Tarea 3 Optimización de Flujo de Redes

Evely Gutiérrez Noda

8 de abril de 2018

1. Introducción

En el siguiente reporte se abordará el tema estudiado en la clase de Optimización de Flujo de Redes. En dicha clase continuamos el estudio sobre la construcción de grafos, utilizando lenguaje Python [1] para la programación cuando queremos crear un grafo, además utilizamos la herramienta Gnuplot [2], la cual vinculamos con Python para representar el grafo antes programado.

Esta vez en la clase se realizaron experimentos con el algoritmo de **Floyd-Warshall** y **FordFulkerson**.

2. Descripción del trabajo en clase

En la tarea anterior ya quedo programado un grafo con ciertas características, ahora se utilizará este grafo que se creó, para trabajar con el algoritmo **FloydWarshall** y **FordFulkerson**.

Partiendo de la **clase Grafo** que ya teníamos, donde se definieron los parámetros **n**(nodos), **x**, **y** (vértice **x** y vértice **y** respectivamente), un conjunto **E**, donde más tarde se almacenará cada par de vértices que serán unidos por una arista, el color de esta arista y el peso correspondiente a la misma. También se creó el parámetro **destino**, donde se almacenará el nombre del archivo en el cual se guardan los nodos y aristas creados.

Para el trabajo con estos algoritmos fue necesario crear en la estructura de la **clase Grafo**, un conjunto para almacenar los nodos vecinos a la hora de conectar cada uno de ellos con una arista, de modo que la **clase Grafo** quedó definida de la siguiente forma:

```
class Grafo:

    def __init__(self):
        self.n = None
        self.x = dict()
        self.y = dict()
        self.E = []
        self.destino = None
        self.vecinos = dict()#Nuevo conjunto para guardar los nodos
            vecinos
        self.i = None
```

Luego dentro de la **clase Grafo**,que ya contenía funciones para la programación de un grafo, se agregaron los algoritmos de **FloydWarshall** para caminos cortos y **FordFulkerson** para flujo máximo. Se realizaron algunos cambios en el código utilizado en la Tarea 2 [[3]] para poder trabajar con estos algoritmos, ya que anteriormente no se trabajaba con un arreglo donde se guardarán los nodos vecinos, y este arreglo fue necesario crearlo para poder utilizar estos algoritmos como se mencionó antes.

El algoritmo **FloydWarshall** es un algoritmo de análisis sobre grafos para encontrar el camino mínimo en grafos dirigidos ponderados. El algoritmo encuentra el camino entre todos los pares de vértices en una única ejecución. Esto es semejante a construir una tabla con todas las distancias mínimas entre pares de ciudades de un mapa, indicando además la ruta a seguir para ir de la primera ciudad a la segunda. Este es uno de los problemas más interesantes que se pueden resolver con algoritmos de grafos.

Este algoritmo trabaja comparando todos los posibles caminos a través del grafo entre cada par de vértices, esto lo hace mejorando paulatinamente una estimación del camino más corto entre dos vértices, hasta que se sabe que la estimación es óptima. El algoritmo usa la metodología de Programación Dinámica para resolver el problema.

A continuación, se muestra como quedó modificado el algoritmo para poderlo utilizar en el grafo que se crea con el código de la Tarea 2. [[3]]

```
def FloydWarshall(self):
      d = {}
      for nodo in range(self.n - 1):
          d[(nodo, nodo)] = 0 # distancia reflexiva es cero
          for (vecino, peso) in self.vecinos[nodo]: # para vecinos, la
               distancia es el peso
              d[(nodo, vecino)] = peso
      for intermedio in self.vecinos:
          for desde in self.vecinos:
              for hasta in self.vecinos:
                  di = None
                  if (desde, intermedio) in d:
                      di = d[(desde, intermedio)]
                  ih = None
                  if (intermedio, hasta) in d:
                      ih = d[(intermedio, hasta)]
                  if di is not None and ih is not None:
                      c = di + ih # largo del camino via "i"
                      if (desde, hasta) not in d or c < d[(desde,
                           hasta)]:
                          d[(desde, hasta)] = c # mejora al camino actual
      return d
```

El algoritmo de **FordFulkerson** propone buscar caminos en los que se pueda aumentar el flujo, hasta que se alcance el flujo máximo. La idea es encontrar una ruta con un flujo positivo neto que una los nodos origen y destino. Su nombre viene dado por sus creadores.

Existe un flujo máximo que viaja desde un único lugar de origen hacia un único lugar de destino a través de arcos que conectan nodos intermediarios. Los arcos tienen una capacidad máxima de flujo, y se trata de enviar desde la fuente

al destino la mayor cantidad posible de flujo.

Este algoritmo se utiliza para reducir los embotellamientos entre ciertos puntos de partida y destino en una red. por ejemplo: en Sistemas de Vías Públicas, Transporte de petróleo desde la refinería hasta diversos centros de almacenamiento, Distribución de energía eléctrica a través de una red de alumbrado público, etc.

A continuación se muestra como quedó el algoritmo **FordFulkerson** luego de los cambios realizados para poderlo utilizar con las características de los grafos creados en la clase **Grafo**.

```
    def FordFulkerson(self, s, t): # algoritmo de Ford y Fulkerson
        if s == t:
            return 0
        maximo = 0
        f = dict()
        while True:
            aum = self.camino(s, t, f)
            if aum is None:
                break # ya no hay
            incr = min(aum.values(), key = (lambda k: k[1]))[1]
            u = t
            while u in aum:
                v = aum[u][0]
                actual = f.get((v, u), 0) # cero si no hay
                inverso = f.get((u, v), 0)
                f[(v, u)] = actual + incr
                f[(u, v)] = inverso - incr
                u = v
            maximo += incr

        return maximo
```

Se realizaron varias pruebas con varios grafos de distintas características para medir el tiempo de ejecución de estos algoritmos. Estos grafos tenían una ponderación aleatoria con valores entre uno y diez. Uno de los ejemplos se realizó con un grafo ponderado **G1** con tamaño inicial de nodos **(n)** igual a 90, y el otro grafo ponderado **G2**, con tamaño inicial de nodos igual a 100. Los valores que arrojaron los algoritmos **FloydWarshall** y **FordFulkerson** se muestran en el resumen siguiente, donde los flujos máximos fueron $\mathbf{G1 = 48}$ y $\mathbf{G2 = 21}$. Los tiempos de ejecución de estos algoritmos se midió en segundos, desde el instante en que fueron llamadas las funciones que los definen hasta que arrojaran un resultado, en este ejemplo arrojaron tiempos de nueve segundos para el **G1** y de un segundo para el **G2**.

```
Valores de N mas 10 :
90
prueba.txt
Resultado de FloydWarshall en cantidad de conjuntos :
8010
Tiempo de FloydWarshall:
0.7996770825935364
Tiempo de FordFulkerson:
```

```
4.868632634034739
FordFulkerson desde el inicio hasta n - 1
275

------------------------------------------------------------
Valores de N mas 10 :
120
prueba.txt
Resultado de FloydWarshall en cantidad de conjuntos :
14280
Tiempo de FloydWarshall:
1.8796005775237887
Tiempo de FordFulkerson:
15.671588563978176
FordFulkerson desde el inicio hasta n - 1
368
```

Este trabajo con los algoritmos se realizó con varios grafos de distintos tamaños y algunos del mismo tamaño, se fue aumentando en cinco en cada corrida el tamaño de los grafos, se midió el tiempo de ejecución para cada vez que se corrían los algoritmos.Se realizaron pruebas de corridas de hasta treinta veces con tamaños de grafos que llegaron hasta ciento cincuenta, y en cada una se almacenaron los tiempos de ejecución para cada algoritmo en un fichero de texto que se utilizó para representar los resultados en un diagrama que se muestra en la figura 1.

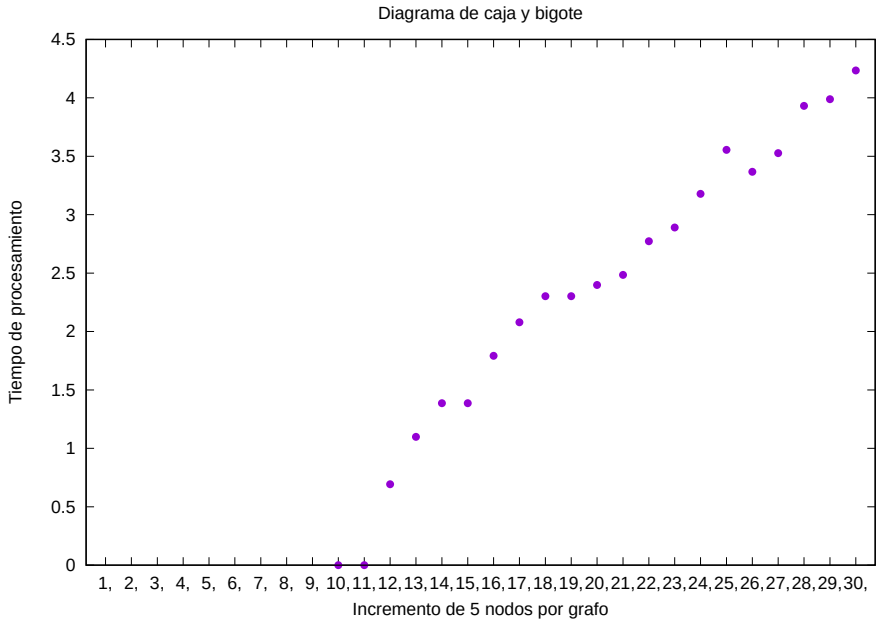

Figura 1: Diagrama Caja y Bigote.

Referencias

[1] Python Software Fundation, www.python.org/

[2] Gnuplot, www.gnuplot.info

[3] Tarea2, https://github.com/EvelyGutierrez/Optimizacion-de-flujo-de-redes/blob/master/Tarea2/VersionFinal/Tarea%202.pdf