Optimización de Flujo en Redes: Medición experimental de la complejidad asintótica de los algoritmos Ford-Fulkerson y Floyd-Warshall

Alcantar Gómez Alan Arnoldo

9 de abril de 2018

1. Introducción

En el presente trabajo se explican las modificaciones que se tuvieron que realizar a los códigos reportados en el reporte 2 [2] para poder implementar los códigos de Ford-Fulkerson y Floyd-Warshall, con el objetivo de realizar mediciones experimentales de la complejidad asintótica de ambos algoritmos y compararlas con la teoría. Por último, mencionar que los códigos están escritos en *Python* y se utilizó *Gnuplot* para graficar los resultados.

2. Descripción de los algoritmos Ford-Fulkerson y Floyd-Warshall

2.1. Ford-Fulkerson

Sea $G(V, E)$ un grafo, con V vértices, E aristas y donde por cada par de vértices (u, v), tenemos una capacidad $c(u, v)$ y un flujo $f(u, v)$. Lo que se busca es maximizar el valor del flujo desde una fuente s hasta un sumidero t. El método inicia con $f(u, v) = 0$ para todo (u, v) en V y en cada iteración, se incrementa el flujo en G mediante el resultado de la busqueda del camino de aumento minimo que existe en la red residual G_f. El código para cuando no existen más caminos aumentantes en la red residual G_f y regresa el flujo máximo encontrado [3]. En este caso las entradas que necesita el algoritmo son las aristas E del grafo y los puntos fuente y sumidero, el algoritmo se puede encontrar en la referencia [1].

2.2. Floyd-Warshall

Es un algoritmo básico para REACHABILITY que computa los caminos más cortos de un grafo ponderado, donde los pesos tienen que ser no negativos.

El algoritmo construye de una manera incremental estimaciones a los caminos más cortos entre dos vértices hasta llegar a la solución óptima. El algoritmo etiqueta los vértices V de un grafo $G(V, E)$ de la forma $V = \{1, 2, ..., n\}$. Por medio. A través de una función $C(i, j, k)$ construye el camino más corto entre los vértices i y j pasando solamente por vértices con etiqueta $\leq k$ [4]. Lo único que necesita el algoritmo es el conjunto de vértices V y las aristas E del grafo $G(V, E)$, el algoritmo se puede encontrar en la referencia [1].

3. Modificaciones al código para poder implementar los algoritmos

En nuestro código primero fue necesario generar el conjunto de vértices porque se utilizaría en el código Floyd-Warshall.

```
def nodos(self, n, t):  #'n' es el numero de nodos y 't' es la
    temperatura maxima del sistema
   self.vertices.add(i)
```

Si nos fijamos bien en el código Floyd-Warshall pasa por todos los vecinos de cada uno de los vértices.

```
def floy_warshall(self):      #no se le da de comer
   d = {}
   for v in self.vertices:
      d[(v,v)] = 0 #la distancia reflexiva es cero
      for u in self.vecinos[v]: #para vecinos, la distancia es el peso
         d[(v,u)] = self.aristas[(v,u)]
```

Por lo tanto, cuando se genere el grafo dependiendo de qué tipo sea definirá la forma en que se guardan las aristas y pesos. Por ejemplo, si el grado es simple ponderado y el par de vértices (i, j) se conectan, también se debe conectar el par (j, i) porque en este caso no importa la dirección y da la mismo ir de i a j que dé j a i y por lo mismo ambos se deben agregan como vecinos uno del otro. En caso de generar un grafo dirigido ponderado si el par de vértices (i, j) se conectan, solamente se debe agregar j como vecino de i porque está la posibilidad que el par (j, i) no se conecten.

```
def conectar(self, n, modo):  #'n' es el numero de nodos y 'modo' puede
    ser 'simple', 'dirigido', 'ponderado' (siempre debe ponerse entre
    comillas simples)
     if modo == 'simple' or modo == 'ponderado':
        for i in range(n-1):
           for j in range(i+1,n):
              if random() <= dis:
                 self.aristas[(i,j)] = peso
                 self.aristas[(j,i)] = peso
                 if not i in self.vecinos:
```

```
                self.vecinos[i] = set()
            if not j in self.vecinos:
                self.vecinos[j] = set()
            self.vecinos[i].add(j)
            self.vecinos[j].add(i)
    elif modo == 'dirigido' or modo == 'campechano':
        for i in range(n):
            for j in range(n):
                if random() <= dis:
                    self.aristas[(i,j)] = peso
                    if not i in self.vecinos:
                        self.vecinos[i] = set()
                    self.vecinos[i].add(j)
```

4. Experimentación y resultados

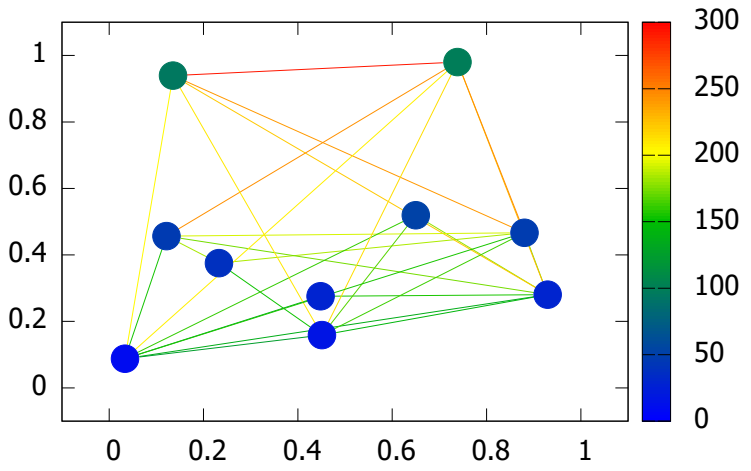

Figura 1: Grafo simple ponderado con 10 vértices.

En las figuras 4 y 4 podemos observar los tipos de grafo simple ponderado y dirigido ponderado que se pueden generar con nuestro código, en ellos el valor del peso se refleja con el color de las aristas que unen los vértices.

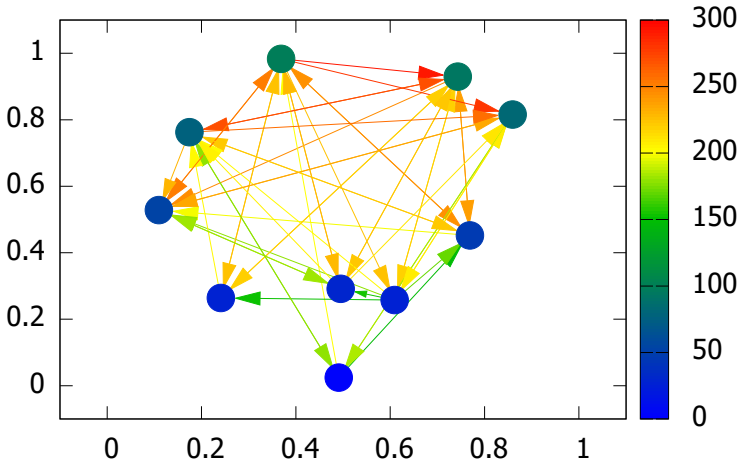

Figura 2: Grafo dirigido ponderado con 10 vértices.

Para realizar la experimentación se creó un archivo, donde el tamaño de cada instancia esta guardado en la lista i, los algoritmos Ford-Fulkerson y Floyd-Warshall se llamaron 15 veces para cada tamaño de instancia en i y los tiempos de computo eran guardados en los diccionarios tiempos ford y tiempos floy. Después se realizó una prueba de normalidad a los tiempos obtenidos en cada tamaño de instancia con la función *stats.shapiro*()y el valor de p se guardaba en p ford y p floy. Por último, se calcularon los cuartiles de los tiempos de cada tamaño de instancia utilizando la función *numpy.percentile*().

```
i = [20,40,60,80,100,120,140,160,180,200] #200
for n in i:
   for k in range(15):
      tiempos_ford[n].append(g.ford_fulkerson(s,t)[1])
      tiempos_floy[n].append(g.floy_warshall()[1])
for n in i:
   p_ford.append((n,stats.shapiro(tiempos_ford[n])[1]))
   p_floy.append((n,stats.shapiro(tiempos_floy[n])[1]))
for n in i:
   n,np.percentile(tiempos_ford[n],0),np.percentile(tiempos_ford[n],25),
   np.median(tiempos_ford[n]),np.percentile(tiempos_ford[n],75),
   np.percentile(tiempos_ford[n],100)
for n in i:
   n,np.percentile(tiempos_floy[n],0),np.percentile(tiempos_floy[n],25),
   np.median(tiempos_floy[n]),np.percentile(tiempos_floy[n],75),
   np.percentile(tiempos_floy[n],100)
```

En las figuras 4 y 4 se pueden observar los diagramas de caja y bigote para los tiempos de Ford-Fulkerson y Floyd-Warshall de cada tamaño de instancia. Se puede observar en ambas figuras que casi todas las instancias no presentan

una distribución uniforme y que el ajuste cubico $a * x * *3$ se ajusta de mejor manera a los promedios de los tiempos de Floyd-Warshall que a los promedios de Ford-Fulkerson.

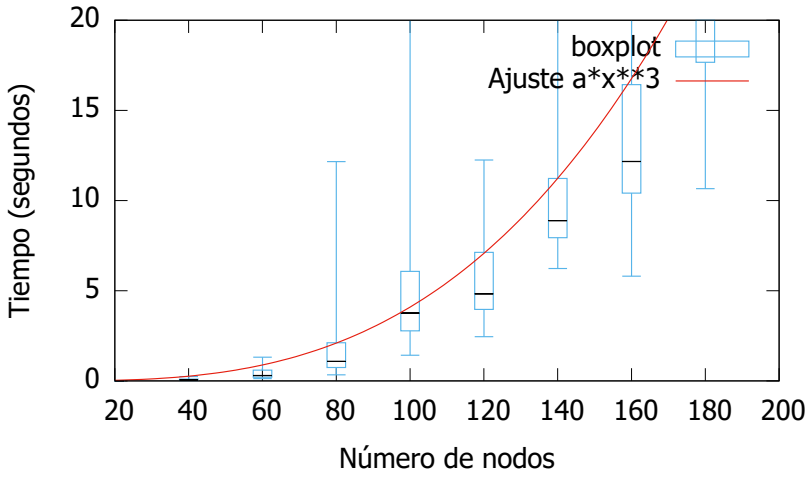

Figura 3: Diagramas de caja y bigotes para los tiempos de Ford-Fulkerson y ajuste de una curva cubica

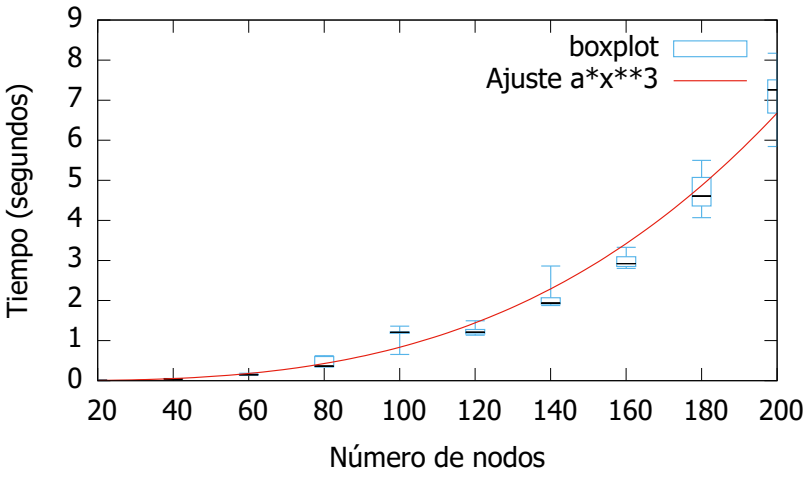

Figura 4: Diagramas de caja y bigotes para los tiempos de Floyd-Warshall y ajuste de una curva cubica

5. Conclusión

En el presente trabajo se implementaron los algoritmos Ford-Fulkerson y Floyd-Warshall en nuestro código para poder medir los tiempos de computo de ambos algoritmos y poder estimar la complejidad de ellos a través de un ajuste de curvas. La implementación de ambos códigos sirvió para comprender como es posible encontrar el flujo máximo en el corte de un grafo y obtener las distancias más cortas entre cualesquiera dos vértices del grafo. El ajuste de curva a los tiempos de computo de los algoritmos implementados, nos indica que la complejidad de estos es polinómica.

Referencias

[1] Alcantar G. Alan, *implementación de los algoritmos Ford-Fulkerson y Floyd-Warshall*, https://github.com/alan-arnoldo-alcantar/flujos/blob/master/Reporte

[2] Alcantar G. Alan, *Reporte 2*, https://github.com/alan-arnoldo-alcantar/flujos/blob/master/Reporte

[3] Wikipedia, *Algoritmo Ford Fulkerson*, https://es.wikipedia.org/wiki/Algoritmo de Ford-Fulkerson

[4] Wikipedia, *Algoritmo Floy Warshall*, https://es.wikipedia.org/wiki/Algoritmo de Floyd-Warshall