20200', '20700', '20200', '20202', '20300', '20101', '20401', '20402', '40408', '20403', '20404', '20405', '20500', '40309', '20503', '20501', '20902', '20900', '20901', '50007', '10302', '10301', '304', '50010', '318', '317', '307', '50011', '10101', '10102', '10104', '10402', '10401']
- Elementos generados: 363.
- Tiempo medio de inserción: 4.137 e^{-6} segundos.
- Tiempo de ejecución: 0.394 segundos.

3.4.2 Problema de complejidad media.

Recorrido para el problema de complejidad media.

- Origen: 359.
- Destino: 10604.
- Coste: 42376.80.
- Camino: ['359', '364', '382', '40905', '40903', '40904', '40907', '40909', '40604', '40603', '40707', '40706', '40911', '40710', '40921', '40710', '40913', '40302', '40303', '40304', '40305', '40306', '40407', '40408', '20403', '20404', '20405', '40309', '40308', '40307', '40407', '40406', '40409', '40400', '20400', '20401', '20402', '20100', '20101', '20300', '20202', '20200', '20700', '20802', '20801', '20803', '20804', '50005', '10605', '10604']
- Elementos generados: 147.
- Tiempo medio de inserción: 1.362 e^{-8} segundos.
- Tiempo de ejecución: 0.159 segundos.

3.4.3 Problema de complejidad avanzada.

Recorrido para el problema de complejidad avanzada.

- Origen: 40902.
- Destino: 10807.
- Coste: 51096.
- Camino: ['40902', '40904', '40907', '40909', '40604', '40603', '40707', '40706', '40911', '40710', '40921', '40710', '40913', '40302', '40303', '40304', '40305', '40306', '40407', '40408', '20403', '20404', '20405', '40309', '40308', '40307', '40407', '40406', '40409', '40400', '20400', '20401', '20402', '20100', '20101', '20300', '20202', '20200', '20700', '20802', '20801', '20803', '20805', '20806', '20804', '50005', '50004', '50003', '50012', '21200', '21302', '21303', '21300', '50001', '10905', '10904', '10903', '10815', '10808', '10807']
- Elementos generados: 280.
- Tiempo medio de inserción: 3.576 e^{-7} segundos.
- Tiempo de ejecución: 0.282 segundos.

3.5 Soluciones para la búsqueda en profundidad iterativa.

A continuación se va a aplicar el recorrido en profundidad iterativa para cada uno de los problemas propuestos.

3.5.1 Problema de complejidad sencilla.

Recorrido para el problema de complejidad sencilla.

- Origen: 10504.
- Destino: 10401.
- Coste: 3469.6.
- Camino: ['10504', '10503', '10502', '10501', '10401']
- Elementos generados: 23.
- Tiempo medio de inserción: 3.524 e^{-7} segundos.
- Tiempo de ejecución: 0.024 segundos.

3.5.2 Problema de complejidad media.

Recorrido para el problema de complejidad media.

- Origen: 359.
- Destino: 10604.
- Coste: 12909.60.
- Camino: ['359', '355', '356', '357', '40501', '333', '40500', '40101', '334', '319', '309', '50010', '50011', '10307', '10603', '10604']
- Elementos generados: 122.
- Tiempo medio de inserción: 8.207 e^{-7} segundos.
- Tiempo de ejecución: 0.153 segundos.

3.5.3 Problema de complejidad avanzada.

Recorrido para el problema de complejidad avanzada.

- Origen: 40902.
- Destino: 10807.
- Coste: 24995.20.
- Camino: ['40902', '40904', '40907', '40909', '40604', '40603', '40608', '40506', '40600', '40501', '333', '40500', '40101', '40103', '20900', '20902', '20903', '21001', '20808', '20805', '20807', '20802', '20801', '20800', '50003', '10902', '10903', '10815', '10808', '10807']
- Elementos generados: 200.
- Tiempo medio de inserción: 7.033 e^{-7} segundos.
- Tiempo de ejecución: 0.295 segundos.

3.6 Soluciones para el recorrido mediante costo uniforme.

A continuación se va a aplicar el mediante en costo uniforme para cada uno de los problemas propuestos.

3.6.1 Problema de complejidad sencilla.

Recorrido para el problema de complejidad sencilla.

- Origen: 10504.
- Destino: 10401.
- Coste: 3469.6.
- Camino: ['10504', '10503', '10502', '10501', '10401']
- Elementos generados: 68.
- Tiempo medio de inserción: 1.612 e^{-7} segundos.
- Tiempo de ejecución: segundos.

3.6.2 Problema de complejidad media.

Recorrido para el problema de complejidad media.

- Origen: 359.
- Destino: 10604.
- Coste: 9676.
- Camino: ['359', '371', '367', '352', '327', '326', '317', '307', '50011', '10307', '10603', '10604']
- Elementos generados: 397.
- Tiempo medio de inserción: 6.245 e^{-7} segundos.
- Tiempo de ejecución: 0.476 segundos.

3.6.3 Problema de complejidad avanzada.

Recorrido para el problema de complejidad avanzada.

- Origen: 40902.
- Destino: 10807.
- Coste: 16300.8
- Camino: ['40902', '40901', '40908', '40910', '40916', '40402', '40401', '40400', '20400', '20300', '21304', '21303', '21300', '50001', '10703', '10802', '10806', '10807'].
- Elementos generados: 588
- Tiempo medio de inserción: 2.513 e^{-8}
- Tiempo de ejecución: 0.818 segundos.

3.7 Soluciones para el recorrido mediante A*.

A continuación se va a aplicar el recorrido mediante A* para cada uno de los problemas propuestos. Para obtener la valoración final de la solución se han tenido en cuenta los aspectos de distancia, número de bares y número de comercios. El atributo de distancia euclídea se ha usado para generar el camino y una vez que generamos el camino la solución se evalúa ponderando los tres factores que utilizábamos inicialmente.

3.7.1 Problema de complejidad sencilla.

Recorrido para el problema de complejidad sencilla.

- Origen: 10504.
- Destino: 10401.
- Coste: 3469.6.
- Camino: ['10504', '10503', '10502', '10501', '10401']
- Elementos generados: 49.
- Tiempo medio de inserción: 3.892 e^{-7} segundos.
- Tiempo de ejecución: 0.059 segundos.

3.7.2 Problema de complejidad media.

Recorrido para el problema de complejidad media.

- Origen: 359.
- Destino: 10604.
- Coste: 9676.
- Camino: ['359', '371', '367', '352', '327', '326', '317', '307', '50011', '10307', '10603', '10604']
- Elementos generados: 390.
- Tiempo medio de inserción: 4.890 e^{-7} segundos.
- Tiempo de ejecución: 0.403 segundos.

3.7.3 Problema de complejidad avanzada.

Recorrido para el problema de complejidad avanzada.

- Origen: 40902.
- Destino: 10807.
- Coste: 16300.8
- Camino: ['40902', '40901', '40908', '40910', '40916', '40402', '40401', '40400', '20400', '20300', '21304', '21303', '21300', '50001', '10703', '10802', '10806', '10807'].
- Elementos generados: 590.
- Tiempo medio de inserción: 1.171 e^{-8} segundos.
- Tiempo de ejecución: 0.748 segundos.

3.8 Soluciones para búsqueda local mediante temple simulado.

A continuación se va a aplicar el recorrido mediante temple simulado para cada uno de los problemas propuestos. Cabe destacar que el algoritmo se ha inicializado con una solución bastante mala y no siempre saca la solución más óptima. Los resultados los genera a partir de la solución inicial. Si ponemos una solución inicial que se acerca a la óptima el algoritmo se esfuerza más. El tiempo de ejecución depende básicamente del factor con el que decrementamos la temperatura.

3.8.1 Problema de complejidad sencilla.

Recorrido para el problema de complejidad sencilla. Aquí el primer nodo sucesor generado formaba parte del camino óptimo.

- Origen: 10504.
- Destino: 10401.
- Coste: 5124.
- Camino: ['10504', '10505', '10504', '10503', '10502', '10501', '10401']
- Tiempo de ejecución: 10.6 segundos.

3.8.2 Problema de complejidad media.

Recorrido para el problema de complejidad media.

- Origen: 359.
- Destino: 10604.
- Coste: 10386.40
- Camino: ['359', '355', '345', '335', '328', '320', '326', '317', '307', '50011', '10307', '10603', '10604']
- Tiempo de ejecución: 54.138 segundos.

3.8.3 Problema de complejidad avanzada.

Recorrido para el problema de complejidad avanzada.

- Origen: 40902.
- Destino: 10401.
- Coste: 16300.8
- Camino: ['40902', '40901', '40908', '40910', '40916', '40402', '40401', '40400', '20400', '20300', '21304', '21303', '21300', '50001', '10703', '10802', '10806', '10807']
- Tiempo de ejecución: 109.957 segundos.

4. Conclusiones y comparaciones.

Una vez que están hechas las pruebas se pueden extraer distintas conclusiones sobre cada uno de los métodos aplicados. La búsqueda en anchura genera caminos medianamente óptimos de una manera sencilla aunque el tiempo que emplea para generar la solución es parecido al de costo uniforme o A*. Por lo tanto si se quiere generar un camino óptimo minimizando la distancia recorrida se recomienda usar costo uniforme o A* para problemas complejos. Por otro parte tenemos el recorrido en profundidad, este método expande el nodo más profundo hasta que encuentra la solución y normalmente genera soluciones bastante pésimas, a pesar de todo, los métodos basados en la profundidad son los que más rápido generan solución si tenemos en cuenta la complejidad del camino generado. Por ejemplo un recorrido en A* o costo uniforme emplea poco menos de un segundo para generar un camino de menos de treinta nodos mientras que la profundidad puede generar una solución de más de cincuenta nodos en menos de medio segundo. Otra método de profundidad que se ha aplicado para generar soluciones es el llamado profundidad acotada. Este método rechaza soluciones que están por debajo de un límite de profundidad, el problema es que el usuario tiene que estimar un límite de profundidad adecuado si se quiere generar una solución que tenga una optimalidad aceptable. Existen casos en los que un límite de profundidad adecuado puede generar la solución óptima. Por otra parte, si no se desea estimar un límite de profundidad se recomienda utilizar el método de profundidad iterativa. La profundidad iterativa explora todos los nodos que se encuentran a determinadas profundidades secuencialmente. Es un algoritmo rápido que genera soluciones aceptables, sin embargo, si el usuario desea un recorrido óptimo se recomienda utilizar la búsqueda A* o el recorrido por costo uniforme. Estos dos últimos algoritmos son los que más tiempo emplean para generar la solución pero siempre proporcionan una solución óptima a nuestro problema. Por último tenemos el recorrido por enfriamiento simulado. Este algoritmo de búsqueda local puede resultar útil para cuando se desee mejorar una solución propuesta. El algoritmo toma una solución inicial pésima y trata de mejorarla, la optimalidad de la solución generada depende de la temperatura inicial y del factor de enfriamiento. La temperatura está ajustada para que el algoritmo no emplee más de 3 minutos en tiempo de ejecución.

5. Valoración de la metodología utilizada.

La realización del trabajo en pasos pequeños a contribuido a la mejora de la versión final. Dedicar unas horas cada dos semanas para cubrir un aspecto del problema implica que no se descuiden detalles importantes. Por otra parte se considera una buena idea que todos compartamos el mismo grafo así podemos comparar los resultados entre los compañeros. Por último considero que el concepto de la lista ordenada para implementar los algoritmos ha ayudado a la implementación de la práctica.

Anexo 1. Código fuente.
A continuación se va a detallar el código fuente de la práctica, sólo se ha incluido el código correspondiente a la implementación de los algoritmos ignorando el código de la interfaz gráfica.

Clase estado.
Esta clase almacena información sobre el estado

```
class estado:

    def __init__(self, IDVertex = '', profundidad = 0, costo = 0.0,
                 euclidea = 0.0):

        self.IDVertex = IDVertex
        self.profundidad = profundidad
        self.costo = costo
        self.euclidea = euclidea

    def IDvertex(self):
        return self.IDVertex

    def getProfundidad(self):
        return self.profundidad

    def getCosto(self):
        return self.costo

    def getEuclidea(self):
        return self.euclidea

    def setIDvertex(self, IDVertex):
        self.IDVertex = IDVertex

    def setProfundidad(self, profundidad):
        self.profundidad = profundidad

    def setCosto(self, costo):
        self.costo = costo

    def setEuclidea(self, euclidea):
        self.euclidea = euclidea
```

Clase nodo.
Esta clase almacena información sobre el nodo y su estado asociado.

```
import sys
from estado import estado
class nodo:
    def __init__(self, criterio = 0, father = None, IDVertex = '',
                 profundidad = 0, costo = 0.0, euclidea = 0.0):

        if criterio == 0:                           # anchura
            self.valor = float(profundidad)
        elif criterio == 1 or criterio == 2 or criterio == 3:
            # profundidades
            self.valor = -1.0 * float(profundidad)
        elif criterio == 4:                         # costo uniforme
            self.valor = costo
        elif criterio == -4:                        # costo(Maximizar)
            self.valor = -1.0 * costo
        elif criterio == 5:                         # A*
            self.valor = costo + euclidea
        elif criterio == -5:                        # A*(Maximizar)
            self.valor = -1.0 * costo
        elif criterio == 6:                         # Temple simulado
            self.valor = costo
        else:
            print "Criterio no válido"
            sys.exit(-1)

        self.father = father
        self.state = estado(IDVertex, profundidad, costo, euclidea)

    def getValor(self):
        return self.valor

    def getFather(self):
        return self.father

    def getState(self):
        return self.state

    def setValor(self, valor):
        self.valor = valor

    def setFather(self, father = None):
        self.father = father
    def setState(self, state = None):
        self.state = state
```

Clase coordenada.
Código correspondiente a la clase coordenada.

```
class coordenada:
    def __init__(self, vertice = '', latitud = 0.0, longitud = 0.0):

        self.vertice = vertice
        self.latitud = latitud
        self.longitud = longitud

    def Vertice(self):
        return self.vertice
    def Latitud(self):
        return self.latitud
    def Longitud(self):
        return self.longitud
```

Clase solución.
Código correspondiente a la clase solución.

```
class solucion:
    def __init__(self, camino = [], valoracion = 0.0):

        self.camino = camino
        self.valoracion = valoracion

    def getCamino(self):
        return self.camino

    def getvaloracion(self):
        return self.valoracion

    def setCamino(self, camino):
        self.camino = camino

    def setValoracion(self, valoracion):
        self.valoracion = valoracion
```

Clase algoritmos.
Código correspondiente a la clase algoritmo. Esta clase implementa los métodos de búsqueda.

```
import sys
import csv
import time
import random
import math
from nodo import nodo
from coordenada import coordenada
from solucion import solucion
from pygraph.classes.graph import graph
from bisect import bisect_left
from bisect import insort

class algoritmos:

    def __init__(self):

        self.l = []        # Lista ordenada
        self.keys = [None] # Lista de claves
        # Atributos para recoger estadísticas
        self.nelementos = 0.0
        self.SumTime = 0.0
        self.Tinicio = 0.0
        self.encontrado = 0
        self.tiempo = 0.0
        self.destino = ''
        self.tMedio = 0.0
```

```
def buscar(self, g, src, dst, lim, modo): # Método para buscar

    camino = []  # Lista en donde guardamos los nodos
    algoritmos.__inicializarGlobals(self)
    llave = 0
    self.encontrado = 0
    self.destino = dst

    # preparativos para nuestra lista ordenada
    elegido = nodo(modo, None, src, 1, 0.0, 0.0)
    self.l.append(elegido)
    self.keys[0] = self.l[0].valor

    # inicializo la lista de visitados
    visitados = []
    visitados.append(elegido.state.IDVertex)

    while len(self.l) != 0:
        llave = 0
        # Coger el 1°
        elegido = self.l[0]
        camino.append(elegido)
        # Eliminar el 1°
        self.l.remove(elegido)
        self.keys.remove(elegido.valor)

        if elegido.state.IDVertex == dst: # Si el nodo el es destino
            if modo == 3: # condición para abortar la profundidad iterativa
                self.encontrado = 1
            sol = solucion([], 0.0)
            sol.camino, sol.valoracion = algoritmos.__trayecto(self, modo,
                                                               camino, g)
            return sol # Objeto de la clase solucion

        else:

        # Condiciones necesarias para el caso que haya límite de profundidad
            if (modo == 2 or modo == 3) and elegido.state.profundidad < lim:
                llave = 1
            if modo == 0 or modo == 1 or modo == 4 or modo == 5 or modo < 0:
                llave = 1

            if llave:
                # Lista de sucesores sin visitados
                sucesores = algoritmos.__listaVecinos(self, g,
                                                      elegido.state.IDVertex,
                                                      visitados)
                # Insertar sucesores
                for sucesor in sucesores:
                    algoritmos.__insertar(self, g, elegido, sucesor, modo)
                    visitados.append(sucesor)

    if modo == 3 and self.encontrado == 0:
        sol = algoritmos.buscar(self, g, src, dst, lim + 1, modo)
        return sol
```

```
def templeSimulado(self, g, src, dst):
     T = 100.0
     alpha = 0.6
     it = 2
     solActual = solucion([], 100000.0) # Incial...
     solCandidata = solucion([], 0.0)
     solFinal = solucion([], 0.0)
     # Solución inicial
     nodoActual = nodo(6, None, src, 0, solActual.valoracion, 0.0)
     # Solucion Inicial(Final)
     solFinal.camino = []
     solFinal.valoracion = 100000.0
     t0 = time.time();
     # Temple simulado ...
     while T > 0.1:
         for i in range(it):
             # Selecciona un sucesor aleatorio
             cand = algoritmos.__randSucesor(self, g, nodoActual.state.IDVertex)
             # Se devuelve una solución vecina
             solCandidata = algoritmos.buscar(self, algoritmos.genGraph(self),
                                              cand, dst, 0, 5)
             # Creamos un tipo nodo que es el sucesor candidato
             nodoCandidato = nodo(6, nodoActual,
                                  cand,
                                  nodoActual.state.profundidad + 1,
                                  solCandidata.valoracion,
                                  0.0)
             # Devuelve la diferencia de valoraciones entre el nodo actual y el
             # candidato.
             # DE = - costeValoracionCandidata - (- costeValoraciónActual)
             DE = algoritmos.__difValor(self, solActual.valoracion,
                                        solCandidata.valoracion)
             # Si la valoración es mejor que la actual...
             if DE > 0:
                 # Actualizamos los valores de nodo actual y solución actual
                 solActual.camino = algoritmos.__anexionSol(self,
                                                    solActual.camino,
                                                    solCandidata.camino,
                                                    nodoActual.state.IDVertex)

                 solActual.valoracion = algoritmos.__evalSol(self, g,
                                                             solActual.camino)
                 nodoActual = nodoCandidato
             else:
                 # Se calcula el fator de probabilidad
                 u = random.randint(0, 1)
                 if u <= math.pow(math.e, DE / T):
                     # Actualizamos los valores de nodo actual y solución actual
                     solActual.camino = algoritmos.__anexionSol(
                                                    self,
                                                    solActual.camino,
                                                    solCandidata.camino,
                                                    nodoActual.state.IDVertex)

                     solActual.valoracion = algoritmos.__evalSol(self, g,
                                                                 solActual.camino)
                     nodoActual = nodoCandidato
```

```
    # Si es una solución más óptima la seleccionamos como final
    if solActual.valoracion < solFinal.valoracion:
        solFinal.camino = solActual.camino
        solFinal.valoracion = solActual.valoracion

    # Decrementar la temperatura por un factor alpha
    T = alpha * T
    # Incrementar el número de iteraciones
    it = it + 2
tf = time.time() - t0;
return solFinal, tf
```

```
# Método para insertar en la lista ordenada
def __insertar(self, g, padre, vertice, modo):
    # factor para maximizar o minimizar
    if modo >= 0:
        factor = 1.0
    else:
        factor = -1.0

    # calulo del costo
    costoArista = algoritmos.__calcularCosto(self, g, padre.state.IDVertex,
                                             vertice, factor)
    costo = padre.state.costo + costoArista

    # calculamos la distancia euclidea...
    distancia = algoritmos.__euclidea(self, vertice, self.destino)

    # se crea un tipo nodo para insertar en la lista
    nd = nodo(modo, padre, vertice, padre.state.profundidad + 1, costo,
              distancia)

    # INSERCIÓN EN LA LISTA
    tIniIns = time.time()
    pos = bisect_left(self.keys, nd.valor)
    self.l.insert(pos, nd)
    insort(self.keys, nd.valor)
    self.SumTime = + (time.time() - tIniIns)
    # FIN INSERCIÓN

    self.nelementos = self.nelementos + 1

# Método que anexiona a una solución actual una solución candidata a partir de
# un determinado nodo de la solución actual.
def __anexionSol(self, actual, candidata, vertice):
    l = []
    for elem in actual:
        if elem == vertice:
            break
        l.append(elem)
    l.append(vertice)
    return l + candidata

# Método que devuelve la valoración de una ruta
def __evalSol(self, g, camino):
    costo = 0.0
    for i in range(len(camino) - 1):
        costo += algoritmos.__calcularCosto(self, g, camino[i], camino[i + 1],
                                            1.0)
    return costo

# Método que devuelve un sucesor aleatorio
def __randSucesor(self, g, nodo):
    sucesores = g.neighbors(nodo)
    n = random.randint(0, len(sucesores) - 1)
    sucesor = sucesores[n]
    return sucesor
```

```
# Método que devuelve la diferencia de valoración de dos soluciones
def __difValor(self, valorActual, valorCandidato):
    return ((-1.0) * valorCandidato) - ((-1.0) * valorActual)


# Método que devuelve la distancia euclídea entre dos nodos.
def __euclidea(self, origen, destino):
    csvReader = csv.reader(open('nodes.csv', 'rb'))
    for row in csvReader:
        # Aquí se extraen las coordenadas del .csv
        if row[0] == origen:
            src = coordenada(row[0], math.fabs(float(row[1])),
                             math.fabs(float(row[2])))
        if row[0] == destino:
            dst = coordenada(row[0], math.fabs(float(row[1])),
                             math.fabs(float(row[2])))
    # El radio de la tierra...
    Rt = 6371.0
    # (X2 - X1), (Y2 - Y1) y pasarlos a radianes
    delta = math.fabs(dst.latitud - src.latitud) * math.pi / 180.0
    teta = math.fabs(dst.longitud - src.longitud) * math.pi / 180.0
    # Sustituir en la formula
    distancia = 2.0 * Rt * math.sqrt(math.pow(math.sin(delta / 2.0), 2) +
                                     math.pow(math.sin(teta / 2.0), 2))
    return 0.2 * (distancia * 1000) # devolver la distancia en metros ponderada

# Método que devuelve una lista de sucesores sin visitados
def __listaVecinos(self, g, n, visitados):
    vecinos = g.neighbors(n)
    # eliminar los visitados
    for vertex in vecinos:
        if visitados.count(vertex) != 0:
            vecinos.remove(vertex)

    return vecinos

# Método para calcular el coste acumulado de un nodo
def __calcularCosto(self, g, src, dst, factor):
    d = g.edge_weight((src, dst))
    attr = g.edge_attributes((src, dst))
    if factor > 0: # Maximizar o minimizar
        c = 0.8 * (d + 1000.0)
    else:
        c = (attr[0] + attr[1]) - d * 1000.0
    return c
# Método que inicializa los atributos de la clase en el caso de que se hagan
# búsquedas sucesivas.
def __inicializarGlobals(self):
    self.l = []
    self.keys = [None]
    self.nelementos = 0.0
    self.SumTime = 0.0
    self.Tinicio = time.time()
    self.encontrado = 0
    self.tiempo = 0.0
    self.destino = ''
    self.tMedio = 0.0
```

```
# Método que construye la solución a partir de todos los nodos que se han ido
# generando hasta que se llega al destino
def __trayecto(self, modo, camino, g):
    algoritmos.__verTipo(self, modo, camino)
    s = []
    camino.reverse()

    s.append(camino[0].state.IDVertex)
    n = camino[0].father

    while n != None:
        s.insert(0, n.state.IDVertex)
        n = n.father

    costo = algoritmos.__evalSol(self, g, s);
    print 'Costo Total: %s' %(costo)
    print s

    algoritmos.__stats(self)
    algoritmos.__separador(self)

    return s, costo

# Método que almacena las estadísticas de una búsqueda
def __stats(self):
    if self.nelementos == 0: # Evita la división entre cero en el caso que no
                                se genere nada
        self.nelementos = 1

    self.tiempo = time.time() - self.Tinicio
    self.tMedio = self.SumTime / self.nelementos
    # Estadísticas en consola
    print 'Estadísticas:'
    print 'Elementos generados: %d' %(self.nelementos)
    print 'Tiempo medio de inserción: %.4e segundos' %(self.tMedio)
    print 'Todo ello en: %.3f segundos' %(self.tiempo)

# Método que imprime en consola el método que estamos usando
def __verTipo(self, modo, camino):
    s = ''
    if modo == 0:   # anchura
        print 'Recorrido en Anchura'
    elif modo == 1: # profundidad
        print 'Recorrido en Profundidad'
    elif modo == 2: # profundidad acotada
        print 'Recorrido en Profundidad Acotada'
    elif modo == 3: # profundidad iterativa
        print 'Recorrido en Profundidad Iterativa'
    elif modo == 4 or modo == -4: # costo
        print 'Recorrido en Costo Uniforme'
    elif modo == 5 or modo == -5: # A*
        print 'Recorrido mediante A*'
    elif modo == 6: # Temple simulado
        print 'Recorrido mediante temple simulado'
```

```
# Método que imprime un separador en la consola.
def __separador(self):
    print '-------------------------------------------------------'

# Método utilizado para generar el grafo
def genGraph(self):
    g = graph()
    csvReader = csv.reader(open('arcs.csv', 'rb'))
    for row in csvReader:                       # Cada fila es una arista
        algoritmos.__addVertex(self, g, row)    # añadir vertice
        algoritmos.__addEdge(self, g, row)      # añadir arista
    return g

# Método para añadir nodos al grafo
def __addVertex(self, g, row):
    for i in range(2):
        if g.has_node(row[i]) == False:
            g.add_node(row[i])

# Método para añadir aristas al grafo
def __addEdge(self, g, row):
    attr = [0.0, 0.0]       # lista de dos elementos que contiene los bares y
                            # comercios
                            # l[0] -> bares, l[1] -> comercios
    d = float(row[2])       # distancia
    attr[0] = float(row[3]) # bares
    attr[1] = float(row[4]) # tiendas
    label = row[5]          # nombre de la calle
    g.add_edge((row[0], row[1]), d, label, attr)
```