

# Trabajo Practico 2

De Rosa - Schapira - Guerrero

Primer Cuatrimestre 2017

# Contents

<b>1</b>	<b>Clases de complejidad</b>	<b>1</b>
<b>2</b>	<b>Algoritmos de camino mínimo</b>	<b>3</b>
2.1	Comparación de algoritmos . . . . .	3
2.1.1	Dijkstra . . . . .	3
2.1.2	Floyd-Warshall . . . . .	3
2.1.3	Bellman-Ford . . . . .	3
2.1.4	Conclusiones . . . . .	3
2.2	Arbitrage . . . . .	4
2.3	Análisis individual de los algoritmos . . . . .	4
2.3.1	Dijkstra . . . . .	4
2.3.2	Floyd Warshall . . . . .	4
2.3.3	Bellman Ford . . . . .	5
<b>3</b>	<b>Comandos</b>	<b>6</b>

# 1 Clases de complejidad

A continuación se resolverán los seis problemas planteados, en caso de ser polinomiales se propondrá un pseudo-código del algoritmo que lo resuelva y en caso de ser NP-Completo se hará la reducción correspondiente.

1. **Se tiene un conjunto de  $n$  actividades para seleccionar. Cada actividad tiene asociados un tiempo de inicio y tiempo de fin. Se dice que un conjunto de actividades es compatible si no hay dos que se superpongan en un tiempo. Se pide un algoritmo que devuelva verdadero o falso de acuerdo a si se puede encontrar un subconjunto compatible de tamaño  $k$  o superior.**

Este problema pertenece a  $P$ , pues puede ser resuelto de la siguiente manera:

```
Funcion seleccionar_actividades ( ListaAct )
    actTiempoFin = ordenar segun tiempos de fin ( ListaAct )

    k = 0
    #se agarran todas las actividades menos la primera y la ultima
    for i in xrange(1, len(ListaAct)-1 ):
        si la actTiempoFin[ i ].inicio > actTiempoFin[ k ].fin :
            k=i
        else :
            return false

    return true
```

El algoritmo ordena las actividades en base al tiempo de finalización y luego se recorren  $n - 2$  actividades verificando la compatibilidad. Para que este algoritmo funcione en tiempo polinomial la lista de actividades debe ordenarse de manera eficiente. Por ejemplo, utilizando MergeSort que es, en su peor caso  $O(n \log(n))$ , siendo  $n$  la cantidad de elementos. Suponiendo que esto sea así, el ordenamiento será  $O(n \log(n))$  y el ciclo subsiguiente  $O(n)$ , resultando el algoritmo  $O(n \log(n))$  y, por lo tanto, perteneciente a  $P$ .

2. **Se tiene un conjunto de  $n$  actividades para seleccionar. Cada actividad tiene asociados un conjunto de tiempos de inicio y fin. Se dice que un conjunto de actividades es compatible si no hay dos que se superpongan en un tiempo. Se pide un algoritmo que devuelva verdadero o falso de acuerdo a si se puede encontrar un subconjunto compatible de tamaño  $k$  o superior.**
3. **En teoría de grafos, un camino hamiltoniano es un camino que visita cada vértice del grafo exactamente una vez. Se pide un algoritmo que indique si un grafo  $G$  tiene un camino hamiltoniano o no.**
4. **En teoría de grafos, un camino hamiltoniano es un camino que visita cada vértice del grafo exactamente una vez. Se pide un algoritmo que indique si un digrafo acíclico  $D$  tiene un camino hamiltoniano o no.**
5. **Se tiene un grafo dirigido y pesado  $G$ , cuyas aristas tienen pesos que pueden ser negativos. Se pide devolver verdadero o falso de acuerdo a si el grafo tiene algún ciclo con peso negativo.**

El problema propuesto puede ser resuelto utilizando el algoritmo de Bellman-Ford (descrito en la segunda parte de este trabajo) en tiempo polinómico, pues el algoritmo es  $O(|V||E|)$  siendo  $|V|$  la cantidad de vértices y  $|E|$  la cantidad de aristas, el cual, en el peor de los casos será  $|E| = |V|^2$  por lo que el algoritmo tendrá un orden  $O(n^3)$  con  $n = |V|$ . El algoritmo es el siguiente:

```
BellmanFord(Grafo, verticeInicial):
    # se definen inicialmente todas las distancias en INFINITO
    # menos la del verticeInicial que se define en 0
    for vertice in Grafo:
        distancias[vertice] = INFINITO
        padres[vertice] = None
    distancia[verticeInicial] = 0
    #se relajan todas las aristas
    for i=0...len(Grafo):
        #Se itera para todos los vertices del grafo
        for each arista(u, v) in aristas:
            if distancias[v] > distancias[u] + peso(u, v):
                distancias[v] = distancias[u] + W(u, v)
                padres[v] = u
```

```

#se verifica si hay ciclos negativo
for each arista(u, v) in aristas:
    if distancia[v] > distancia[u] + peso(u, v):
        #Hay ciclo negativo
        return True
return False

```

6. Se tiene un grafo dirigido y pesado  $G$ , cuyas aristas tienen pesos que pueden ser negativos. Se pide devolver verdadero o falso de acuerdo a si el grafo tiene algún ciclo con exactamente igual a cero.

Este problema es  $NP-C$  y lo demostraremos reduciendo el *Subset Sum Problem* ( $S$ ) a nuestro problema, que llamaremos  $Z$ . Demostrando así que  $S \leq_P Z$ .

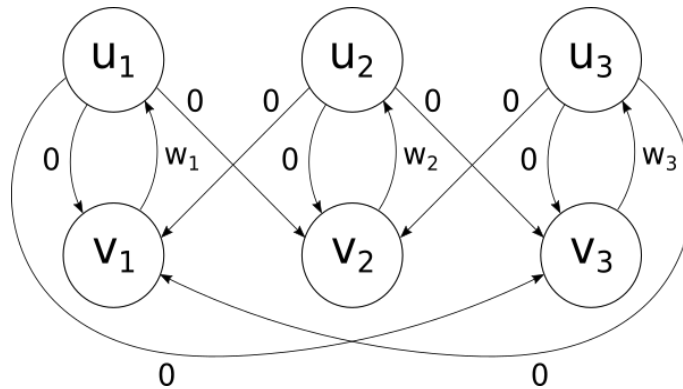
Supongamos que tenemos un problema  $S$  con una entrada  $W = \{w_1, \dots, w_n\} \subseteq \mathbb{Z}$  la cual es aceptada por  $S$  si existe un  $A \subseteq \mathbb{Z}$  tal que:

$$\sum_{a \in A} a = 0$$

Lo que haremos es resolver el problema  $S$  a través del problema propuesto. Para esto, construiremos un grafo  $G = (V, E)$  con  $2n$  vertices de la siguiente manera:

- Para cada elemento  $w_i$  creamos dos vertices:  $u_i$  y  $v_i$ .
- Para cada  $v_i$  agregamos la arista  $(v_i, u_i)$  a  $E$  con peso  $w_i$ .
- Para cada  $u_i$  y cada  $v_j$  agregamos la arista  $(u_i, v_j)$  a  $E$  con peso 0.

Por lo tanto, un problema  $S$  de tres elementos formaría un grafo de la siguiente forma:



Una vez que tenemos esto planteado, enviamos nuestro grafo a un hipotético algoritmo que resuelva el problema  $Z$  teniendo en cuenta que:

- Si hay un ciclo, pasa por al menos un elemento completo (esto es,  $u_i$  y  $v_i$ ).
- Al pasar por dicho elemento, la distancia total aumenta en  $w_i$  (pues los pesos son 0 y  $w_i$ ).
- Si un ciclo pasa por múltiples elementos, la distancia total será la suma de todos sus pesos.

Dicho esto, si el algoritmo que resuelve  $Z$  encuentra un ciclo de distancia 0, entonces existe un subconjunto de  $W$  que suma 0. Por lo tanto  $S \leq_P Z$  y entonces  $Z \in NP-C$ .

## 2 Algoritmos de camino mínimo

### 2.1 Comparación de algoritmos

Todos los algoritmos se basan en una operación llamada *relajación de aristas*. Relajar la arista  $v \rightarrow w$  significa verificar si la mejor manera de ir de  $s$  a  $w$  es  $s \rightarrow v \rightarrow w$  y, en tal caso, actualizar la información contenida en  $distancia[w]$  y  $padre[w]$ .

#### 2.1.1 Dijkstra

Este algoritmo encuentra el camino mínimo desde un vértice en particular hacia todo el resto de los vertices (*single-source shortest-path problem*).

Es un algoritmo Greedy y se basa en tomar siempre, entre todos los vértices adyacentes, el que esté más cerca del origen y ver si se puede llegar más rápido a través de este vértice a los demás actualizando las distancias a cada paso hasta que el vértice no utilizado más cercano sea el destino.

Es el más veloz de los tres algoritmos implementados con un tiempo asintótico de  $O(|E| \log(|V|))$  pero los pesos de las aristas deben ser *siempre*  $W \geq 0$ , en otro caso, falla.

#### 2.1.2 Floyd-Warshall

Este algoritmo resuelve el *all-pairs shortest path problem*, es decir, hallar los caminos mínimos entre todos los pares de vértices del grafo.

Utiliza la técnica de la Programación Dinámica y se basa en

Tiene un tiempo asintótico de  $O(|V|^3)$  pero si bien es el más lento de los tres algoritmos analizados permite, una vez que se ejecuta, adquirir el camino mínimo entre cualquier par de vértices del grafo en  $O(1)$ . Además, si bien se supone que el grafo no tiene ciclos negativos, se puede utilizar el algoritmo para detectarlos analizando la diagonal de la matriz de camino.

#### 2.1.3 Bellman-Ford

Utiliza la técnica de la Programación Dinámica para resolver el *single-source shortest-path problem*. El algoritmo se basa en repetir  $|V| - 1$  veces el proceso de relajación de aristas para las  $|E|$  aristas del grafo. Una vez realizado esto, se relajan las  $|E|$  aristas una vez mas. Si se obtiene un mejor resultado que antes, entonces el grafo tiene ciclos negativos.

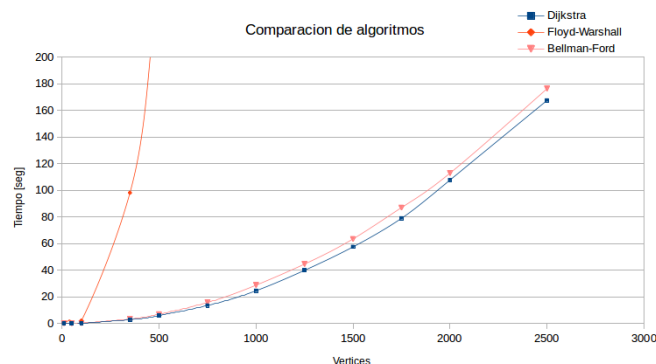
El tiempo asintótico de este algoritmo es, entonces,  $O(|V||E|)$ , que es más lento que Dijkstra pero permite detectar ciclos negativos.

#### 2.1.4 Conclusiones

Concluyendo esta sección, podemos decir que:

- Dijkstra es la mejor opción para resolver el *single-source shortest-path problem* cuando sabemos que no habrá ciclos negativos y todas las aristas tienen peso positivo.
- Bellman-Ford es la mejor opción cuando queremos detectar ciclos negativos o tenemos un grafo con aristas de peso negativo.
- Floyd-Warshall es la mejor opción cuando queremos resolver el *all-pairs shortest path problem* y además permite hallar ciclos negativos.

En cuanto a la relación de tiempos de ejecución, aquí hay un gráfico descriptivo:



## 2.2 Arbitrage

Para resolver este problema lo que se puede hacer es cambiar los pesos de cada arista  $W(E[u, v])$  por un nuevo valor  $W(E[u, v]) = -\text{Log}(W(E[u, v]))$  pues si nuestro objetivo es hallar un ciclo tal que  $w_1 * w_2 * \dots * w_k > 1$  entonces buscamos un ciclo tal que  $\text{Log}(w_1 * w_2 * \dots * w_k) > \text{Log}(1)$  y por ende, queremos un ciclo tal que  $-\text{Log}(w_1) - \dots - \text{Log}(w_k) < 0$ . Por lo tanto, si tenemos nuestros nuevos pesos, lo único que necesitamos es buscar ciclos negativos en nuestro grafo actualizado.

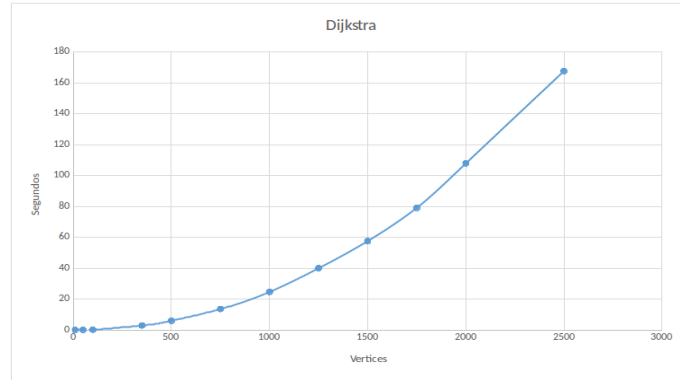
Para esto, como se mencionó en la sección anterior, se pueden utilizar tanto el algoritmo de Bellman-Ford como el de Floyd-Warshall.

## 2.3 Análisis individual de los algoritmos

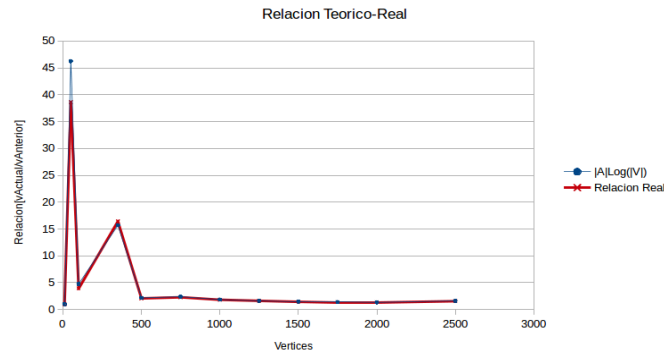
### 2.3.1 Dijkstra

Como se mencionó previamente, el tiempo asintótico de este algoritmo es  $O(|E| \text{Log}(|V|))$  y en la práctica los resultados son muy similares.

Aquí se muestra un gráfico del tiempo de ejecución en segundos, en función de la cantidad de vértices:



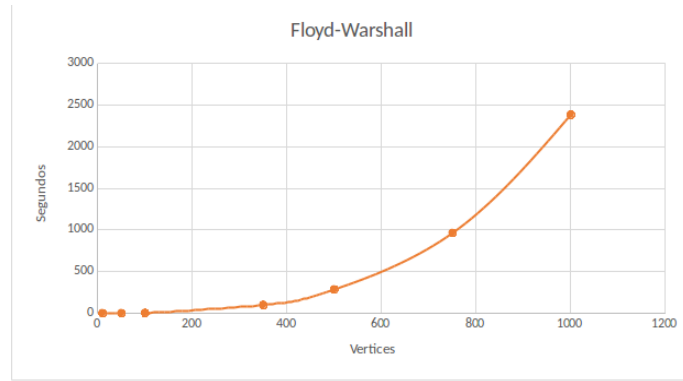
Y aquí se adjunta un gráfico en el que se compara, para los valores de  $|V|$  de cada ejecución, la relación de la función  $f(V, E) = |E| * \text{Log}(|V|)$  para el valor actual y el anterior, y la relación de tiempos de ejecución reales:



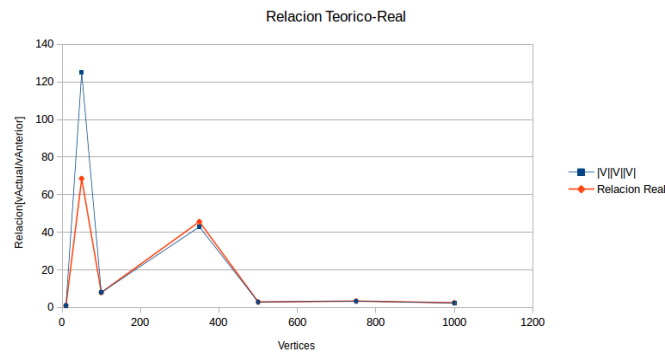
Este gráfico permite ver lo antes dicho: El tiempo de ejecución en la práctica es muy similar al teórico.

### 2.3.2 Floyd Warshall

Como se mencionó en la sección 2.1.2, el tiempo asintótico de este algoritmo es  $O(|V|^3)$  y en la práctica, al igual que Dijkstra, esta asíntota se respeta bastante. A raíz de esto, no fue posible ejecutar el algoritmo con muestras muy grandes ( $|V| > 1000$ ) pero se obtuvieron muestras suficientes para obtener esta información de la relación tiempo de ejecución - cantidad de vértices:



Y aquí se adjunta un gráfico en el que se compara, para los valores de  $|V|$  de cada ejecución, la relación de la función  $f(V, E) = |V|^3$  para el valor actual y el anterior, y la relación de tiempos de ejecución reales:



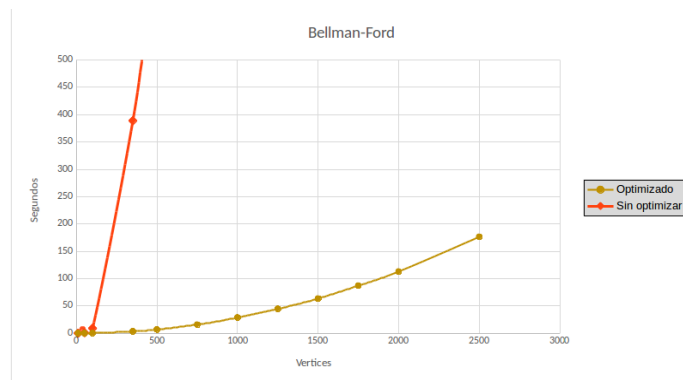
Este gráfico permite ver lo antes dicho: El tiempo de ejecución en la práctica es muy similar al teórico.

### 2.3.3 Bellman Ford

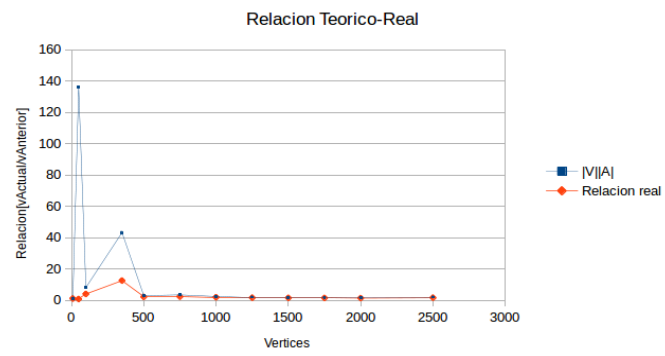
Como antes se ha dicho, el tiempo asintótico de este algoritmo es  $O(|V||E|)$ , esto en el caso tratado en este trabajo se traduce a  $O(|V|^3 - |V|^2)$ . De todos modos, existe una optimización que, si bien no baja el tiempo asintótico, mejora mucho el tiempo promedio en la práctica.

Esta optimización se basa en, simplemente, verificar a cada paso de las  $|V| - 1$  repeticiones si hubo alguna modificación en las distancias o padres de cada vértice y, en caso contrario, terminar la ejecución. De todos modos, cabe destacar que en caso de que haya ciclos negativos esta optimización no sirve de nada (pues siempre habrá un caso más óptimo que el anterior).

Aquí se puede ver un gráfico comparativo de la ejecución del algoritmo optimizado y sin optimizar, en ambos casos sin ciclos negativos:



Y aquí se adjunta un gráfico en el que se compara, para los valores de  $|V|$  de cada ejecución, la relación de la función  $f(V, E) = |V| * |E|$  para el valor actual y el anterior, y la relación de tiempos de ejecución reales:



En el gráfico de arriba se puede apreciar la optimización antes mencionada.

### 3 Comandos