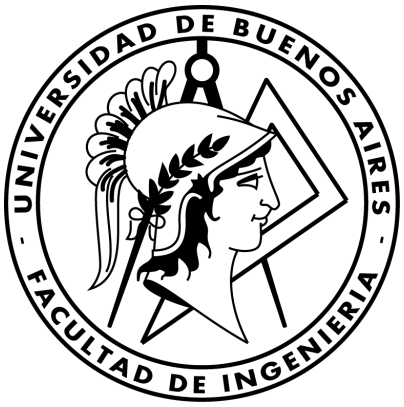

Trabajo Práctico N°2

75.29 - Teoría de Algoritmos I

Facultad de Ingeniería de la Universidad de Buenos Aires

1er. Cuatrimestre 2017

Federico Brasburg, *Padrón Nro. 96.653*
federico.brasburg.@gmail.com

Pablo Rodrigo Ciruzzi, *Padrón Nro. 95.748*
p.ciruzzi@hotmail.com

Andrés Otero, *Padrón Nro. 96.604*
oteroandres95@gmail.com

5 de junio de 2017

Índice

1. Clases de Complejidad

1.1. Punto 1

Es un problema que tiene un algoritmo que lo resuelve de manera óptima en tiempo polinomial. Dicho algoritmo es del tipo greedy y es el siguiente:

```
def actividades_compatibles(actividades):
    ordenar actividades por tiempo de finalizacion en orden creciente
    inicializar S con la primer actividad (la que tiene el fin mas chico)
    para cada actividad A en la lista ordenada de actividades:
        si A.principio ≥ S[ultimaActividadAgregada].fin entonces:
            agregar A a S
    devolver S
```

El algoritmo es $O(n.log(n))$ en tiempo debido al ordenar. Este algoritmo devuelve cuáles actividades hacen máxima la cantidad de actividades que se realizan; basta con contarlas y ver si es mayor o igual a k para dar respuesta al problema original.

1.2. Punto 2

Este es un problema NP-Completo. Para demostrar esto, primero mostramos que está en NP. Un problema está en NP si existe un algoritmo de orden polinomial que, a través de una instancia y un certificado, responda si el certificado da respuesta a la instancia del problema. En este problema siendo la instancia del problema todas las actividades posibles (y sus tiempos) y el certificado el conjunto de actividades compatibles, el algoritmo es trivial, ya que basta con ver que las actividades del certificado sean compatibles y que la cantidad de actividades sea mayor o igual a k.

Ahora hay que encontrar un algoritmo X tal que $X \leq Problema2$ (siendo $\leq$ un operador polinomial, y Problema2, el problema de este ejercicio). Esto es, encontrar X de manera de poder reducirlo polinomialmente a Problema2 para poder decir "X no es más difícil que Problema2" y si X es de los llamados "difíciles" entonces Problema2 también lo será.

El elegido como problema "X" es Set Packing, que es un problema NP-Completo clásico, en el cual existe un universo U, un conjunto de subconjuntos de U que llamaremos S y en el cual, dado un k, hay que ver si existe un subconjunto S de subconjuntos de U de tamaño k o mayor. La transformación polinómica que proponemos para llevar ese problema al problema que queremos demostrar es el siguiente:

- k de Set Packing es igual al de las actividades.
- Transformamos los elementos de U en S a números mediante una función de hashing en un valor número entero.
- A cada subconjunto contenido en S lo pasamos a llamar tareas, resultando $T_1, T_2, ..., T_n$ tareas.
- Para elemento en cada T_i, creamos un intervalo en ese T_i, siendo el *start* el elemento y el *end* el elemento más uno, quedando así cada tarea con intervalos de duración 1.
- Le pasamos estas tareas y el k a nuestro problema y la respuesta que dé va a ser la misma que la de Set Packing.

Por lo tanto, Problema2 es NP-Completo.

1.3. Punto 3

Utilizando el método especificado en 1.2, pasamos a demostrar que el problema del camino Hamiltoniano es un problema NP-Completo:

• Demostramos que está en NP

Esto es trivial en el problema: dado un certificado que sería una secuencia de vértices del grafo simplemente hay que comprobar que existe una arista que va desde un determinado vértice al siguiente en la secuencia. Tomando en cuenta que la secuencia cuenta con todos los vértices, esta comprobación es de tiempo lineal. Expresado de manera formal sería: *Dada una secuencia ($V_{Inicial}$, ..., V_{Final}) de tamaño*

$|V|, \exists\ (V_i,\ V_{i+1}) \in E\ \forall\ i{=}1..N \wedge \exists\ (V_{Final}, V_{Inicial}) \in E$

• Lo reducimos a un problema NP-Completo:
Vamos a reducir el problema a 3-SAT usando un **"gadget"**.
`3-SAT` $\leq_p$ `Hamilton`

El problema 3-SAT utiliza cláusulas booleanas de 3 variables que pueden ser verdad o no. Entonces, vamos a resolver una instancia general de 3-SAT con una instancia particular del problema del camino Hamiltoniano.

Dada una instancia de 3-SAT con variables $x_1, ..., x_n$ y cláusulas $c_1, ..., c_k$, podemos generar un digrafo que tenga n caminos. Cada camino tiene $3k + 3$ vértices y cada camino va en ambos sentidos. Además conectamos el primer y el último nodo de cada camino, con el primero y el último del próximo camino. Por otro lado, agregamos un nodo al principio que conecte con el primer y último nodo del primer camino, así como hacemos lo mismo con un nodo final que se conecte con el primer y último nodo del último *path*. Para terminar, se conectan estos últimos 2 nodos agregados entre sí. Finalmente queda un grafo como se muestra en la figura 1.

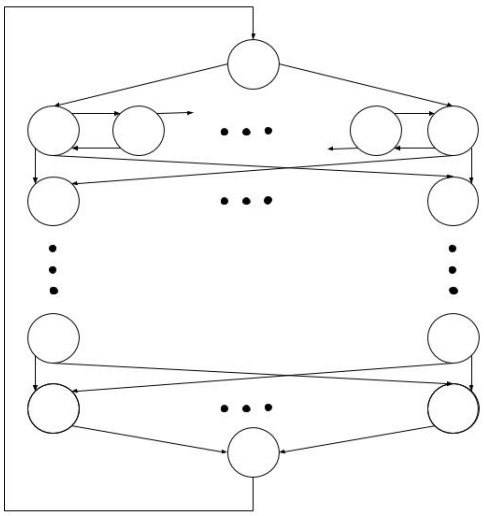

Figura 1: Grafo resultante de la reducción de 3-SAT.

De esta manera lo que estamos haciendo es replicando en un grafo que tiene 2^n posibles ciclos hamiltonianos, las 2^n posibles combinaciones de las n variables booleanas. Lo podemos pensar como si el ciclo se recorre de izquierda a derecha, su variable análoga vale 1 y si es al revés vale 0.

Teniendo esto en mente, podemos terminar de armar nuestro grafo utilizando un nodo más (k en total) para simular una cláusula. Éstas van a estar conectadas a aristas en el mismo sentido que debe recorrer los caminos para que su variables den el valor asociado. Para clarificar, teniendo una cláusula h que tiene la forma $(x_1 \vee x_2 \vee x_3)$, se agrega un vértice V_h que va a estar conectado a $V_{1,L}, V_{2,L+1}, V_{3,L}$ y aristas desde $V_{1,L+1}, V_{2,L}, V_{3,L+1}$. Para clarificar la notación $V_{1,L}$ es el vértice L de izquierda a derecha del primer camino. Ahora si P_1 (El primer camino) se recorre de izquierda a derecha puede visitar a V_h. Para hacerlo más general cada cláusula C_j de variables x, y, z estará conectado con $V_{x,3j}, V_{y,3j}, V_{z,3j}, V_{x,3j+1}, V_{y,3j+1}, V_{z,3j+1}$. De esta manera sabemos que una cláusula puede ser satisfecha si y sólo si una de sus variables toma el valor necesario, que es análogo a decir que es visitado si el camino asociado a la variable correspondiente es recorrido de manera correcta respecto al valor de esa variable. Finalmente, queda entonces un grafo similar al de la figura 2.

Para finalizar podemos decir que el problema es satisfacible si y sólo si existe un ciclo hamiltoniano que lo cumpla, ya que las cláusulas serán visitadas (y solo pueden ser visitadas si se cumplen) si existe un camino hamiltoniano.

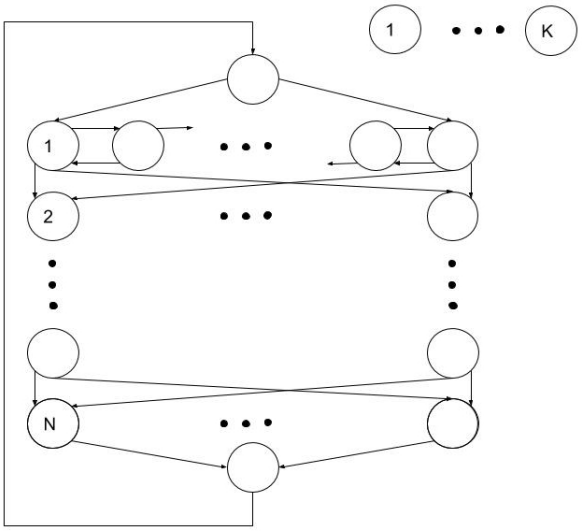

Figura 2: Grafo final de la reducción de 3-SAT.

1.4. Punto 4

Este problema es P y su algoritmo de resolución es bastante sencillo. Utilizando el algoritmo de orden topológico encontramos el orden lineal que conserva la unión entre vértices. Una vez que terminamos este ordenamiento, simplemente si todos los nodos se conectan con su siguiente entonces existe un camino hamiltoniano. La idea surgió de un ejercicio parecido hecho anteriormente.

```
DAG_Hamiltoniano (grafo G):
    lista = Ordenar_Topologicamente(G)
    for i in |lista|−1:
        if (lista[i],lista[i+1]) ∉ E[G]:
            return False
    return True
Ordenar_Topologicamente (grafo G):
    lista=[]
    for each vertice u ∈ V[G]:
        estado[u] = NO_VISITADO
        padre[u] = NULL
    tiempo = 0
    for each vertice u ∈ V[G]:
        if estado[u] = NO_VISITADO:
            Visitar(u)
    return lista
Visitar(nodo u)
    estado[u] = VISITADO
    tiempo = tiempo + 1
    distancia[u] = tiempo
    for each v ∈ Adyacencia[u]:
        if estado[v] = NO_VISITADO:
            padre[v] = u
            Visitar(v)
    estado[u] = TERMINADO
    tiempo = tiempo + 1
    finalizacion[u] = tiempo
    list.add(u)
```

1.5. Punto 5

El problema de búsqueda de ciclos negativos es posible de resolver en forma polinómica. Generalmente, algunos algoritmos de grafos cuya intención primera no es encontrar ciclos negativos, tienen la funcionalidad de encontrarlos como un "beneficio" adicional. Un ejemplo conocido es el algoritmo de Bellman-Ford, cuyo objetivo es encontrar, dado un origen, caminos mínimos a todos los nodos de un grafo, aún para grafos con aristas con peso negativo. A continuación mostramos dicho algoritmo, y cómo hace para detectar los ciclos negativos en tiempo polinomial:

```
Bellman−Ford(G, s):
∀ v ∈ G.vertices:
    v.dist = ∞
s.dist = 0
∀ i = 1..|G.vertices| − 1:
    ∀ a=(u,v) ∈ G.aristas:
        if v.dist > u.dist + d(u,v)
            v.dist = u.dist + d(u,v)
```

Una vez finalizado este simple algoritmo (Que tiene tiempo $O(|V|.|A|)$, es decir, polinomial), con la información obtenida, se puede hacer la detección de ciclos negativos, donde se devolvería verdadero o falso indicando si hay o no un ciclo negativo:

```
∀ a=(u,v) ∈ G.aristas:
    if v.dist > u.dist + d(u,v)
        return true
return false
```

1.6. Punto 6

Este problema se puede comprobar que es un problema NP-Completo, así como se describió en 1.2. Para ello, basta con comprobar las 2 condiciones necesarias para dicha condición:

• El problema es NP

Esto es relativamente simple. Básicamente, recibiendo una instancia (un grafo) y un certificado (Un conjunto de vértices y aristas indicando el ciclo), habría que comprobar que es un ciclo (Lo cual se puede comprobar en $O(|V| + |A|)$) y que la suma de los pesos de las aristas es 0.

• El problema es NP-Hard

Para esto, se utiliza algún problema NP-Completo de tal manera que, reduciendo ese problema al nuestro en tiempo polinomial, se demuestra que nuestro problema es al menos tan difícil que dicho problema NP-Completo. En este caso, el problema NPC a utilizar va a ser Subset Sum, ya que por su naturaleza (Ver si dentro de un conjunto de números hay un subconjunto cuya suma es igual a 0), parecería ser una buena elección para nuestro problema. Entonces, hay que encontrar una transformación polinómica para demostrar que:

`Subset Sum` $\leq_p$ `ZWC (Zero Weight Cycle)`

La idea de la reducción es, dado un conjunto $S = \{x_1, x_2, ..., x_n\}$ de números, generar un grafo completo de n vértices, donde a cada vértice i, desde cada uno de los otros, entre una arista con peso x_i. Así se puede asegurar que, al ser completo, siempre que exista un ciclo de peso 0, existirá un subconjunto $T \subseteq S$, cuya suma sea igual a 0. El único caso particular a tener en cuenta ya que traería problemas, es si dentro de S se encuentra el número 0. En ese caso, la idea sería detectar antes de pasarlo al grafo y devolver directamente que existe el subconjunto con suma igual a 0 (Ya que sino no se detectaría el grafo con camino de peso igual a 0). Por último, cabe aclarar que dicha transformación es polinómica, ya que para generar el grafo completo dado el conjunto S, se requiere la creación de n vértices y $n.(n-1)$ aristas. Es decir, la transformación es $O(n^2)$.

Por ejemplo, dado el conjunto $S = \{5, 4, -2, -3\}$, el grafo y su respectiva solución sería el de la figura 3, donde se puede observar que, como existe un subconjunto $\{5, -2, -3\}$ cuya suma es 0, existe al menos un ciclo en el grafo correspondiente.

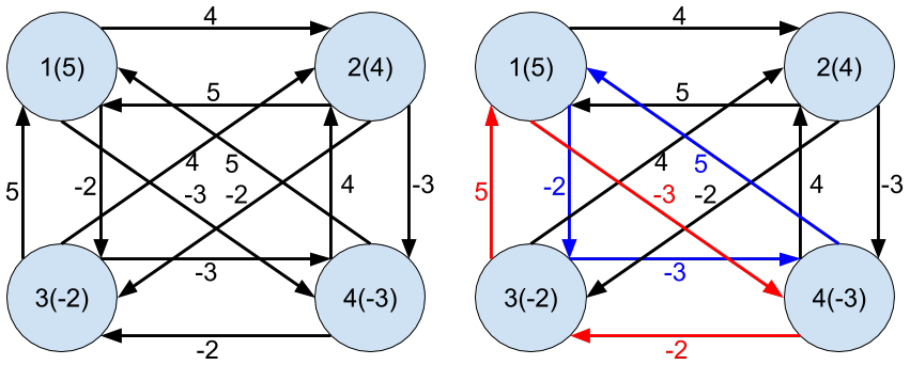

Figura 3: Problema 6.

2. Algoritmos de camino mínimo

2.1. Dijkstra

Dijkstra funciona de manera Greedy, donde dado un origen busca en cada iteración el camino más corto a todos los nodos. Optimizado utilizando una estructura de tipo *heap* para encontrar la menor distancia en cada paso, el algoritmo es $O(|A| + |V|.log(|V|))$.

2.2. Bellman-Ford

El algoritmo de Bellman-Ford es un algoritmo de programación dinámica. Funciona análogamente al de Dijkstra pero en vez de buscar en cada iteración la arista de peso mínimo, relaja (es decir, recalcula la distancia para todos los nodos) en cada iteración todas las aristas, realizándolo para cada nodo. El tiempo de ejecución del algoritmo es $O(|A|.|V|)$.

2.3. Floyd-Warshall

El objetivo de Floyd-Warshall es calcular todos los caminos mínimos entre todos los pares del grafo. Intuitivamente, uno pensaría hacer $|V|$ veces Bellman-Ford, lo cual resultaría en un algoritmo $O(|A|.|V|.|V|) \approx O(|V|^4)$. De todas formas, este algoritmo corre en $O(|V|^3)$. La idea es, utilizando una matriz de distancias, ir calculando el camino mínimo entre todos los pares, agregando en cada iteración la posibilidad de pasar por un vértice más. Este algoritmo se corresponde con la técnica de programación dinámica.

2.4. Comparación

Como principales diferencias, podemos decir que Dijkstra no acepta aristas con pesos negativos, mientras que Bellman-Ford y Floyd-Warshall sí lo hacen. Fuera de eso, el resultado de Dijkstra y Bellman-Ford debería ser el mismo, ya que resuelvan el mismo problema. Por otro lado, los tiempos de ejecución son distintos, siendo Dijkstra el de menor tiempo, y siendo Bellman-Ford y Floyd-Warshall del mismo de orden de magnitud. Por último, como ya se mencionó, Dijkstra y Bellman-Ford dan el resultado para un vértice de origen, mientras que Floyd-Warshall da los caminos mínimos para todo el grafo.

2.5. Detección de ciclos negativos

Tal como se mencionó en el punto 1.5 de la primera parte del TP, se podría utilizar el algoritmo de Bellman-Ford para encontrar ciclos negativos.

Por otro lado, Dijkstra no es capaz, ya que ni siquiera admite aristas con pesos negativos, por lo que, en una instancia válida, no habría ciclos y aún en una instancia inválida (Con aristas con pesos negativos), un ciclo negativo sería imposible de detectar.

En cuanto a Floyd-Warshall, también es capaz de identificar ciclos negativos. En este caso, el resultado devuelto por el algoritmo es incorrecto, y uno puede darse cuenta de dicha situación viendo que en la

diagonal principal de la matriz hay valores negativos (Lo cual es ilógico si no hay ciclos negativos, ya que la menor distancia de un nodo a sí mismo debería ser siempre 0).

2.6. Resultados

Habiendo generado 10 archivos con grafos completos con pesos positivos aleatorios entre 0 y 2, logramos correr los siguientes algoritmos en los tiempos indicados:

Tamaño	Dijkstra	Bellman-Ford	Floyd-Warshall	Dijkstra Unitario	Bellman-Ford Unitario
10	0,000026	0,003331	0,000353	3,10E-06	0,00045
20	0,000032	0,068222	0,002055	5,01E-06	0,00355
50	0,000179	2,468362	0,033961	6,91E-06	0,06193
100	0,000506	42,353214	0,239872	8,82E-06	0,42655
250	0,002726		3,854090	2,00E-05	6,46512
500	0,012659			2,10E-05	55,54547
1000	0,043105			4,10E-05	
1500	0,109387			6,10E-05	
2000	0,178826			8,39E-05	
2500	0,285274			0,00010	

Primero, cabe destacar que las primeras 3 columnas se corresponden con el problema de caminos mínimos para todos los nodos con todos. En los casos de Dijkstra y Bellman-Ford, lo que se hizo es correr n veces dicho algoritmo. Las últimas 2 columnas se corresponden con una única corrida de Dijkstra y Bellman-Ford, con un origen cualquiera.

Dicho esto, podemos ver que, entre 10 y 100 por ejemplo, Floyd-Warshall aumenta su tiempo con un orden de magnitud $O(|V|^3)$, así como Bellman-Ford en $O(|V|^3)$ en el caso "unitario" y en $O(|V|^4)$ en el otro caso. A su vez, Dijkstra que debería correr en tiempo $O(|V|^2 + |V|.log(|V|))$, vemos que es completamente inferior a lo que se espera teóricamente. La única explicación que encontramos es que, por la particularidad del grafo entero, no nos encontremos en el peor caso y por ello se comporte de una mejor manera, ya que revisando el algoritmo, éste es correcto. Cabe recordar que en este caso, al ser un grafo completo $|A| \cong |V|^2$. Cabe aclarar que por el tipo de crecimiento de cada algoritmo es que se pudo correr más o menos grafos para cada uno.

Dicho esto, y viendo los gráficos 4 y 5 (En donde los de la derecha son con el eje de las ordenadas en escala logarítmica), podemos ver que para resolver el problema, siempre lo más rápido es Dijkstra (Teniendo en cuenta que son todos pesos positivos). Por otro lado, tal como se afirmó en el párrafo anterior, se puede ver en el segundo gráfico que Floyd-Warshall y Bellman-Ford "unitario" crecen de la misma manera, mientras que Dijkstra es extraña y prácticamente lineal (En el caso unitario), tal como se puede observar en el gráfico 6.

Como una conclusión final, podemos decir que si las condiciones son las necesarias, lo mejor a utilizar es Dijkstra, mientras que si existen aristas con peso negativo, lo mejor a utilizar es Floyd-Warshall (Al menos en el caso de grafos completos), ya que performa mejor que Bellman-Ford, y se obtiene más información sobre el grafo.

2.7. Cómo correrlo

Es importante mencionar que se deben crear las carpetas `in` y `out` en la raíz del proyecto, ya que será donde se guarden los archivos de los grafos generados y los resultados de las corridas respectivamente.

Una vez hecho esto, para correr la creación de los grafos completos y la posterior utilización de los algoritmos junto con la generación de los resultados, desde la carpeta `src`, correr el comando `python tiempos.py`.

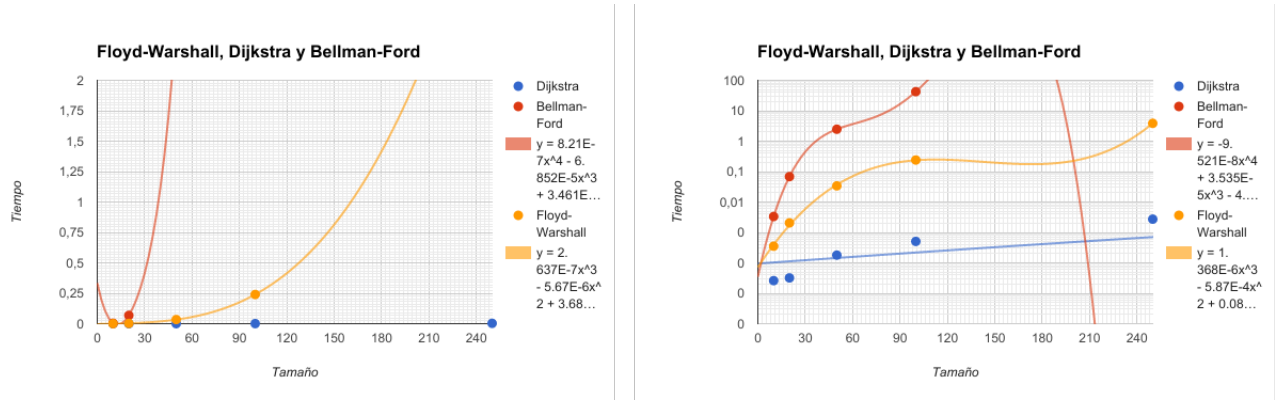

Figura 4: Floyd Warshall, Bellman-Ford, Dijkstra.

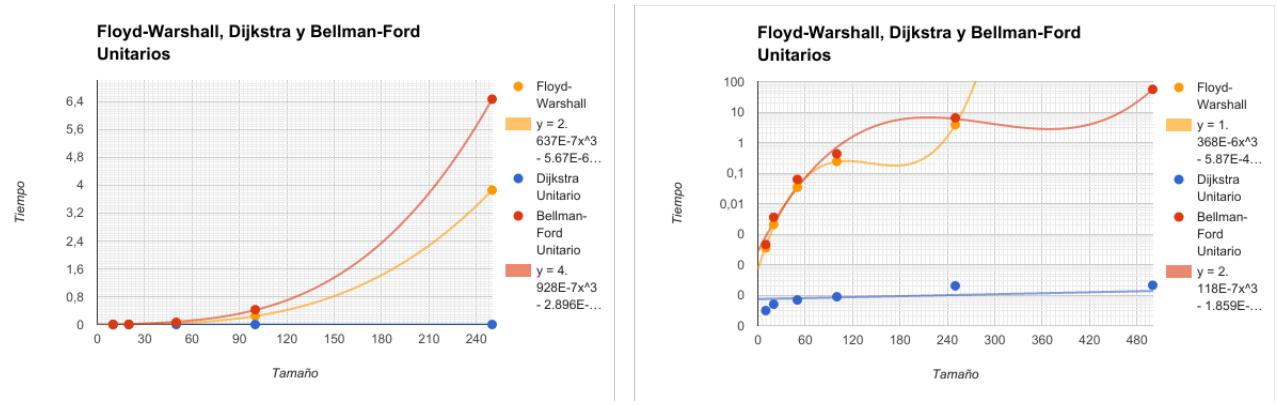

Figura 5: Floyd Warshall, Bellman-Ford Unitario, Dijkstra Unitario.

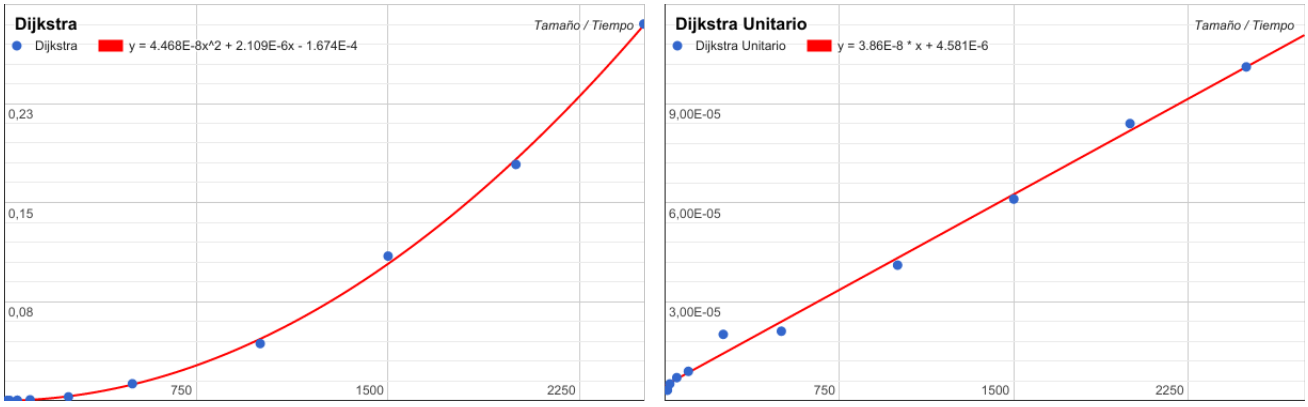

Figura 6: Dijkstra (Unitario y no).

3. Código

bellman_ford.py

```
INFINITO = float("inf")


class BellmanFord(object):
    def __init__(self, grafo):
```

```
        self . grafo = grafo

    def __inicializar__iterador (self ):
        """Devuelve un diccionario con la distancia en infinito y otro con los padres seteado
            en none para cada vertice del grafo """
        distancia = {}
        padre = {}
        for actual in self . grafo . devolver__vertices() :
            distancia [ actual] = INFINITO
            padre[actual] = None
        return padre, distancia

    def bellmanFord(self, ID):
        """Algoritmo de Floyd Warshall"""
        vertices = self . grafo . devolver__vertices()
        aristas = self . grafo . devolver__aristas()
        padre, distancia = self . __inicializar__iterador ()
        distancia [ ID] = 0 # La distancia a si mismo es 0
        for i in vertices :
            for u in aristas :
                for v in aristas [ u]:
                    w = float(aristas [ u][ v]. peso)
                    if distancia [ u] + w < distancia[v]: # Relajamiento de la arista, si el
                        nuevo camino es menos costoso que el anterior este se convierte en el
                        nuevo camino
                        distancia [ v] = distancia[ u] + w
                        padre[v] = u
        return padre, distancia

    def resolver__camino__minimo(self, ID):
        return self . bellmanFord(ID)
```

creador_grafos.py

```
import random


def crearDigrafoCompleto(n, nombre):
    cantVertices = n * (n - 1)
    arch = open(nombre, 'w')
    arch. write(str (n) + "\n")
    arch. write(str (cantVertices ) + "\n")
    for i in range(n):
        for j in range(n):
            if i != j:
                arch. write(str (i ) + "␣" + str(j) + "␣" + str(random.random() * 2) + "␣" + "\n
                    ") # Crea una arista del vertice i al vertice j con un valor al azar entre 0
                    y 2
    arch. close ()
```

dijkstra.py

```
import heapq

INFINITO = float("inf")
CERO = 0
```

```
class Dijkstra(object):
    def __init__(self, grafo):
        self.grafo = grafo

    def _inicializar_iterador(self):
        """Inicializa variables del iterador"""
        visitado = {}
        distancia = {}
        padre = {}
        for actual in self.grafo.devolver_vertices():
            visitado[actual] = False
            distancia[actual] = INFINITO
            padre[actual] = None
        return visitado, padre, distancia

    def dijkstra(self, ID):
        """Realiza el algoritmo de Dijkstra, devuelve las distancias y los padres encontrados
            desde el ID"""
        vertices = self.grafo.devolver_vertices()
        if ID not in vertices:
            return
        heap = []
        visitado, padre, distancia = self._inicializar_iterador()
        distancia[ID] = CERO # La distancia a si mismo es 0
        nodo = Nodo(ID, distancia[ID], padre[ID])
        heapq.heappush(heap, nodo)
        while heap:
            nodo = heapq.heappop(heap)
            ID = nodo.ID
            if not visitado[ID]:
                visitado[ID] = True
                padre[ID] = nodo.padre
                distancia[ID] = nodo.distancia
                for ID_ady in self.grafo.adyacentes(ID):
                    nueva_distancia = distancia[ID] + self.grafo.peso_arista(ID, ID_ady)
                    if nueva_distancia < distancia[ID_ady]: # Relajamiento
                        nodo_nuevo = Nodo(ID_ady, nueva_distancia, ID)
                        heapq.heappush(heap, nodo_nuevo)
        return distancia, padre

    def resolver_camino_minimo(self, ID):
        return self.dijkstra(ID)

class Nodo(object):
    """Clase auxiliar para representar un vertice con toda la informacion perteneciente al
        algoritmo"""
    def __init__(self, ID, distancia, padre=None):
        self.ID = ID
        self.distancia = distancia
        self.padre = padre

    def __cmp__(self, otro):
        if self.distancia == otro.distancia:
            return 0
```

```
        if self.distancia > otro.distancia:
            return 1
        return -1

    def inverse_cmp(self, otro):
        return -self.__cmp__(otro)
```

floyd_warshall.py

```
INFINITO = float("inf")


class FloydWarshall(object):
    def __init__(self, grafo):
        self.grafo = grafo

    def floydWarshall(self):
        """Algoritmo de Floyd Warshall"""
        n = self.grafo.devolver_cant_vertices()
        self.camino = [[INFINITO for i in range(n)] for j in range(n)]  # Inicializo todas las
            distancias de la matriz en infinito
        for i in range(n):
            self.camino[i][i] = 0  # Inicializo las distancias de un nodo a si mismo en 0
        aristas = self.grafo.devolver_aristas_list()
        for a in aristas:
            self.camino[a.id1][a.id2] = float(a.peso)

        vertices = self.grafo.devolver_vertices()
        for k in vertices:
            for j in vertices:
                for i in vertices:
                    nuevo_camino = self.camino[i][k] + self.camino[k][j]
                    if self.camino[i][j] > nuevo_camino:
                        self.camino[i][j] = nuevo_camino
        return self.camino

    def resolver_camino_minimo(self):
        return self.floydWarshall()
```

grafo.py

```
PRIMERO = 0
SEGUNDO = 1
TERCERO = 2


class Arista(object):
    def __init__(self, id1, id2, peso):
        self.id1 = id1
        self.id2 = id2
        self.peso = peso

    def peso(self):
        return self.peso

    def __str__(self):
```

```
        return str(self.id1) + "␣a␣" + str(self.id2) + ",␣peso␣" + str(self.peso)


class Grafo(object):
    def ___init___(self):
        """Crea un Grafo dirigido (o no) con aristas pesadas (o no)"""
        self.aristas = {}
        self.vertices = []
        self.aristas_list = []

    def devolver_aristas(self):
        """Devuelve las aristas del grafo"""
        return self.aristas

    def devolver_aristas_list(self):
        return self.aristas_list

    def devolver_vertices(self):
        return self.vertices

    def devolver_cant_vertices(self):
        """Devuelve los nodos del grafo"""
        return len(self.vertices)

    def agregar_vertice(self, id):
        """Agrega un vertice que se identifica con un nombre y un ID"""
        self.vertices.append(id)
        self.aristas[id] = {}

    def agregar_arista_no_dirigida(self, id1, id2, peso=0):
        """Agrego una arista no dirigida entre los nodos con id1 y id2"""
        self.agregar_arista_dirigida(id1, id2, peso)
        self.agregar_arista_dirigida(id2, id1, peso)

    def agregar_arista_dirigida(self, id1, id2, peso=0):
        """Agrego una arista dirigida entre los nodos con id1 y id2"""
        arista = Arista(id1, id2, peso)
        self.aristas[id1][id2] = arista
        self.aristas_list.append(arista)

    def son_vecinos(self, id1, id2):
        """Devuelve si id1 y id2 son vecinos"""
        try:
            if self.aristas[id1][id2]:
                return True
            return False
        except:
            return False

    def peso_arista(self, id1, id2):
        """Devuelve el peso de la arista entre id1 e id2"""
        if self.son_vecinos(id1, id2):
            return self.aristas[id1][id2].peso
        raise ValueError

    def adyacentes(self, id):
```

```
        """Pide un id de un nodo existe y devuelve una lista de los id de sus adyacentes"""
        adyacentes = []
        for arista in self.aristas[id]:
            adyacentes.append(arista)
        return adyacentes

    def _leer(self, nombre, dirigido=False):
        """Lee un grafo (dirigido o no) de un archivo con nombre"""
        try:
            mi_arch = open(nombre)
            cant_nodos = int(mi_arch.readline())
            for i in range(0, cant_nodos):
                self.agregar_vertice(i)
            cant_aristas = int(mi_arch.readline())
            for i in range(0, cant_aristas):
                linea = mi_arch.readline()
                numeros = linea.split(" ")
                peso = 0
                if len(numeros) > 2:
                    peso = numeros[TERCERO].rstrip('\n')
                else:
                    numeros[SEGUNDO] = numeros[SEGUNDO].rstrip('\n')
                if dirigido:
                    self.agregar_arista_dirigida(int(numeros[PRIMERO]), int(numeros[
                        SEGUNDO]), peso)
                else:
                    self.agregar_arista_no_dirigida(int(numeros[PRIMERO]), int(numeros[
                        SEGUNDO]), peso)
            mi_arch.close()
            return True
        except:
            return False

    def leer_dirigido(self, nombre):
        self._leer(nombre, True)

    def leer_no_dirigido(self, nombre):
        self._leer(nombre, False)
```

parser.py

```
from grafo import Grafo

CERO = 0
UNO = 1


class Parser(object):
    def leer_grafo_no_dirigido(self, nombre):
        """Lee un archivo de un grafo no dirigido sin peso"""
        try:
            grafo = Grafo()
            grafo.leer_no_dirigido(nombre)
            return grafo
        except:
            print "Ocurrio un error leyendo el archivo de grafo no dirigido " + nombre
```

```
            return False

    def leer_grafo_dirigido(self , nombre):
        """Lee un archivo de un grafo dirigido sin peso"""
        try:
            grafo = Grafo()
            grafo.leer_dirigido(nombre)
            return grafo
        except:
            print "Ocurrio un error leyendo el archivo de grafo dirigido " + nombre
            return False
```

tiempos.py

```
from creador_grafos import crearDigrafoCompleto
from bellman_ford import BellmanFord
from floyd_warshall import FloydWarshall
from dijkstra import Dijkstra
from parser import Parser
import time
import csv
from os.path import isfile

FLOYDWARSHALL = 0
DIJKSTRA = 1
BELLMANFORD = 2

grafos_a_probar = {}
grafos_a_probar[FLOYDWARSHALL] = [10, 20, 50, 100, 250]
grafos_a_probar[DIJKSTRA] = [10, 20, 50, 100, 250, 500, 1000]
grafos_a_probar[BELLMANFORD] = [10, 20, 50, 100]
grafos_utilizados = [10, 20, 50, 100, 250, 500, 1000]

def devolver_algoritmo(n, grafo):
    if n == FLOYDWARSHALL:
        return FloydWarshall(grafo)
    if n == DIJKSTRA:
        return Dijkstra(grafo)
    if n == BELLMANFORD:
        return BellmanFord(grafo)

def crear_grafos_de_prueba():
    for tamanio_grafo in grafos_utilizados:
        path = "../in/grafoprueba" + str(tamanio_grafo) + ".txt"
        if not isfile (path):
            crearDigrafoCompleto(tamanio_grafo, path)

def realizar_pruebas():
    parser = Parser()
    lista_iteraciones = []

    for tamanio_grafo in grafos_utilizados:
        print 'Leyendo grafo de tamanio ' + str(tamanio_grafo)
        grafo = parser.leer_grafo_dirigido("../ in/grafoprueba" + str(tamanio_grafo) + ".txt")

        # Analizo los algoritmos para todos los caminos minimos del grafo
```

```
        for numero_de_algoritmo in [DIJKSTRA]:
            if tamanio_grafo in grafos_a_probar[numero_de_algoritmo]:
                algoritmo = devolver_algoritmo(numero_de_algoritmo, grafo)
                print "\t" + algoritmo.__class__.__name__
                start = time.time()
                if numero_de_algoritmo == FLOYDWARSHALL:
                    algoritmo.resolver_camino_minimo()
                else:
                    for i in range(0, tamanio_grafo - 1):
                        algoritmo.resolver_camino_minimo(str(i))
                end = time.time()
                lista_iteraciones.append((algoritmo.__class__.__name__, str(tamanio_grafo)
                    , str(end - start)))

        # Analizo Dijkstra y Bellman-Ford en forma "unitaria" (Es decir, para un unico origen)
        for numero_de_algoritmo in [DIJKSTRA]:
            if tamanio_grafo in grafos_a_probar[numero_de_algoritmo]:
                algoritmo = devolver_algoritmo(numero_de_algoritmo, grafo)
                print "\t" + algoritmo.__class__.__name__ + "_unitario"
                start = time.time()
                algoritmo.resolver_camino_minimo(str(0))
                end = time.time()
                lista_iteraciones.append((algoritmo.__class__.__name__+"_unitario", str(
                    tamanio_grafo), str(end - start)))

    # Creo archivo .csv para crear los graficos facilmente
    f = open("../out/corrida_tiempo.csv", 'wt')
    try:
        writer = csv.writer(f)
        writer.writerow(('Algoritmo', 'Tamanio', 'Tiempo'))
        for corrida in lista_iteraciones:
            writer.writerow(corrida)
    finally:
        f.close()

    texto = ""
    for corrida in lista_iteraciones:
        texto += "El algoritmo " + corrida[0] + " para la instancia " + corrida[1] + " tardo "
            + corrida[2] + "\n"
    f = open("../out/corrida_tiempo.txt", 'w')
    f.write(str(texto))
    f.close()

crear_grafos_de_prueba()
realizar_pruebas()
```