Medición experimental de la complejidad asintótica con `python` y `gnuplot`

Francisco Gerardo Meza Fierro

1. Introducción

En esta práctica se trabaja con el código que se encuentra en el mismo repositorio que este reporte [1], a éste se agregan las funciones de Floyd-Warshall y Ford-Fulkerson y se analizan con ellos los grafos que se crean; por último se medirán los tiempos de ejecución del código y se discutirán los resultados.

Igual que en la previa práctica, las imágenes que presenta está práctica fueron obtenidas a través de `gnuplot` y `R` [2] mediante archivos generados a través de `python`.

2. Implementación de los algoritmos

Antes de comenzar, se hace una observación: el vector `aristas` que antes se tenía fue reemplazado por un diccionario llamado `arcos`, el cual contiene tanto los nodos vecinos, como la arista creada con esos nodos y el peso que ésta tiene, esto con finalidad de tener un mejor acceso a la lectura y obtención de los datos, así como ahorrar unas cuántas líneas de código.

Para implementar los algoritmos, se tomaron como base los propuestos por la Doctora Elisa Schaeffer en la página de su curso en línea de Matemáticas Discretas [3] y ambos fueron modificados para conveniencia del código que se ha estado creando y modificando en las últimas prácticas.

2.1. Floyd-Warshall

Este algoritmo arroja como resultado los caminos más cortos que hay dentro de un grafo iniciando en un nodo y terminando en otro dentro del mismo. Al igual que el código propuesto por la Doctora, este algoritmo se escribió en una función llamada `floyd_warshall`.

La figura 1 ayuda a entender lo que arroja esta función.

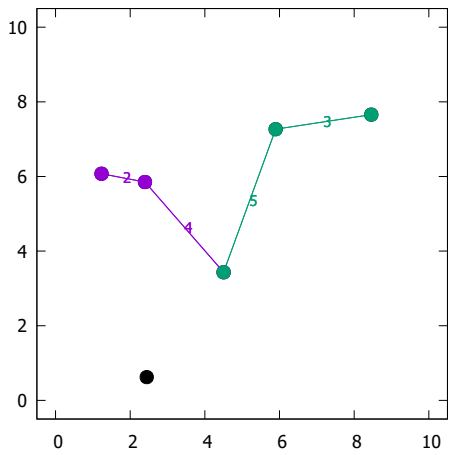


Figura 1: Un grafo no dirigido de 6 nodos.

Al mandar imprimir la función `floyd_warshall` se obtiene lo siguiente:

```
>>> G.floyd_warshall()
{((2.3963319559077236, 5.850828164493934), (2.3963319559077236, 5.850828164493934))
    : 0,
((2.3963319559077236, 5.850828164493934), (4.502935664110876, 3.433348250886316)):
    4,
((4.502935664110876, 3.433348250886316), (2.3963319559077236, 5.850828164493934)):
    4,
((2.3963319559077236, 5.850828164493934), (1.2318763374293729, 6.0724598165220245))
    : 2,
((1.2318763374293729, 6.0724598165220245), (2.3963319559077236, 5.850828164493934))
    : 2,
((5.8966419700369945, 7.267263025325613), (8.457020553683313, 7.657459980163711)):
    3,
((8.457020553683313, 7.657459980163711), (5.8966419700369945, 7.267263025325613)):
    3,
((5.8966419700369945, 7.267263025325613), (4.502935664110876, 3.433348250886316)):
    5,
((4.502935664110876, 3.433348250886316), (5.8966419700369945, 7.267263025325613)):
    5,
((5.8966419700369945, 7.267263025325613), (5.8966419700369945, 7.267263025325613)):
     0,
((8.457020553683313, 7.657459980163711), (8.457020553683313, 7.657459980163711)):
    0,
((2.443317209206021, 0.6218825151581664), (2.443317209206021, 0.6218825151581664)):
     0,
((4.502935664110876, 3.433348250886316), (4.502935664110876, 3.433348250886316)):
    0,
((1.2318763374293729, 6.0724598165220245), (1.2318763374293729, 6.072459816522024))
    : 0,
((4.502935664110876, 3.433348250886316), (1.2318763374293729, 6.0724598165220245)):
     6,
((1.2318763374293729, 6.0724598165220245), (4.502935664110876, 3.433348250886316)):
     6,
((8.457020553683313, 7.657459980163711), (4.502935664110876, 3.433348250886316)):
    8,
((4.502935664110876, 3.433348250886316), (8.457020553683313, 7.657459980163711)):
    8,
((2.3963319559077236, 5.850828164493934), (5.8966419700369945, 7.267263025325613)):
     9,
((2.3963319559077236, 5.850828164493934), (8.457020553683313, 7.657459980163711)):
    12,
((5.8966419700369945, 7.267263025325613), (2.3963319559077236, 5.850828164493934)):
     9,
((5.8966419700369945, 7.267263025325613), (1.2318763374293729, 6.0724598165220245))
    : 11,
((8.457020553683313, 7.657459980163711), (2.3963319559077236, 5.850828164493934)):
    12,
((8.457020553683313, 7.657459980163711), (1.2318763374293729, 6.0724598165220245)):
     14,
((1.2318763374293729, 6.0724598165220245), (5.8966419700369945, 7.267263025325613))
    : 11,
((1.2318763374293729, 6.0724598165220245), (8.457020553683313, 7.657459980163711)):
     14}
```

Ya que la figura 1 es un grafo muy pequeño, resulta tarea sencilla verificar que lo arrojado por esta función son los pesos que hay de ir de un nodo a otro dentro del grafo, donde la primer mitad mostrada resultan los pesos que hay entre nodos vecinos y la otra mitad corresponde al camino formado entre un par de nodos no vecinos.

Los arcos que tienen peso 0 son debido a que la función propuesta considera que el costo de ir de un nodo a sí mismo es cero. Esto es para evitar que se formen ciclos en cada nodo.

Es evidente que hay arcos repetidos, esto es debido a que, como no hay dirección, el código lo representa como dos arcos diferentes mientras que realmente se trata del mismo pero sin dirección.

La figura 2 ayuda a entender lo que arroja la función `floyd_warshall` cuando se tiene un grafo dirigido.

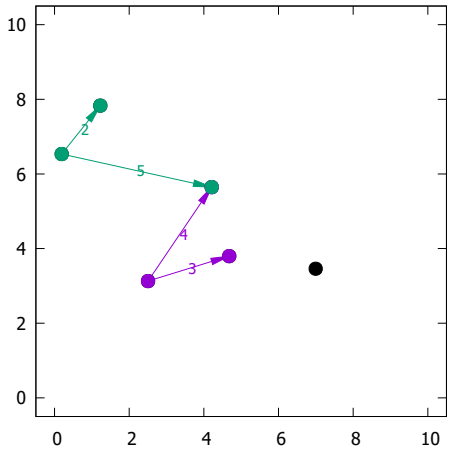


Figura 2: Un grafo dirigido de 6 nodos.

```
>>> G.floyd_warshall()
{((2.5040838290192555, 3.1297774003549494), (2.5040838290192555,
    3.1297774003549494)): 0,
((2.5040838290192555, 3.1297774003549494), (4.6806149764111735, 3.798405343462561))
    : 3,
((2.5040838290192555, 3.1297774003549494), (4.209260024364079, 5.648637895489027)):
    4,
((0.190619883498927, 6.5345711523322505), (1.2250082732229828, 7.831595883170258)):
    2,
((0.190619883498927, 6.5345711523322505), (4.209260024364079, 5.648637895489027)):
    5,
((0.190619883498927, 6.5345711523322505), (0.190619883498927, 6.5345711523322505)):
    0,
((4.6806149764111735, 3.798405343462561), (4.6806149764111735, 3.798405343462561)):
    0,
((6.995839698559441, 3.4589001482527926), (6.995839698559441, 3.4589001482527926)):
    0,
((1.2250082732229828, 7.831595883170258), (1.2250082732229828, 7.831595883170258)):
    0,
((4.209260024364079, 5.648637895489027), (4.209260024364079, 5.648637895489027)):
    0}
```

Resulta evidente que la cantidad de arcos es mucho menor a los arcos mostrados en un grafo no dirigido a pesar de que tanto la figura 1 como la figura 2 tengan cuatro arcos, ya que la figura 2 al ser un grafo dirigido no considera ambas direcciones de cada arco y la figura 1 sí.

2.2. Ford-Fulkerson

Este algoritmo arroja como resultado el flujo máximo que hay entre un nodo y otro dentro del mismo grafo recorriendo todos los caminos posibles por los que el flujo puede trasladarse desde el nodo inicial hasta el nodo final. Al igual que el código propuesto, este algoritmo se escribió en una función llamada `ford_fulkerson`.

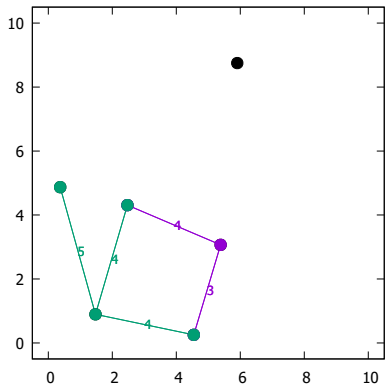
Se propuso que el nodo inicial sería el primer *nodo madre* que se crea y el nodo final sería el último *nodo madre* en crearse.

Para entender lo que arroja esta función tomando un grafo no dirigido, la figura 1 vuelve a ser útil.

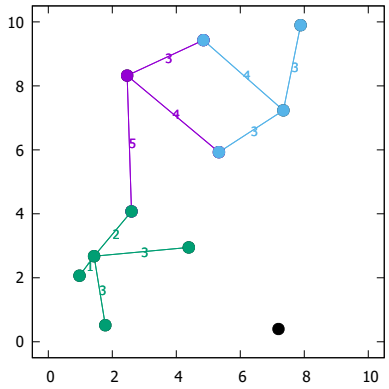
```
>>> G.ford_fulkerson()
4
```

Es evidente que el flujo máximo es 4, ya que solo hay un único camino (formado por dos aristas) que conecta a los *nodos madre* y aunque una arista tenga más capacidad que la otra, esta última no puede satisfacer la capacidad, por lo que el flujo máximo sería (en este caso) el de la arista con menor peso.

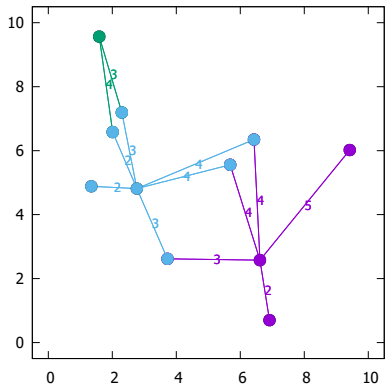
La figura 3 muestra más ejemplos de lo que la función `ford_fulkerson` arroja como resultado



(a) Grafo con flujo máximo de 7.



(b) Grafo con flujo máximo de 6.



(c) Grafo con flujo máximo de 11.

Figura 3: Diferentes grafos con 20 nodos.

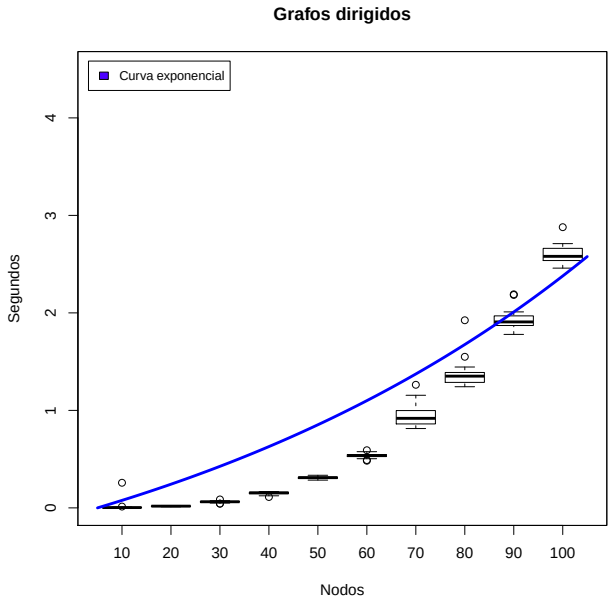
La figura 2 es también un buen ejemplo de lo que la función `ford_fulkerson` arrojaría como resultado en un grafo dirigido:

```
>>> G.ford_fulkerson()
0
```

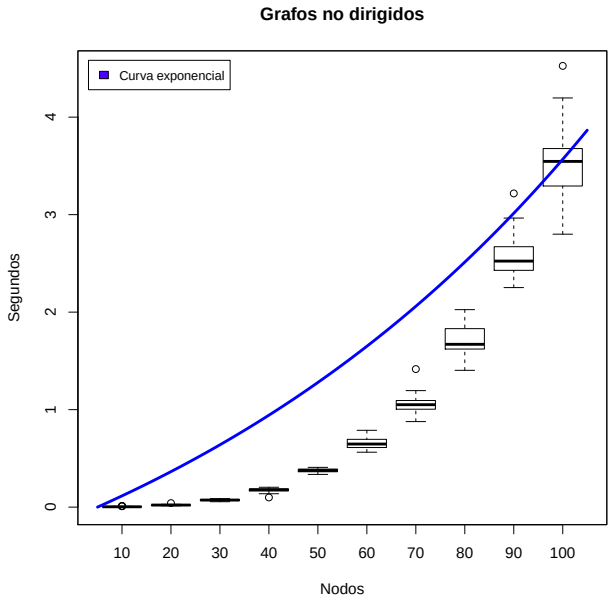
Este resultado es evidente ya que los únicos nodos que se conectan a otros son los *nodos madre* por lo que la única manera en que pudiese haber un camino dirigido desde el primer *nodo madre* al último sería que estos dos nodos estén conectados pero esto no podrá ser posible debido a la construcción de los grafos como lo fue explicado en la primer práctica [4]. Por lo tanto, este algoritmo resulta inútil usarlo con grafos dirigidos creados en estas prácticas.

3. Medición de tiempos

Para ver el tiempo de ejecución código, éste se corrió varias veces variando entre diez tamaños distintos de grafos y repitiendo treinta veces el código para cada tamaño de grafo, guardando en cada corrida el tiempo de ejecución con la ayuda de la función `time.clock()`. Todos los tiempos fueron guardados en un archivo *.csv* que se exportó a `R`, donde con ayuda de la función `boxplot()` se graficaron los tiempos, los cuales se muestran en la figura 4.



(a) Tiempos para grafos dirigidos.



(b) Tiempos para grafos no dirigidos.

Figura 4: Tiempos de las corridas.

Con estos resultados pueden resaltarse dos cosas:

- El incremento en el tiempo de ejecución evidentemente no es lineal, ya que pese a que el incremento en el número de nodos es lineal, el incremento en la cantidad de aristas que un nodo comparte con algún otro es mucho mayor que un incremento lineal que, según la figura 4, podría decirse exponencial ya que la curva experimental obtenida se asemeja mucho a la curva exponencial teórica.
- El tiempo de ejecución para grafos dirigidos es mucho menor que para grafos no dirigidos, ya que al ser dirigidos y como se mencionó al final de la sección pasada, las únicas aristas que el código revisa son las aristas formadas con los nodos vecinos al primer *nodo madre* y estas aristas al no estar conectadas con otras el código termina mucho antes que en el caso de que los grafos fueran no dirigidos ya que, análogamente, tiene más caminos por revisar.

Referencias

[1] Francisco Meza. *Visualización de grafos simples, ponderados y dirigidos con* `gnuplot` https://github.com/Cicciofano/FlujoEnRedes/tree/master/Tarea2, 2018.

[2] *The R Project for Statistical Computing* https://www.r-project.org/

[3] Satu Elisa Schaeffer *Matemáticas Discretas, Curso en línea* https://elisa.dyndns-web.com/teaching/mat/discretas/md.html

[4] Francisco Meza. *Representación de redes a través de la teoría de grafos en* `python` https://github.com/Cicciofano/FlujoEnRedes/blob/master/Tarea1, 2018.