Entregable 3

El problema del viajante de comercio en un grafo euclídeo

EI1022
Algoritmia

Mihaita Lupoiu
Pedro Segarra Cabedo
Sergio Varea Aguilella

Índice

a) Memoria del entregable.

En este entregable se pide la resolución del problema del viajante de comercio en un grafo euclídeo. Dado un conjunto de puntos en el plano que representa un grafo euclídeo totalmente conectado, se quiere obtener un ciclo hamiltoniano, es decir encontrar un camino que parta de un vértice, termine en el mismo vértice y visite todos los vértices una sola vez (excepto, el de partida, que también es el de llegada, y por tanto se visita dos veces).

Para resolver el problema del viajante de comercio euclídeo, se van a utilizar dos estrategias algorítmicas:

- La primera de ellas es la de búsqueda con retroceso, en el caso de grafos pequeños, permite obtener el ciclo hamiltoniano óptimo. Con esta estrategia se obtendrán todos los ciclos hamiltonianos del grafo, y calculando la distancia que se recorre para cada uno de ellos, devolveremos la de menor longitud.

- La segunda estrategia a utilizar es la de utilizar algoritmos voraces. Dado que la primera estrategia – búsqueda con retroceso – el coste del algoritmo es exponencial, no se pueden usar sobre grafos de tamaño mediano o grande.
  Aquí usaremos un esquema algorítmico de resolución voraz, intentando hallar una solución cuyo valor sea una aproximación razonable a la óptima.

a.1) Resolución con la estrategia algorítmica de búsqueda con retroceso.

Para la realización de este apartado abordamos el entregable como los anteriores, leyendo información del libro de la asignatura, y concretamente fuimos a la página 67 donde se mostraba un algoritmo de búsqueda con retroceso para resolver un ciclo hamiltoniano.

Una vez leído el apartado del libro y entendido el algoritmo, decidimos realizar la función de lectura del fichero.

```
def readFile(fichero):
    aristas = []
    vertices = []
    index = 0

    for line in open(fichero,encoding="utf-8"):
        tupla=line.split(" ")
        if len(tupla) == 1:
             vertices = [0]*int(tupla[0])
        else:
            vertices[index] = (float(tupla[0]),float(tupla[1]))
            index += 1

    for i in range(len(vertices)):
        for j in range(i+1,len(vertices)):
            aristas.append((vertices[i],vertices[j]));

    return aristas,vertices
```

Con esta función devolvemos una lista de aristas y vértices.

Ejemplo de funcionamiento, dado el fichero p00.vjt

```
5
0 0
0.1 0.5
0 1
1 1
1 0
```

La primera dificultad que experimentamos con la lectura del fichero, es que nos daba error al realizar la misma, no estábamos entendiendo bien el fichero de entrada, y no realizábamos de forma adecuada la lectura.

Una vez comprendido la estructura de entrada del fichero, realizamos correctamente la lectura. A continuación mostramos una captura para el fichero p00.vjt

La función readFile, devuelve la siguientes listas, las aristas para el grafo y los vértices:

```
aristas → [((0.0, 0.0), (0.1, 0.5)), ((0.0, 0.0), (0.0, 1.0)), ((0.0, 0.0), (1.0, 1.0)),
((0.0, 0.0), (1.0, 0.0)), ((0.1, 0.5), (0.0, 1.0)), ((0.1, 0.5), (1.0, 1.0)), ((0.1,
0.5), (1.0, 0.0)), ((0.0, 1.0), (1.0, 1.0)), ((0.0, 1.0), (1.0, 0.0)), ((1.0, 1.0), (1.0,
0.0))]
[(0.0, 0.0), (0.1, 0.5), (0.0, 1.0), (1.0, 1.0), (1.0, 0.0)]

vértices → [(0.0, 0.0), (0.1, 0.5), (0.0, 1.0), (1.0, 1.0), (1.0, 0.0)]
```

Para evitar que una arista no se pruebe consigo misma, evitar autociclo lo que no nos interesa. Entonces realizamos lo siguiente, partimos de i+1 en el segundo for.

```
for i in range(len(vertices)):
    for j in range(i+1,len(vertices)):
```

Una vez leído el fichero creamos un grafo no dirigido:

```
grafo=UndirectedGraph(E=aristas)
```

Y a continuación llamamos a la función:

```
soluciones = HamiltonianCycleSolver().find(grafo)
```

Aquí hemos modificado la clase de la librería algoritmia HamiltonianCycleSolver, creando una función find. Y modificando también el solve de la misma.

```
class HamiltonianCycleSolver:#[solver
    def __init__(self):
        self.enumerator = BacktrackingEnumerator(createSolution
                                                 =lambda space, i, d, f: f + (f[0],))

    def solve(self, G):
        space = HamiltonianCycleStateSpace(G)
        return self.enumerator.enumerate(space) #]solver
    def find(self,S):
        sol=HamiltonianCycleSolver().solve(S)
        return sol
```

Esta clase realiza el backtracking para realizar el ciclo hamiltoniano, hemos modificado la de la librería algoritmia por el siguiente motivo:

- Hemos añadido el método **find**, el cual invoca al solve. Hacemos esto por comodidad para llamarla fuera con el programa principal.
- Y el método **solve** se ha modificado para que devuelva todas las posibles soluciones. La de la librería algoritmia devuelve la primera solución que encuentra, pero a nosotros nos interesan todas las soluciones para calcular de entre ellas la de longitud mínima.

Una vez obtenido las posibles soluciones, las pasamos a la función mejorResultado para obtener el ciclo hamiltoniano de menor longitud.

```
solucion = mejorResultado(soluciones)
```

Aquí se llama a la función mejorResultado. Esta función realiza una llamada a la función calculaDistancia, que devolverá la distancia que se recorre en esa solución. Haciendo una comparativas en mejorResultado de las distancias sabemos qué solución nos ofrece el mejor resultado, es decir el ciclo hamiltoniano de longitud mínima.

```
def mejorResultado(soluciones):
    solucion = next(soluciones)
    distanciamin = calculaDistancia(solucion)

    for t in soluciones:
       distancia=calculaDistancia(t)
       if distancia < distanciamin:
           solucion = t
           distanciamin = distancia
    return solucion
```

Y la función calculaDistancia, devuelve pasada una posible solución, la longitud de la distancia. Para que luego mejorResultado se quede con la mínima.

```
def calculaDistancia (solucion):
    i = 0
    j = 1
    distancia = 0
    while(j<len(solucion)):
        distancia+=sqrt(((solucion[i][0]-solucion[j][0])*(solucion[i][0]-solucion[j][0]))
+((solucion[i][1]-solucion[j][1])*(solucion[i][1]-solucion[j][1])))
        i += 1
        j += 1
    return distancia
```

Para realizar el cálculo de la distancia, hemos realizado el cálculo basándonos en la distancia euclídea entre dos puntos.

$$d_E(P_1, P_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Debemos importar sqrt para realizar la raíz cuadrada.

Con estos cálculos ya tenemos el ciclo hamiltoniano con la distancia de longitud mínima. Pero como en el entregable se especifica que mostremos por pantalla una lista de enteros separados por blancos que representen los índices de los vértices del ciclo hamiltoniano óptimo. Debemos realizar los pasos siguientes:

```
lista = []
contador = 0
for i in solucion:
     if contador != len(solucion) - 1:
         lista.append(vertices.index(i))
     else:
         break
     contador += 1
```

En la lista devolvemos los índices de los vértices, para mostrarlos por pantalla. Cómo destacable, recorremos hasta `len(solucion) - 1` ; le restamos 1 porque el testeo de la solución no quiere que guardemos en la lista el vértice de llegada, que es el mismo que el de partida.
Además cómo se quiere en un formato específico para pasar las pruebas, realizamos lo siguiente para formatear la solución.

```
impResultado = ""
for imp in lista:
    impResultado += "{}".format(imp)
    impResultado += " "
print(impResultado.rstrip())
```

La solución para el ejemplo p00.vjt es la siguiente:

```
2 1 0 4 3
```

Y el penúltimo paso que hemos realizado, ha sido el de guardar la solución de los ficheros de las diferentes pruebas, y realizar a continuación la validación de los mismos.

Para finalizar hemos generado el fichero sol_bt.txt requerido en la práctica.

El coste del algoritmo temporal, depende del número de vértices. O (|V|!) , por tanto hay |V| ! ciclos hamiltonianos posibles. Como indica el enunciado de la práctica tenemos un coste exponencial, y no podemos usarlo sobre grafos de tamaño mediano ni grandes.

A continuación mostramos la resolución con la estrategia voraz.

a.2) Resolución con la estratégica voraz.

Para la resolución de este apartado también partimos del libro de teoría, esta vez del capítulo 8 , estrategia voraz, concretamente la página 54 de este capítulo.

Estuvimos varios días dando vueltas para ver como afrontarlo con Kruskal, pero con la modificación del entregable en su versión 2, decidimos seguir la estrategia mostrada en el entregable v.2.0.

Para ello hemos modificado el _traverse del DepthFirstTraverser(), ordenando los sucesores de cada vértice antes de recorrerlos. En las dos llamadas a G.succs(..), hemos añadido sorted como comentaba el entregable.

Y realizamos la implementación como sigue:

```
        diccionario,vertices=readFile(sys.argv[1])
```

La primera función **readFile,** leemos los puntos del fichero y devolvemos un diccionario, donde las claves son las aristas. Aquí lo que hemos guardado son los índices i,j, en lugar del vector[i] y el vector[j]. De esta manera podemos sacar los índices para la entrega.
El valor del diccionario, es la distancia entre los puntos. Esta distancia la hemos calculado mediante la función **distancia.**

```
def readFile(fichero):
    diccionario={}
     aristas=[]
    vertices=[]
    index = 0

    for line in open(fichero,encoding="utf-8"):
        tupla=line.split(" ")
        if len(tupla) == 1:
             vertices = [0]*int(tupla[0])
        else:
            vertices[index] = (float(tupla[0]),float(tupla[1]))
            index+=1
```

```
    for i in range(len(vertices)):
        for j in range(i+1,len(vertices)):
            distan=distancia(vertices[i],vertices[j])
            diccionario[(i,j)]=distan
            diccionario[(j,i)]=distan

    return diccionario,vertices
```

La función readFile, es similar a la función de estrategia de búsqueda con retroceso, salvo que aquí empleamos el diccionario, para lo que hemos comentado anteriormente. Además devolvemos una lista de vértices como realizábamos en retroceso.

La función **distancia** calcula la distancia euclídea entre dos puntos. Esta función también está detallada para el caso de estrategia de búsqueda con retroceso y realiza el mismo cometido.

```
def distancia(punto1,punto2):
    return sqrt(((punto1[0]-punto2[0])*(punto1[0]-punto2[0]))+((punto1[1]-punto2[1])*(punto1[1]-punto2[1])))
```

A continuación realizamos lo siguiente:

```
d = WeightingFunction(diccionario)
G = UndirectedGraph(E=d.keys())
MST = list(PrimsMinimumSpanningFinder().minimum_spanning_forest(G, d))
grafoPrim=UndirectedGraph(E=MST)
```

Aquí creamos el grafo pasándole la implementación del algoritmo de Prim.

```
solucion=mejorSolucion(grafoPrim)
```

Con la función **mejorSolucion**, se hace una llamada a DepthFirstTraverser(), y aquí es donde se elige el mejor de los recorridos por primero en profundidad en preorden, probando todos los vértices como inicio del recorrido.

```
def mejorSolucion(grafoPrim):
    j=0
    for inici in grafoPrim.V:
        distan=0
        solucio=list(DepthFirstTraverser().traverse(grafoPrim, inici))
        if (j==0):
            mejorDistancia=0
            for i in range(len(solucio)-1):
                mejorDistancia+=diccionario[(solucio[i],solucio[i+1])]
            mejorDistancia+=diccionario[(solucio[len(solucio)-1],solucio[0])]
            solucion=solucio

        else:
            for i in range(len(solucio)-1):
                distan+=diccionario[(solucio[i],solucio[i+1])]
            distan+=diccionario[(solucio[len(solucio)-1],solucio[0])]
                if distan<mejorDistancia:
                mejorDistancia=distan
                solucion=solucio
        j+=1
    return solucion
```

Con esta función calculamos la longitud menor, que es la que queremos, y sólo falta darle formato correcto para que el valida lo admita.

b) Actas de las reuniones.

Acta nº1.-

Reunión celebrada el día 18 de Noviembre de 2013 a las 9.00 en cabina sala de estudio.

Asistentes:
- Mihaita Lupoiu (moderador)
- Pedro Segarra Cabedo(responsable edición)
- Sergio Varea Aguilella(secretario)

Orden del día

1.- Puesta en común del desarrollo del entregable.
2.- Resolución de entrada de datos.

Desarrollo de la Sesión

1.- Puesta en común del desarrollo del entregable

Aportamos nuestro punto de vista sobre el mismo, pero decidimos que la mejor manera para empezar a trabajar con él es cargar los ficheros de ejemplo y crear un grafo con los puntos.

2.- Resolución de entrada de datos.

Esta vez la resolución de la entrada de datos la realizamos de forma diferente a las otras veces. Todos colaboramos con Mihai para la lectura de los datos, Mihai los explica todos los pasos que realiza y cada uno de nosotros lo desarrollamos en nuestro portátil.

Tuvimos el primer inconveniente porque no nos dimos cuenta de cómo venían los datos en el fichero y nos empezó a dar fallos de lectura. Estuvimos toda la sesión sin poder avanzar en más detalles.

Levantamos la sesión a las 11.00.

Tiempo dedicado a la reunión 2 .
Tiempo dedicado al entregable (fuera de la reunión)
Mihai Lupoiu 2 horas.
Pedro Segarra 2 horas.
Sergio Varea 3 horas.

Castellón a 18 de Noviembre de 2013.

Fdo. Sergio Varea

Acta nº 2

Reunión celebrada el día 25 de Noviembre de 2013 a las 8.30 en cabina sala de estudio.

Asistentes:

- Mihaita Lupoiu (moderador)
- Pedro Segarra Cabedo(responsable de edición)
- Sergio Varea Aguilella(secretario)

Orden del día

1.- Puesta en común del desarrollo del entregable
2.- Resolución de apartado a por parte de Sergio.
3.- Comienzo del apartado b.

Desarrollo de la Sesión

1.- Puesta en común del desarrollo del entregable

Teníamos problema con la lectura del fichero del entregable, pero hablando en el seminario con el profesor, supimos que no estabamos leyendo bien el fichero. A raíz de ahí Sergio trajo resuelto el apartado 1.

2.- Resolución de apartado a por parte de Sergio.

Sergio le ha dedicado horas a la resolución del apartado a del entregable y lo ha traído resuelto. Nos ha comentado como ha llegado a la solución y nos ha explicado las funciones y la lectura del fichero correctamente.

3.- Comienzo del apartado b.

Hemos leído en el libro de teoría que se puede aplicar kruskal para la resolución del

entregable y nos ponemos a buscar una solución al mismo

Levantamos la sesión a las 11.00

Tiempo dedicado a la reunión 2horas y media.

Tiempo dedicado al entregable (fuera de la reunión)
Mihaita Lupoiu 3 horas.
Pedro Segarra 2,5 horas.
Sergio Varea 8 horas.

Castellón a 25 de Noviembre 2013.

Fdo. Sergio Varea.

Nota.- Hemos estado realizando reuniones ocasionales fuera del horario dedicado exclusivamente a Algoritmia. Después de la versión 2 del entregable, vimos que la opción mejor era realizar los pasos como indican en la versión segunda enviada por los profesores.

Sergio Varea es quién ha llevado este entregable adelante, y el chequeo de la solución ha estado elaborado por Mihai. Pedro se ha dedicado a realizar de la forma mejor posible la memoria del mismo.

c) Valoración personal.

Mihaita Lupoiu: "*Mi opinión personal sobre este entregable es que en la realización del backtraking la cosa fue relativamente fácil. Hemos tenido pequeños fallos pero más o menos lo teníamos todos claros que se tenía que hacer. En cambio en el greedy las pruebas que yo había intentado no me han funcionado del todo bien, pero no estaba lejos de la solución que tenemos.*
Me hubiera gustado dedicar más tiempo a este entregable, pero he tenido que dedicar tiempo a otras asignaturas y no me ha sido posible.
"

Pedro Segarra: " *El entregable ha sido el que he entendido de forma más sencilla, además había bastante documentación en el libro. El apartado a, lo he tenido bastante claro, y con las explicaciones de Sergio lo tengo claro. El apartado b, gracias a la versión 2 del entregable, hemos visto como realizarlo".*
"Lo más negativo es la falta de tiempo para realizar el entregable, ya no por la duración del mismo, sino por la carga de trabajo que se acumula entre proyectos de otras asignaturas y parciales".

Sergio Varea: *" En mi opinión estos ejercicios han estado muy bien y aunque tenían cierta dificultad, esto los hacía más interesantes". Estos ejercicios me han ayudado a entender mejor como funcionan los algoritmos de backtracking y de voraces"*