Trabajo Practico 2

De Rosa - Schapira - Guerrero

Primer Cuatrimestre 2017

Contents

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-Completos 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 verificar_compatibilidad ( ListaAct )
    actTiempoFin = ordenar segun tiempos de fin ( ListaAct ):

    j = 0
    longitudSubconjuntos = []
    #se agarran todas las actividades
    for i in xrange(1, len(ListaAct)):
        si la actTiempoFin[ i ].inicio > actTiempoFin[ i-1 ].fin:
            j = ++
        else:
            heapq.(longitudSubconjuntos.append(j))
            j=0

    return longitudSubconjuntos


Funcion seleccionar_actividades(k):
    heap_finalizaciones = verificar_compatibilidad()
    maxima_longitud_subconjunto = lista_finalizaciones.obtenerMaximo()

    #si el maximo elemento no cumple, entonces ninguno cumple
    return (maxima_longitud_subconjunto >= k)
```

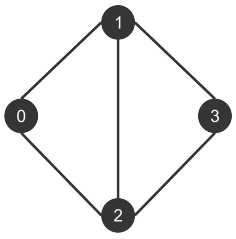
   El algoritmo ordena las actividades en base al tiempo de finalización y luego se recorren $n-1$ actividades verificando la compatibilidad. Para que este algoritmo funcione en tiempo polinomial la lista de actividades debe ordenarse de manera eficiente. Por ejemplo, utilizando MergeSort que es, en su peor caso $O(nLog(n))$, siendo n la cantidad de elementos.

   Suponiendo que esto sea así, el ordenamiento será $O(nLog(n))$ y el ciclo subsiguiente $O(n)$, resultando el algoritmo $O(nLog(n))$ y, por lo tanto, perteneciente a P. Vale aclarar que se utiliza un heap como estructura auxiliar con el objetivo de obtener el maximo eficientemente.

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.**

   Este problema pertenece a la familia de los $NP-C$ y esto se puede demostrar haciendo una reducción del problema de *k-Coloreo* de un Grafo (que llamaremos K), que es $NP-C$. Haciendo esto, demostraremos que nuestro problema (P) verifica $K \leq_P P$.

   Supongamos que tenemos el grafo G:



   A partir de las incidencias de cada vértice v_i, es decir, sabiendo a que otros vertices v_j es adyacente cada v_i podemos saber qué vértices no deben repetir colores.

   Sabiendo esto, se puede generar un diccionario donde se guarda la información sobre que conjuntos de

vértices compartiran un intervalo con inicio i_k y fin f_k, pues esos serán los vértices que no serán compatibles. El diccionario tendría la forma $\{[0,1,2] : [i_0, f_0], [1,3] : [i_1, f_1], [2,3] : [i_2, f_2]\}$ y se iría completando avanzando por el grafo y, para cada v_i, verificando si ya comparte un conjunto con todos sus vecinos y en caso contrario, creando uno nuevo, con un nuevo intervalo.

Una vez generado el diccionario, se procedería a identificar a cada v_i como una actividad a_i con los intervalos correspondientes. Esto resultaría en: $\{0 : [[i_0, f_0]], 1 : [[i_0, f_0], [i_1, f_1]], 2 : [[i_0, f_0], [i_2, f_2]], 3 : [[i_1, f_1], [i_2, f_2]]\}$

Finalmente, y con cada vértice adaptado al modelo de actividades del problema original, podríamos resolver el problema K procesando nuestras nuevas actividades con el algoritmo que resuelva el problema P original, para un k cualquiera.

Por lo tanto, $K \leq_P P$ y, entonces, $P \in NP - C$.

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.**

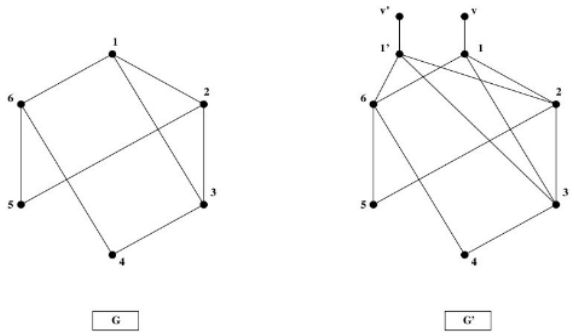
Richard Karp fue un informático teórico que desarrolló en 1972 una lista con 21 problemas $NP - Completos$. Dentro de esta lista, se encuentran:

- El problema del circuito Hamiltoniano Dirigido
- El problema del circuito Hamiltoniano NO Dirigido

A partir de tener estos conocimientos, se procede a reducir el problema en cuestión a uno de ellos, es decir, reducir el problema de un camino hamiltoniano al de un circuito. Para demostrar que el problema de encontrar un camino hamiltoniano es $NP - Completo$, primero se debe mostrar que pertenece a la clase NP.

Para un G dado se puede encontrar un camino hamiltoniano de manera no determinista, eligiendo aristas de G que son incluidas en el camino. Luego se recorre el camino y se asegura de que cada vértice es visitado exactamente una vez. Esto claramente se puede realizar en tiempo polinómico por lo tanto el problema en sí pertenece a NP.

Sabiendo que pertenece a NP, se procede a realizar la reducción a un "Circuito Hamiltoniano No Dirigido" siendo un circuito (en este caso) un camino hamiltoniano que empieza y termina en el mismo vértice. A partir de un grafo G=(V,E) se construye un grafo G' tal que G contiene un ciclo hamiltoniano si y sólo si G' contiene un ciclo hamiltoniano. Esto se realiza eligiendo arbitrariamente un vértice u (perteneciente a G) y a la vez añadiendo u' (perteneciente a G') con todas sus aristas. A continuación se añaden los vértices v y v' al grafo G' y se conecta por un lado v con u, y por otro v' con u' como muestra la siguiente figura:



Supongamos primero que G contiene un ciclo hamiltoniano. Entonces se obtiene un ciclo hamiltoniano en G' si se arranca por v, se continúa el ciclo que se obtuvo de G retornando a u' en vez de u y terminando el camino en v'. Por ejemplo, al observar la figura se puede apreciar que G tiene un ciclo hamiltoniano 1, 2, 5, 6, 4, 3, 1; por consiguiente el camino de G' corresponde a v, 1, 2, 5, 6, 4, 3, 1' , v'.

Inversamente, supongamos que G' contiene un camino hamiltoniano. En ese caso, el camino necesariamente tiene que tener sus extremos en v y v'. Este camino puede ser transformado a un ciclo en G. Si se descartan v y v' entonces los extremos del camino deben estar en u y u' y si se remueve u' se obtiene un ciclo en G cerrando el camino anterior en u en vez de u'.

Esta construcción no funcionaria si G es un grafo simple, por lo tanto este caso tiene que ser manejado por separado. En conclusión, se ha demostrado que G contiene un ciclo hamiltoniano si y sólo si G' contiene un camino hamiltoniano, probando así que el problema en cuestión es $NP - Completo$.

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.**

Dado que a todo Grafo Dirigido Acíclico (DAG) se le puede hacer un ordenamiento topológico tal que los vértices queden ordenados como $(v_1, v_2)(v_2, v_3)...(v_{n-1}, v_n)$ se ordena el grafo G de tal manera. Una vez logrado esto, se recorren todos los vértices verificando que haya una arista desde cada v_i a cada v_j, con $i < j$. Pues, dado que en el camino Hamiltoniano no se puede volver atrás, la única manera de recorrer todos los vertices es no salteandose ninguno.

A continuación se muestran los algoritmos para el ordenamiento topológico de un DAG y la solución al problema planteado:

```
funcion ordenTopologico(G):
    L = Lista vacia que contendra los elementos orenados
    S = Set con todos los vertices sin aristas entrantes
    while S no esta vacia:
        tomar un vertice v de S
        L.append(v)
        para cada vertice u con una arista(v, u):
            scar la arista(v, u) de G
            si u no tiene mas aristas entrantes:
                S.add(u)
    si G tiene aristas:
        return error (G tiene al menos un ciclo)
    else
        return L

funcion tieneCaminoHamiltoniano(G):
    #Primero se ordena a los vertices con un orden topologico (O(V+E))
    L = ordenTopologico(G)
    #Luego se verifica que exista una arista de cada vertice al siguiente (O(V))
    for i=0...len(L)-2:
        if ( no hay arista(L[i], L[i+1]) ):
            return False
    return True
```

Dado que el ordenamiento topólogico es $O(|V| + |E|)$ y el ciclo de verificación es $O(|V|)$ el algoritmo es polinómico y tiene un tiempo asintótico de $O(|V| + |E|)$.

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 (descripto en la segunda parte de este trabajo) en tiempo polinómico, pues el algoritmo es $O(|V||E|)$ siendo $|V|$ la cantidad de vertices 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 incialmente 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 reduciéndo el *Subset Sum Problem (S)* a nuestro problema, que llamaremos Z. Demostrando así que $S \leq_P Z$.

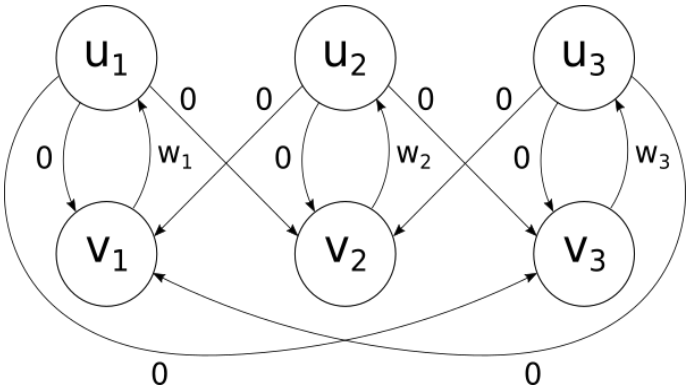
Supongamos que tenemos un problema S con una entrada $W = \{w_1, ..., w_n\} \subseteq \mathbb{Z}$ la cual es aceptada por S si existe un $A \subseteq \mathbb{Z}$ tal que:

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

. 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, vj) 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, envíamos 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 $Z \in NP - C$.

Ahora procedemos a demostrar que existe un verificador $V \in P$ para el problema dado. Es decir, existe un algoritmo de tiempo polinomial para determinar si un conjunto de aristas dado como supuesta solución al problema (certificado) es efectivamente una solución o no.

El siguiente algoritmo cumple con lo pedido:

```
funcion cicloCeroCorrecto( G, aristas ):
    pesoTotal = 0
    for arista in aristas:
        pesoTotal += peso(arista)
    if pesoTotal == 0: return true
    else: return false
```

Y el tiempo de este algoritmo es $O(|E|)$, con $G = (V, E)$. Por lo tanto, dada la existencia de $V \in P$, se demuestra que $Z \in NP$.

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->w` significa verificar si la mejor manera de ir de `s` a `w` es `s->v->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* $W >= 0$, 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 que cada iteracion se mejora el camino minimo entre dos vertices, hasta llegar al menor de todos. Este algoritmo trabaja directamente con la matriz de adyacencia del grafo y para cada v_i, busca el camino mínimo hasta la v_j suponiendo que solo se puede pasar por $\{v_0, ..., v_{k-1}\}$.

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 la—s $|E|$ aristas una vez mas. Si se obtiene un mejor resultado que antes, entonces el grafo tiene ciclos negativos.

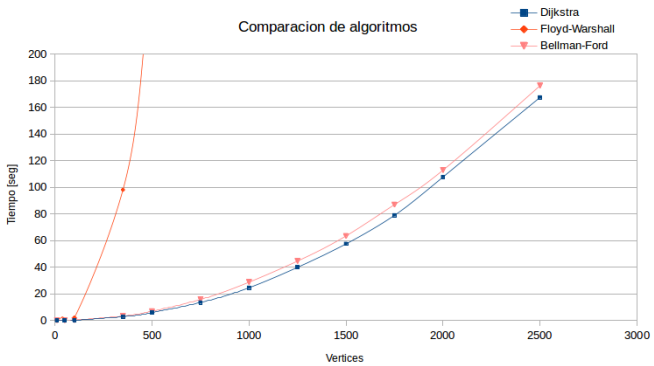
El tiempo asintótico de este algoritmo es, entonces, $O(|V||E|)$, que es más lento que Dijkstra pero permite detectar ciclos negativos.

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 $W(E[u,v]) = -Log(W(E[u,v]))$ pues si nuestro objetivo es hallar un ciclo tal que $w_1 * w_2 * ... * w_k > 1$ entonces buscamos un ciclo tal que $Log(w_1 * w_2 * ... * w_k) > Log(1)$ y por ende, queremos un ciclo tal que $-Log(w_1) - ... - Log(w_k) < 0$. Por lo tanto, si tenemos nuestros nuevos pesos, lo único que necesitamos es buscar ciclos negativos en nuestro grafo actualizado.

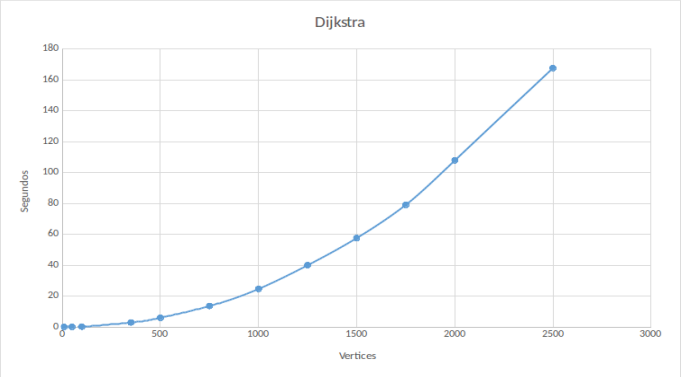
Para esto, como se mencionó en la sección anterior, se pueden utilizar tanto el algoritmo de Bellman-Ford como el de Floyd-Warshall.

2.3 Análisis individual de los algoritmos

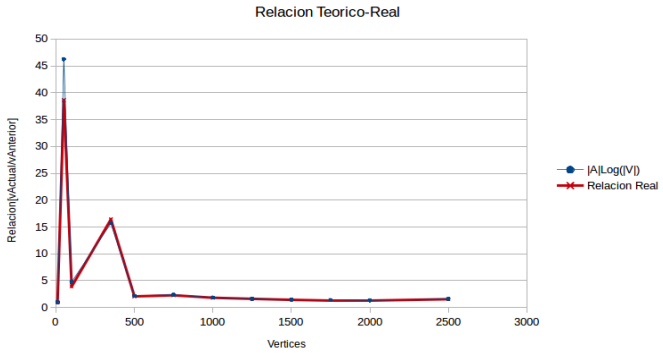
2.3.1 Dijkstra

Como se mencionó previamente, el tiempo asintótico de este algoritmo es $O(|E|Log(|V|))$ 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:



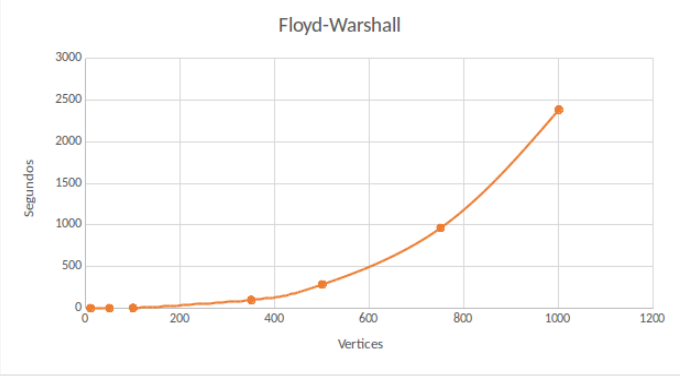
Y aquí se adjunta un gráfico en el que se compara, para los valores de $|V|$ de cada ejecución, la relación de la función $f(V,E) = |E| * Log(|V|)$ para el valor actual y el anterior, y la relación de tiempos de ejecución reales:



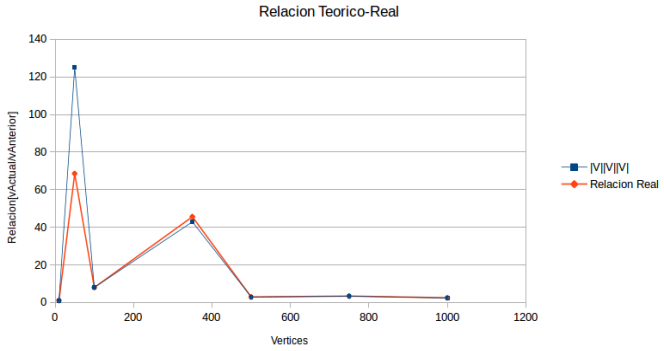
Este gráfico permite ver lo antes dicho: El tiempo de ejecución en la práctica es muy similar al teórico.

2.3.2 Floyd Warshall

Como se mencióno 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:



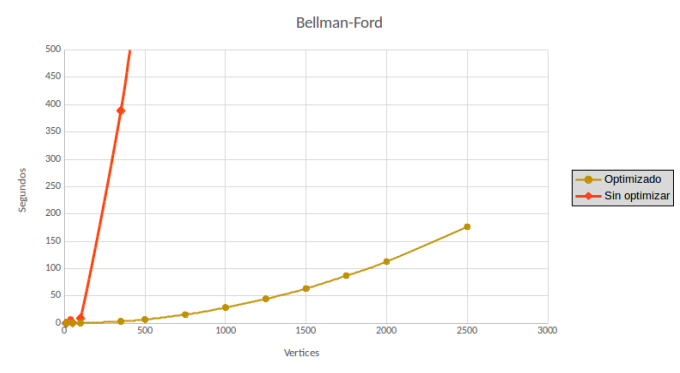
Este gráfico permite ver lo antes dicho: El tiempo de ejecución en la práctica es muy similar al teórico.

2.3.3 Bellman Ford

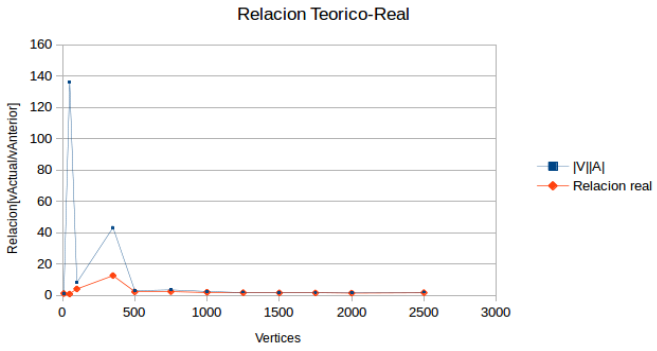
Como antes se ha dicho, el tiempo asintótico de este algoritmo es $O(|V||E|)$, esto en el caso tratado en este trabajo se traduce a $O(|V|^3 - |V|^2)$. De todos modos, existe una optimización que, si bien no baja el tiempo asintótico, mejora mucho el tiempo promedio en la práctica.

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 optimo 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

Para correr el programa, primero se deben crear los archivos de grafos (no fueron incluídos por el peso). Esto se hace de la siguiente forma:

En la carpeta `CaminoMinimo` abrir la consola y ejecutar `python generadorGrafos.py`.

Luego se puede entrar en cada carpeta y ejecutar los programas de la siguiente forma:

En la carpeta `Dijkstra` abrir la consola y ejecutra `python main.py`

En la carpeta `BellmanFord` abrir la consola y ejecutra `python main.py`

En la carpeta `FloydWarshall` abrir la consola y ejecutra `python main.py`