DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 2 (Reentrega)

Modelando problemas reales con grafos

Algoritmos y Estructuras de Datos III

Segundo Cuatrimestre de 2018

Integrante	LU	Correo electrónico
Alliani Federico	183/15	fedealliani@gmail.com
Imperiale Luca	436/15	luca.imperiale95@gmail.com
Raposeiras Lucas Damián	034/15	lucas.raposeiras@outlook.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (54 11) 4576-3359

http://www.fcen.uba.ar

Índice

1. Introducción

En el presente trabajo práctico estudiaremos, analizaremos y resolveremos distintas soluciones para dos problemas que pueden ser modelados mediante grafos. Estos son los problemas de *clusterización* y *arbitraje*.

2. Primer problema: Separando la paja del trigo (*clusterización*)

El problema de *clusterización* consiste en agrupar un conjunto de objetos buscando que los miembros de un mismo grupo sean *similares* entre sí y *diferentes* a los de los otros grupos. Dada la complejidad de las posibles dinámicas que dieron origen a los patrones observables, y la de estos mismos, a menudo se busca simplemente que el resultado sea consistente con la percepción humana.

2.1. El problema

> Dado S un conjunto de objetos con ciertas propiedades, hallar una forma de agrupar los elementos de S por alguna noción de similaridad intraconjunto (o disimilaridad interconjunto).

Particularmente en el caso de este trabajo práctico, trataremos de aproximarnos a una resolución del problema de *clusterización* sobre un conjunto de puntos en un plano euclídeo, donde la principal noción de grupo estará directamente ligada a las distancias entre cada punto.

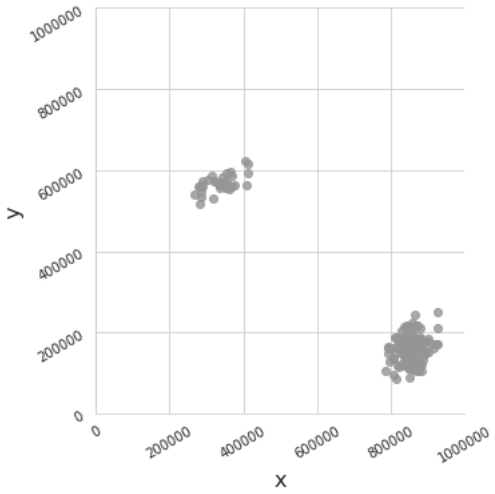

Figura 1: Conjunto de puntos del plano euclídeo de ejemplo

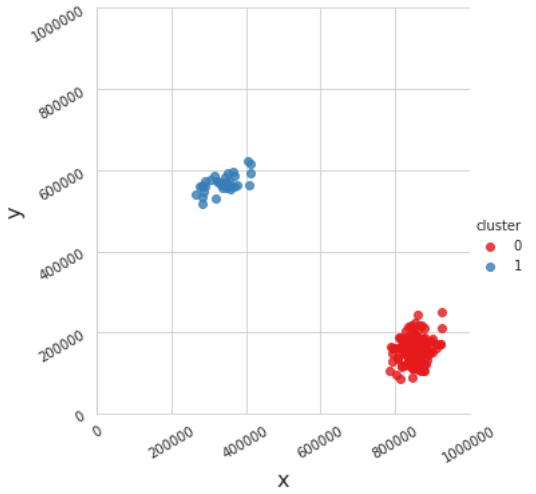

Figura 2: Posible *clusterización* del conjunto de puntos del plano euclídeo de ejemplo mostrado en la Figura 1

Por ejemplo, para el conjunto de ejes ordenados de la Figura 1, podemos pensar que una posible *clusterización* es la mostrada en la Figura 2.

2.2. Resolución

Sea A el conjunto de puntos del plano euclídeo que se quiere *clusterizar*. Comenzaremos definiendo un grafo no dirigido G donde cada nodo representa un elemento de A. Es decir, cada nodo de G es un punto del plano euclídeo que se quiere *clusterizar*. Además, definimos que G es completo, con lo cual todos sus nodos son universales. Por último, definimos el *peso* de una arista (u, v) como la distancia euclídea entre los puntos del plano que son representados por los nodos u y v en G. Vemos que esta representación es correcta, ya que para cada par de puntos (x_1, y_1), (x_2, y_2) del plano, existen dos nodos $u, v \in G$ tales que u representa a (x_1, y_1) y v representa a (x_2, y_2), y $peso\big((u, v)\big) = \|(x_1, y_1) - (x_2, y_2)\|$.

Si bien este modelo de grafo ignora la **posición absoluta** de cada punto (x, y) en el plano, no nos hará falta para determinar el agrupamiento de los mismos.

Ya habiendo modelado el grafo G, el problema a resolver se convierte, en este caso, en encontrar una partición de los nodos de G en base al peso de sus aristas. Más específicamente, para la resolución del problema de *clusterización*, aplicaremos los conceptos presentados en la sección VII de [1] y utilizaremos las dos nociones de *eje inconsistente* exhibidas en las secciones VII y XVII de dicho *paper*. Utilizaremos los algoritmos de Prim, Kruskal y Kruskal con *Path Compression* (compresión de camino) para encontrar un Árbol Generador Mínimo de G. De esta forma podemos detectar y eliminar estos ejes, separando así el AGM en distintas componentes conexas, cuyos nodos representarán distintos *clusters* en la solución del problema original.

Según la noción de *eje inconsistente* presentada en [1], un eje e se denomina inconsistente si su peso es *significativamente* mayor que el promedio de los pesos de los ejes cercanos a ambos lados de e. Se utilizarán dos métodos para medir y comparar la relación entre el peso de un eje e con el de sus vecinos cercanos. El primero de ellos es ver cuántos desvíos estándar separan al peso de e respecto del promedio de los pesos de los ejes cercanos a e. El segundo método consiste en calcular el *ratio* entre el peso de e y el promedio de los pesos de sus ejes cercanos.

En este caso queda a nuestro criterio determinar los parámetros cuantificativos que definen la inconsistencia de un eje e para ambos métodos sugeridos, tanto para el exceso de desvío estándar necesario en el primer método como para el ratio a superar en el segundo.

Proponemos entonces el siguiente algoritmo para detectar un *eje inconsistente* en un Árbol Generador Mínimo según las definiciones exhibidas en los párrafos anteriores.

---

function EJEINCONSISTENTE($G =< V, E >, e = (u, v), distancia, método,$
$excesoDesvíoEstándar, excesoRatio$)
  ejesCercanos ← OBTENEREJESCERCANOS($G, e, distancia$) ▷ $O(n^2)$

  if *método* $= 1$ **then** ▷ $O(1)$
    // Comprobamos si el peso de e excede la cantidad de desvíos estándar especificada por parámetro
    desvíoEstándarEjesCercanos ← $\sqrt{\text{CALCULARVARIANZA}(\textit{ejesCercanos})}$ ▷ $O(m) = O(n)$
    if peso(e) $>$ *excesoDesvíoEstándar* $\times$ *desvíoEstándarEjesCercanos* **then** ▷ $O(1)$
      return `true` ▷ $O(1)$
    end if
  else
    // Comprobamos si el *ratio* entre el peso de e y el promedio de los pesos de sus ejes cercanos excede el *ratio* especificado por parámetro
    promedioEjesCercanos ← $\frac{\text{SUMATORIA}(\textit{ejesCercanos})}{|\textit{ejesCercanos}|}$ ▷ $O(m) = O(n)$
    ratio ← $\frac{\text{peso}(e)}{\textit{promedioEjesCercanos}}$ ▷ $O(1)$
    if *ratio* $>$ *excesoRatio* **then** ▷ $O(1)$
      return `true` ▷ $O(1)$
    end if
  end if

  return `false` ▷ $O(1)$
end function

La complejidad de la función EJEINCONSISTENTE es $O(n^2)$, con n la cantidad de nodos del grafo G.

La función OBTENEREJESCERCANOS obtiene todos los ejes que están a distancia menor o igual a d desde el eje e indicado por parámetro. Específicamente, lo que se hace es recorrer la lista de vecinos de los dos nodos que forman la arista e y, para cada uno de ellos, se lo agrega a un vector y se llama recursivamente a OBTENEREJESCERCANOS con dicho nodo y decrementando el valor de d. La recursión llegará a su fin cuando el valor de d sea exactamente 0. La complejidad temporal de OBTENEREJESCERCANOS sin incluir la llamada recursiva es de $O(n)$, ya que se itera sobre la lista de vecinos de un nodo, y esta lista tiene complejidad espacial $O(n)$, por estar G representado en una *matriz de adyacencia*. Por otro lado, la cantidad de llamadas recursivas está acotada por d, que a su vez está acotada por la cantidad de nodos del grafo. Por lo tanto obtendremos la siguiente ecuación de recurrencia.

$$
\begin{aligned}
T(n) &= T(n-1) + n && (1)\\
&= T(n-2) + n - 1 + n && (2)\\
&= T(n-3) + (n-2) + (n-1) + n && (3)\\
&= \ldots && (4)\\
&= T(0) + (1 + 2 + 3 + \ldots + n) && (5)\\
&= T(0) + \frac{n * (n+1)}{2} && (6)\\
&= O(n^2) && (7)
\end{aligned}
$$

La función CALCULARVARIANZA recorre la lista de ejes dada y, para cada uno de ellos, va sumando el valor de su peso en una variabe acumulada. Además calcula el promedio los pesos de la lista. Con estos valores se recorre la lista nuevamente y se realiza el cálculo de la varianza. Por lo tanto, como la máxima longitud que puede tener esta lista es m, y G es un árbol, esta función tiene complejidad temporal $O(n)$.

La función SUMATORIA tiene complejidad temporal $O(n)$, y su explicación es similar a la explicación de la complejidad de la función CALCULARVARIANZA.

La siguiente función determina cuáles ejes del AGM son inconsistentes y, en base a eso, arma los distintos *clusters* de G.

```
function ObtenerClusters(G =< V, E >)
    G' ← ConvertirAListaDeAristas(G)                                   ▷ O(n²)
    for e ∈ E(G') do                                                   ▷ O(n × ...)
        if EjeInconsistente(G', e, distancia, método, excesoDesvíoEstándar, excesoRatio) then   ▷ O(n²)
            G' ← E(G') − {e}                                           ▷ O(1)
            G ← < V, E(G) − {e} >                                      ▷ O(1)
        end if
    end for
    // El siguiente bloque de código define, para cada nodo v, a qué cluster pertenece
    pertenenciaCluster ← Dicc(n, null)                                 ▷ O(n)
    últimoCluster ← 0                                                  ▷ O(1)
    for v ∈ V(G) do                                                    ▷ O(n × ...)
        if pertenenciaCluster[v] = null then                           ▷ O(1)
            pertenenciaCluster[v] ← últimoCluster                      ▷ O(1)
            DefinirCluster(G, v, pertenenciaCluster, últimoCluster)    ▷ O(n²)
            últimoCluster ← últimoCluster + 1                          ▷ O(1)
        end if
    end for
    return pertenenciaCluster                                          ▷ O(1)
end function
```

La complejidad de la función OBTENERCLUSTERS es $O(n^3)$, con n la cantidad de nodos del grafo G.

Inicialmente se transforma el grafo G, a su representación con *lista de aristas* y lo llamamos G'. Esto se puede implementar con una complejidad temporal de $O(n^2)$, ya que G inicialmente está representado con una *matriz de adyacencia*. A continuación se itera sobre la nueva *lista de aristas*, que son exactamente $n - 1$ por ser G un AGM y se decide, para cada eje, si el mismo es inconsistente o no.

La función EJEINCONSISTENTE tiene complejidad temporal $O(n^2)$, cuya demostración se encuentra al pie del propio algoritmo.

El diccionario *pertenenciaCluster* está implementado sobre un arreglo de tamaño n, con lo cual inicializarlo con cada valor en `null` tiene complejidad temporal $O(n)$. Luego, tanto acceder como modificar un elemento del diccionario tiene complejidad temporal $O(1)$.

La función DEFINIRCLUSTER modifica el diccionario *pertenenciaCluster* de forma tal que el significado de cada nodo de G que es alcanzable desde v obtiene el mismo número de *cluster* que v. Esto se hace recorriendo los vecinos de v en la *matriz de adyacencia* de G y, para cada uno de ellos, se le asigna el valor *últimoCluster* en el diccionario. Luego se llama recursivamente a DEFINIRCLUSTER con el nodo vecino de

v para que toda la componente conexa obtenga el mismo número de *cluster*. En cada llamada se recorren todos los vértices en $O(n)$ y aquellos que son vecinos de v se le asigna el número de *cluster* en $O(1)$. Por lo tanto, la complejidad temporal de la función DEFINIRCLUSTER es $O(n^2)$ ya que utilizamos *matriz de adyacencia*. La demostración es análoga a la de la función OBTENEREJESCERCANOS ya que ambas tienen la misma ecuación de recurrencia.

Finalmente, los siguientes son algoritmos que dan una solución al problema de *clusterización* modelado a un grafo completo G.

```
function CLUSTERIZARCONPRIM(G =< V, E >)
    agm ← PRIM(G)                                          ▷ O(n^2)
    clusters ← OBTENERCLUSTERS(agm)                        ▷ O(n^3)

    return clusters
end function
```

```
function CLUSTERIZARCONKRUSKAL(G =< V, E >)
    agm ← KRUSKAL(G)                                       ▷ O(n^3)
    clusters ← OBTENERCLUSTERS(agm)                        ▷ O(n^3)

    return clusters
end function
```

```
function CLUSTERIZARCONKRUSKALPATHCOMPRESSION(G =< V, E >)
    agm ← KRUSKALPATHCOMPRESSION(G)                        ▷ O(n^2 × log(n))
    clusters ← OBTENERCLUSTERS(agm)                        ▷ O(n^3)

    return clusters
end function
```

Los algoritmos de Prim, Kruskal y Kruskal con *Path Compression* implementados son una de las versiones dadas por la cátedra. Estos se ejecutan con una representación de G en una matriz de adyacencia. En particular, se tomaron las implementaciones con complejidad $O(n^2)$, $O(n^3)$ y $O(n^2 \times log(n))$ para Prim, Kruskal y Kruskal con Path Compression respectivamente. Notar que, antes de ejecutar los algoritmos AGM, $O(m) = O(n^2)$, ya que G es un grafo completo, pero luego de la ejecucón de estos se tiene que $O(m) = O(n)$.

2.3. Análisis de resultados

En esta sección analizaremos si el tiempo de ejecución de los tres algoritmos propuestos para distintas instancias corresponden con la complejidad temporal teórica demostrada.

Para ello utilizaremos instancias que contengan una cantidad de puntos en el rango $\{0, 5, 10, \ldots, 400\}$. Los puntos serán sub-instancias de una instancia mayor de 400 puntos.

Para observar los resultados graficaremos la relación entre los tiempos de ejecución de cada algoritmo y la función de complejidad de cada algoritmo y, en consecuencia, el resultado debería converger a una función constante.

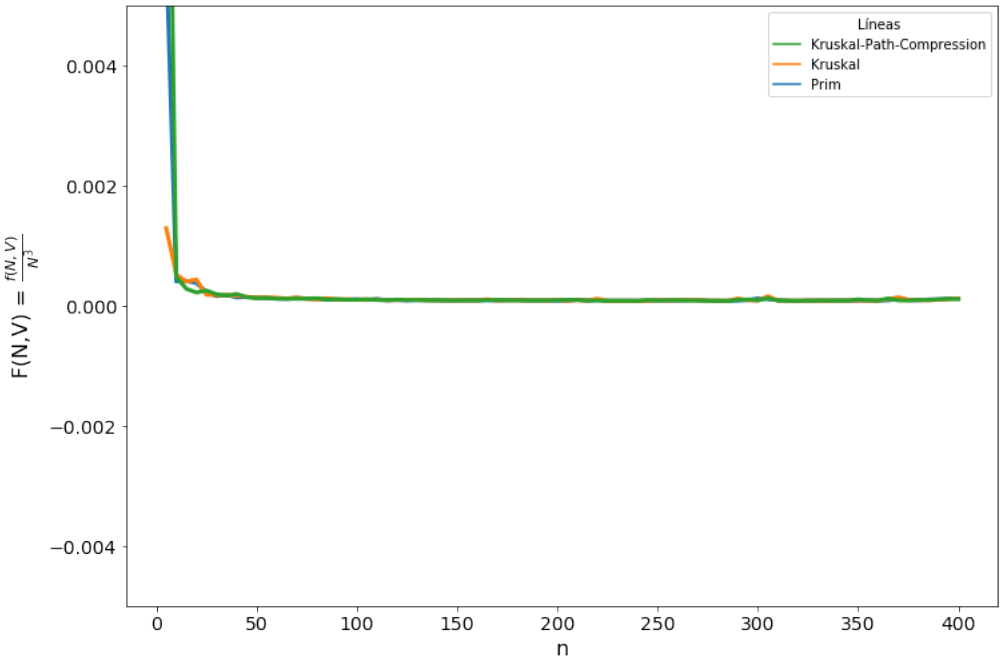

Figura 3: Relación entre los tiempos de ejecución de los algoritmos CLUSTERIZARCONPRIM, CLUSTERIZARCONKRUSKAL, y CLUSTERIZARCONKRUSKALPATHCOMPRESSION, y la función n^3 para valores $0 \leq n \leq 400$.

A través del gráfico de la figura en la Figura 3 puede observarse que las relaciones mostradas tienden a ser constantes a medida que aumenta el valor de n.

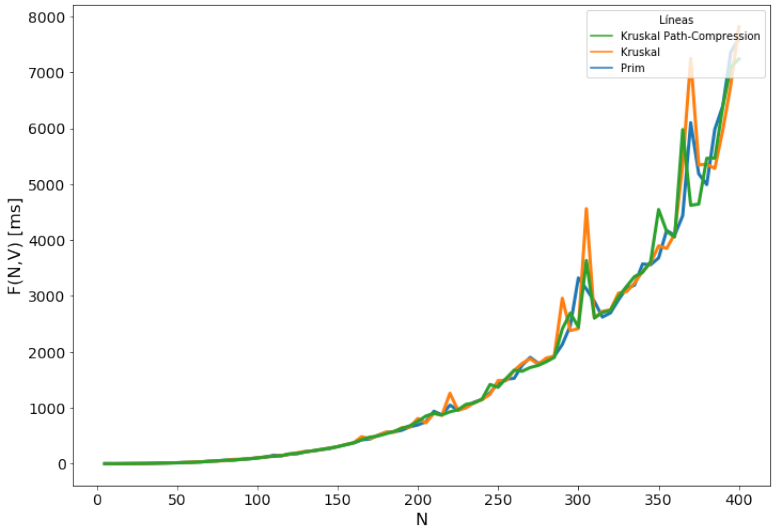

Figura 4: Comparación de los tiempos de ejecución de los algoritmos CLUSTERIZARCONPRIM, CLUSTERIZARCONKRUSKAL y CLUSTERIZARCONKRUSKALPATHCOMPRESSION utilizando la primer versión de *eje inconsistente*.

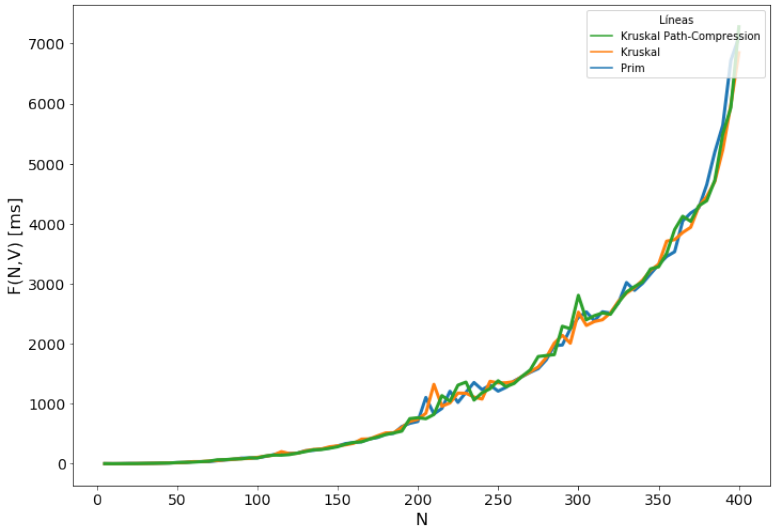

Figura 5: Comparación de los tiempos de ejecución de los algoritmos ClusterizarConPrim, ClusterizarConKruskal y ClusterizarConKruskalPathCompression utilizando la segunda versión de *eje inconsistente*.

En el gráfico de la Figura 4 se comparan los tiempos de ejecución de los algoritmos ClusterizarConPrim, ClusterizarConKruskal y ClusterizarConKruskalPathCompression utilizando la primer versión de *eje inconsistente*, mientras que el gráfico de la Figura 5 muestra la comparación entre los tiempos de ejecución de los algoritmos ClusterizarConPrim, ClusterizarConKruskal y ClusterizarConKruskalPathCompression utilizando la segunda versión de *eje inconsistente*. En estas mediciones se utilizaron las mismas intancias mencionadas previamente.

Como podemos ver en ambos gráficos, la hipótesis de que se comportan de manera muy similar es correcta. Esto se puede explicar debido a que los tres algoritmos tienen la misma complejidad temporal y lo que marca las pequeñas diferencias entre los algoritmos es debido a la diferencia de constantes en sus complejidades.

2.4. Experimentación

2.4.1. Parámetros óptimos

La idea de este experimento es variar los parámetros requeridos para determinar si un eje es inconsistente o no y tratar de obtener los parámetros *óptimos* a nuestro criterio. Para ello utilizaremos dos instancias en las cuales podemos determinar los *clusters a ojo*. La primer instancia corresponde a la de la Figura 1. Según nuestro criterio, la misma puede clusterizarse de forma bien definida, y dicha clusterización coincide con la mostrada en la Figura 2. La segunda intancia se muestra en la Figura 6 y, a nuestro criterio, una clusterización óptima sería la mostrada en la Figura 7

Utilizaremos únicamente el algoritmo Prim con ambas versiones de detección de *ejes inconsistentes*, debido a que la cantidad de ejes inconsistentes no depende del algoritmo de Árbol Generador Mínimo, si no que depende exclusivamente de los parámetros que utilizamos para definir dicha inconsistencia. Los parámetros con los que experimentaremos serán el tamaño de la vecindad, el exceso de desvío estándar y el exceso del promedio.

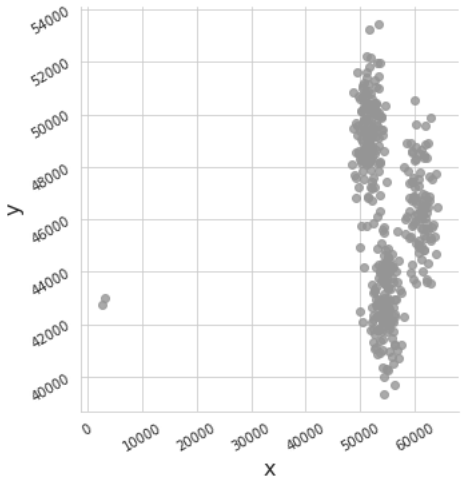

Figura 6: Conjunto de puntos del plano euclídeo de ejemplo

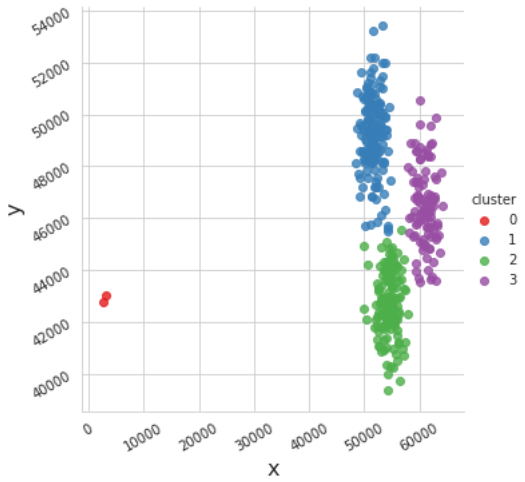

Figura 7: Posible *clusterización* del conjunto de puntos del plano euclídeo de ejemplo mostrado en la Figura 6

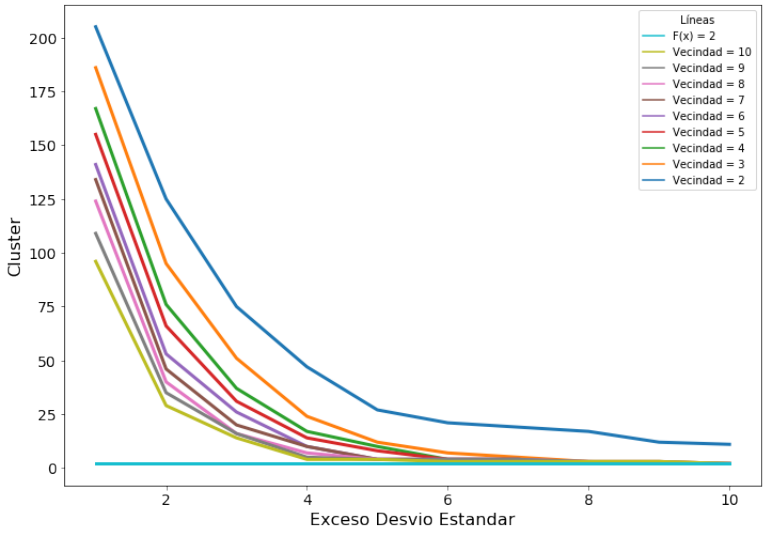

Figura 8: Cantidad de *clusters* en función de los parámetros $vecindad$ y $exceso\ desvío\ estándar$ obtenidos por el algoritmo Prim en la instancia de la Figura 1 utilizando la primer versión de *eje inconsistente*.

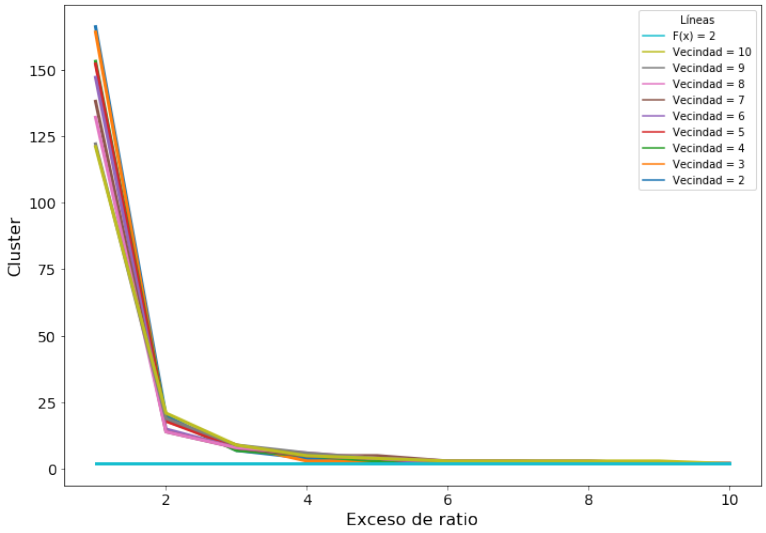

Figura 9: Cantidad de *clusters* en función de los parámetros $vecindad$ y $exceso\ de\ ratio$ obtenidos por el algoritmo Prim en la instancia de la Figura 1 utilizando la segunda versión de *eje inconsistente*.

El gráfico de la Figura 8 muestra la cantidad de *clusters* obtenidos por el algoritmo de Prim para distintos valores de distancia de vecinos y exceso de desvío estándar utilizando la primer versión de *eje inconsistente*. Como mencionamos anteriormente y como se ve en la Figura 2, a nuestro criterio, la cantidad de *clusters* óptima es 2; valor al que tiende el gráfico de la Figura 8.

El gráfico de la Figura 9 muestra la cantidad de *clusters* obtenidos por el algoritmo de Prim para distintos valores de distancia de vecinos y exceso de ratio utilizando la segunda versión de *eje inconsistente*. Como mencionamos anteriormente y como se ve en la Figura 2, a nuestro criterio, la cantidad de *clusters* óptima es 2; valor al que tiende el gráfico de la Figura 9.

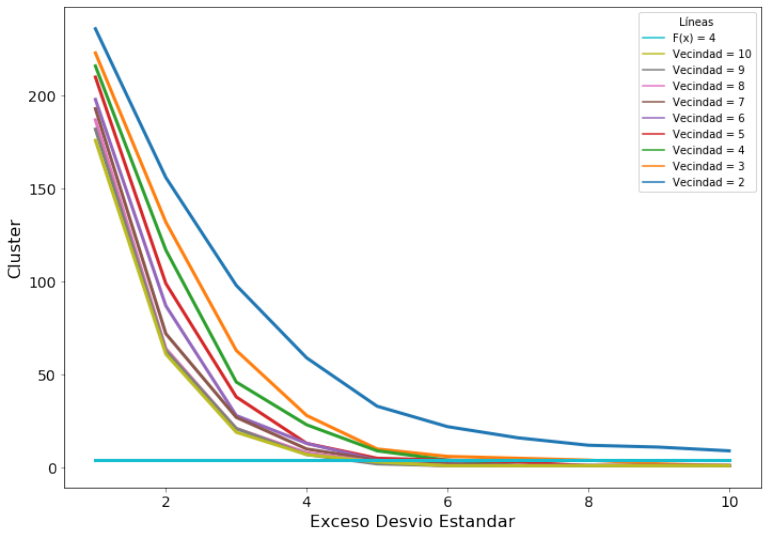

Figura 10: Cantidad de *clusters* en función de los parámetros $vecindad$ y $exceso\ desvío\ estándar$ obtenidos por el algoritmo Prim en la instancia de la Figura 6 utilizando la primer versión de *eje inconsistente*.

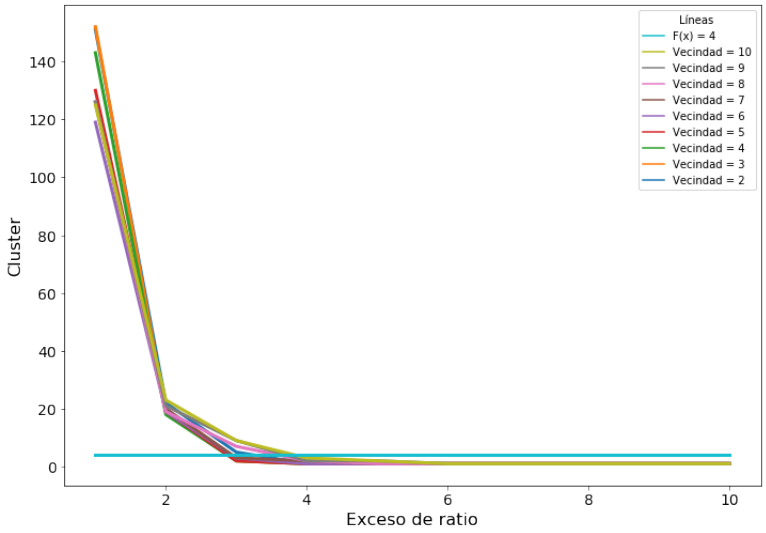

Figura 11: Cantidad de *clusters* en función de los parámetros $vecindad$ y $exceso\ de\ ratio$ obtenidos por el algoritmo Prim en la instancia de la Figura 6 utilizando la segunda versión de *eje inconsistente*.

El gráfico de la Figura 10 muestra la cantidad de *clusters* obtenidos por el algoritmo de Prim para distintos valores de distancia de vecinos y exceso de desvío estándar utilizando la primer versión de *eje inconsistente*. Como mencionamos anteriormente y como se ve en la Figura 7, a nuestro criterio, la cantidad de *clusters* óptima es 4; sin embargo, observando el gráfico de la Figura 10, la misma parece tender a 2. Creemos que esto sucede debido a que, utilizando la primera versión de *eje inconsistente*, a medida que aumenta el $exceso\ desvío\ estándar$, se reduce la cantidad de ejes inconsistentes y por lo tanto, los *clusters* definidos como $1, 2y3$ en la Figura 8 terminan definiendo un solo *cluster*.

El gráfico de la Figura 11 muestra la cantidad de *clusters* obtenidos por el algoritmo de Prim para distintos valores de distancia de vecinos y exceso de ratio utilizando la segunda versión de *eje inconsistente*. Como mencionamos anteriormente y como se ve en la Figura 7, a nuestro criterio, la cantidad de *clusters* óptima es 4; sin embargo, observando el gráfico de la Figura 11, la misma parece tender a 2. Creemos que esto sucede debido a que, utilizando la segunda versión de *eje inconsistente*, a medida que aumenta el $exceso\ de\ ratio$, se reduce la cantidad de ejes inconsistentes y por lo tanto, los *clusters* definidos como $1, 2y3$ en la Figura 8 terminan definiendo un solo *cluster*.

Las conclusiones del problema de *clusterización* son:

- En general, utilizando la primer versión de *eje inconsistente* se llega más rápido al valor óptimo, tal como se ve en las figuras 8 y 9 para la primera instancia, y 10 y 11 para la segunda. Esto sucede debido a que variar el $exceso\ de\ ratio$ de la primer versión de *eje inconsistente* supone un cambio más brusco en la definición de *eje inconsistente* con respecto a el $exceso\ desvío\ estándar$ de la segunda versión de *eje inconsistente*.
- En la instancia de la Figura 1 fue más sencillo generar una *clusterización* óptima, debido a que los están bien definidos, tal como se ve en la Figura 2. En cambio, en la instancia de la Figura 6 los *clusters* definidos como $1, 2y3$ en la Figura 7 no están tan bien definidos, y es posible que los algoritmos de clusterización definan esos *clusters* como un único *cluster*.

3. Segundo problema: Arbitraje

En el mundo de las finanzas se denomina arbitraje al hecho de comprar y vender un recurso de manera simultánea para generar una ganancia.

Particularmente en el caso de este trabajo práctico, el problema a resolver es el de decidir si existe oportunidad de arbitraje entre varias divisas (Pesos, Dólares, Libras, Reales, Bitcoins, etc.).

3.1. El Problema

> Dado un conjunto de n divisas diferentes y las tasas de cambio entre cada una de ellas. Decidir si existe oportunidad de arbitraje entre las divisas del conjunto y, si existe, exhibir un ciclo de divisas que producen el arbitraje.

Por ejemplo, dada la siguiente tabla que indica las tasas de cambio entre `ARS`, `USD` y `EUR`,

Moneda	ARS	USD	EUR
ARS	1	0.027	0.024
USD	36.69	1	0.86
EUR	42.5	1.16	1

la respuesta es sí, existe arbitraje, y un posible ciclo de arbitraje es `ARS` $\rightarrow$ `EUR` $\rightarrow$ `USD` $\rightarrow$ `ARS`, pues partiendo de `1 ARS` se obtienen `0.024 EUR`, que equivalen a `0.02784 USD`, y volviendo a la moneda inicial se obtienen `1.0214 ARS`, que es más que el valor inicial de `1 ARS`.

En este otro ejemplo tomamos los valores de compra-venta de divisas del Banco de la Nación Argentina al día 15 de octubre de 2018:

Moneda	ARS	USD	EUR
ARS	1	0.026	0.022
USD	35.8	1	0.78
EUR	42.5	1.10	1

En este caso no hay arbitraje posible y por lo tanto el mercado está equilibrado. Esto puede demostrarse constructivamente armando todos los ciclos simples posibles de cada divisa y viendo que con ninguno de ellos se puede obtener más unidades que la cantidad inicial en esa divisa .

3.2. Resolución

Para resolver el problema nos gustaría desarrollar un algoritmo que nos permita hallar una secuencia de cambios de divisas que comience y termine en la misma divisa, y que al multiplicar las tasas de cambio de dicha secuencia obtengamos un valor estrictamente mayor que 1 (partiendo de un valor inicial de 1 unidad de esa moneda). Es decir, si partimos de una unidad de la divisa i_1, queremos hallar un ciclo tal que

$$
\begin{aligned}
&1 < C(i_1, i_2) \times C(i_2, i_3) \times \ldots \times C(i_k - 1, i_k) \times C(i_k, i_1) && \iff \\
&1 > \frac{1}{C(i_1, i_2)} \times \frac{1}{C(i_2, i_3)} \times \ldots \times \frac{1}{C(i_k - 1, i_k)} \times \frac{1}{C(i_k, i_1)} && \iff \\
&0 > log\Big(\frac{1}{C(i_1, i_2)}\Big) + log\Big(\frac{1}{C(i_2, i_3)}\Big) + \ldots + log\Big(\frac{1}{C(i_k - 1, i_k)}\Big) + log\Big(\frac{1}{C(i_k, i_1)}\Big) && \iff \\
&0 > \Big(- log\big(C(i_1, i_2)\big)\Big) + \Big(- log\big(C(i_2, i_3)\big)\Big) + \ldots + \Big(- log\big(C(i_k - 1, i_k)\big)\Big) + \Big(- log\big(C(i_k, i_1)\big)\Big),
\end{aligned}
$$

donde $C(u, v)$ es el multiplicador que se le debe aplicar a una unidad de la divisa u para cambiar a la divisa v.

Sea A el conjunto de divisas las cuales se quiere analizar si existe oportunidad de arbitraje. Comenzaremos definiendo un grafo dirigido G donde cada nodo representa un elemento de A. Es decir, cada nodo de G es una divisa. Además, definimos que G es completo, con lo cual G es fuertemente conexo, y

todos sus nodos son universales. Por último, definimos el *peso* de una arista (u, v) como el resultado de aplicar la función $-log(C(u, v))$.

Ya habiendo modelado el grafo G, como queremos hallar una secuencia de divisas que comience y termine en la misma divisa, lo que estamos buscando es un ciclo. En particular, bastará con encontrar un **ciclo negativo** C en G para determinar si existe o no arbitraje ya que, si encontramos un ciclo C negativo, este cumple el bicondicional mostrado anteriormente. Es decir, sus nodos representarán divisas tales que, multiplicando sus tasas de cambio, se obtiene un valor estrictamente mayor que 1.

Precisamente, por lo concluído en el párrafo anterior, representaremos el problema con un grafo G según el modelo descrito anteriormente y utilizaremos un algoritmo que nos permita detectar ciclos negativos. Tanto el algoritmo de Bellman-Ford como el de Floyd-Warshall sirven para este fin. En caso de encontrar un ciclo negativo en G, como vimos anteriormente, podemos concluir que existe arbitraje, con lo cual, si el problema únicamente fuese el de decidir si existe o no arbitraje, esto sería todo lo que tendría que hacer nuestro algoritmo. Sin embargo, como el problema a resolver es otro, debemos además determinar alguna secuencia de transacciones de compra y venta de divisas que determinen y validen que efectivamente existe dicho arbitraje.

3.2.1. Resolución con algoritmo de Bellman-Ford

El algoritmo de Bellman-Ford recibe como parámetros un grafo G y un vértice $v \in G$ y devuelve el camino mínimo entre v y todos los vértices del G, incluyedo el camino mínimo de v a v. Además, como se mencionó anteriormente, detecta ciclos negativos en G. La elección del nodo sobre el cual ejecutar el algoritmo no es determinante para detectar si hay un ciclo negativo en G ya que, por cómo está modelado G, es completo y, en particular es conexo, con lo cual, para todo par de nodos $u, v \in G$, existe un camino de u a v. Esto quiere decir que podemos tomar cualquier nodo de G y siempre podemos llegar a un ciclo negativo (de existir) y, en particular, el algoritmo de Bellman-Ford detectará dicho ciclo.

Además de detectar ciclos negativos, el algoritmo Bellman-Ford almacena el camino mínimo entre un nodo v y todos los demás nodos. Entonces, en el caso de existir un ciclo negativo, para obtener alguno de ellos, partiremos desde el nodo inicial v e iremos avanzando en el camino almacenado por el algoritmo entre el nodo v consigo mismo. Esto lo realizaremos hasta obtener un nodo u que se repite. Entonces, u pertenecerá a un ciclo negativo. Una vez obtenido el nodo u continuaremos su camino hacia v, el cual nunca llegará ya que u pertenece a un ciclo negativo, y en algún momento se repetirá dicho elemento. De esta manera, el ciclo que une a u consigo mismo será la secuencia de compra-venta que hay que realizar en caso de que haya arbitraje.

La complejidad del algoritmo Bellman-Ford depende de la representación del grafo. En este caso el grafo está representado con una *matriz de adyacencia*, con lo cual, la complejidad es $O(n^3)$.

Luego, detectar si existe un ciclo negativo tiene complejidad temporal $O(1)$ ya que basta con comparar si la cantidad de iteraciones es mayor o igual a n.

Por último, una vez que sabemos que hay un ciclo negativo debemos encontrarlo. Como comenzamos desde un vértice y vamos obteniendo el siguiente hasta obtener un vértice repetido, en el peor caso tendremos que recorrer todos los vértices, es decir, n vertices. Por lo tanto, la complejidad de encontrar el ciclo en el peor caso es $O(n)$.

La conclusión es que utilizando Bellman-Ford con *matriz de adyacencia* para determinar si hay arbitraje o no y devolver la secuencia de transacciones en caso de que haya arbitraje tiene una complejidad total de $O(n^3)$.

```
1: function ARBITRAJE-BELLMAN-FORD(G =< V, E >, v)
2:     Π_v ← 0                                                          ▷ O(1)
3:     Π_i ← +∞                    ∀ i ≠ v                              ▷ O(n)
4:     ant_i ← v                                                        ▷ O(n)
5:     i ← 0                                                            ▷ O(1)
6:     hayCambios ← true                                                ▷ O(1)
7:     while hayCambios and i < n do                                    ▷ O(n × ...)
8:         hayCambios ← false                                           ▷ O(1)
9:         for (u, v) ∈ E do                                            ▷ O(n² × ...)
10:            if Π_u + peso(u, v) < Π_v then                           ▷ O(1)
11:                Π_v ←Π_u + w(u, v)                                   ▷ O(1)
12:                ant_v ←u                                             ▷ O(1)
13:                hayCambios ←true                                     ▷ O(1)
14:            end if
15:        end for
16:        i ← i + 1                                                    ▷ O(1)
17:    end while
18:    if i = n then                                                    ▷ O(1)
19:        // Hay ciclos negativos (hay arbitraje). Buscamos ahora un ciclo de divisas
20:            que genere el arbitraje
21:        yaPasé_j ← false            ∀ j ≠ v                          ▷ O(n)
22:        actual ← v                                                   ▷ O(1)
23:        do                                                           ▷ O(n × ...)
24:            yaPasé_actual ← true                                     ▷ O(1)
25:            actual ← ant_actual                                      ▷ O(1)
26:        while ¬yaPasé_actual                                         ▷ O(1)
27:        nodoArbitraje ← actual                                       ▷ O(1)
28:
29:        j ← nodoArbitraje                                            ▷ O(1)
30:        ciclo ← ∅                                                    ▷ O(1)
31:        do                                                           ▷ O(n × ...)
32:            AGREGAR(ciclo, j)                                        ▷ O(1)
33:            j ← ant_j                                                ▷ O(1)
34:        while j ≠ nodoArbitraje                                      ▷ O(1)
35:        AGREGAR(ciclo, nodoArbitraje)                                ▷ O(1)
36:        return true, ciclo                                           ▷ O(1)
37:    end if
38:    return false                                                     ▷ O(1)
39: end function
```

3.2.2. Resolución con algoritmo de Floyd-Warshall

El algoritmo de Floyd recibe como parámetro un grafo G y devuelve el camino mínimo entre todos los vértices de G. Además, como se mencionó anteriormente, detecta ciclos negativos en G.

Además de detectar ciclos negativos, el algoritmo Floyd almacena el camino mínimo entre cada par de nodos de G. Entonces, en el caso de existir un ciclo negativo, para obtener alguno de ellos, partiremos desde cualquier nodo cuyo camino mínimo hacia sí mismo (ciclo) sea negativo e iremos avanzando en dicho ciclo hasta reconstruirlo.

```
1: function ARBITRAJE-FLOYD(G =< V, E >, x)                                  ▷ O(n^3)
2:     Π_ii ← 0                                                              ▷ O(n)
3:     Π_ij ← W_ij                          ▷ Para todo i (Para todo j) i ≠ j ▷ O(n^2)
4:     Next_ij ← j                                                           ▷ O(n^2)
5:     k ← 0                                                                 ▷ O(1)
6:     for k < n do                                                          ▷ O(n × ...)
7:         i ← 0                                                             ▷ O(1)
8:         for i < n do                                                      ▷ O(n × ...)
9:             j ← 0                                                         ▷ O(1)
10:            for j < n do                                                  ▷ O(n × ...)
11:                if Π_ij > Π_ik + Π_kj then                                ▷ O(1)
12:                    Π_ij ←Π_ik + Π_kj                                     ▷ O(1)
13:                    Next_ij ← Next_ik                                     ▷ O(1)
14:                end if
15:            end for
16:        end for
17:    end for
18:    i ← 0                                                                 ▷ O(1)
19:    for i < n do                                                          ▷ O(n × ...)
20:        if Π_ii < 0 then                                                  ▷ O(1)
21:            yaPase_i ← false                                              ▷ O(n)
22:            actual ← i                                                    ▷ O(1)
23:            do                                                            ▷ O(n × ...)
24:                yaPase_actual ← true                                      ▷ O(1)
25:                actual ← next_iactual                                     ▷ O(1)
26:            while ¬yaPase_actual                                          ▷ O(1)
27:            nodoArbitraje ← actual                                        ▷ O(1)
28:            i ← nodoArbitraje                                             ▷ O(1)
29:            ciclo ← ∅                                                     ▷ O(1)
30:            do                                                            ▷ O(n × ...)
31:                AGREGAR(ciclo, i)                                         ▷ O(1)
32:                i ← next_{i,nodoArbitraje}                                ▷ O(1)
33:            while i ≠ nodoArbitraje                                       ▷ O(1)
34:            AGREGAR(ciclo, nodoArbitraje)                                 ▷ O(1)
35:            return true, ciclo                                            ▷ O(1)
36:        end if
37:    end for
38:    return false
39: end function
```

La complejidad del algoritmo Floyd es $O(n^3)$ con cualquier representación del grafo.

Luego, detectar si existe un ciclo negativo tiene complejidad temporal $O(n)$, ya que únicamente debemos verificar si algún elemento de la diagonal de la matriz de distancias mínimas es menor a 0.

Por último, una vez que sabemos que hay un ciclo negativo debemos encontrarlo. Como comenzamos desde un vértice y vamos obteniendo el siguiente hasta obtener un vértice repetido, en el peor caso tendremos que recorrer todos los vértices, es decir, n vertices. Por lo tanto, la complejidad de encontrar el ciclo en el peor caso es $O(n)$.

La conclusión es que utilizando Floyd para determinar si hay arbitraje o no y devolver la secuencia de transacciones en caso de que haya arbitraje tiene una complejidad total de $O(n^3)$.

3.3. Análisis de resultados

En esta sección analizaremos si el tiempo de ejecución de los dos algoritmos propuestos para distintas instancias corresponden con la complejidad temporal teórica demostrada.

Para ello utilizaremos instancias que contengan una cantidad de divisas en el rango $\{2, \ldots, 400\}$. Los factores de cambio pertenecientes al conjunto de monedas serán elegidos aleatoriamente en el rango de

$\{0, 1, \ldots, 100\}$.

Para observar los resultados graficaremos la relación entre los tiempos de ejecución de cada algoritmo y la función de complejidad de cada algoritmo y, en consecuencia, el resultado debería converger a una función constante.

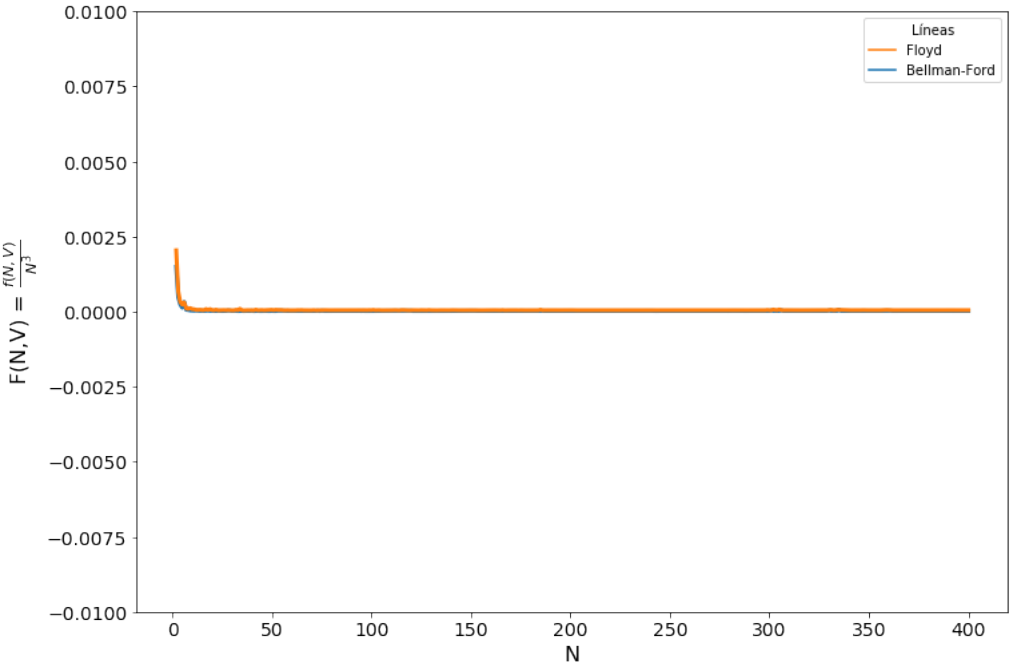

Figura 12: Relación entre los tiempos de ejecución de los algoritmos ARBITRAJE-BELLMAN-FORD y ARBITRAJE-FLOYD y la función n^3 para valores $2 \leq n \leq 400$.

A través del gráfico de la figura en la Figura 12 puede observarse que las relaciones mostradas tienden a ser constantes a medida que aumenta el valor de n.

3.4. Experimentación

3.4.1. Eficiencia

En este experimento vamos a verificar que Bellman-Ford se comporta de manera más eficiente que Floyd para las distintos tipos de instancias. Aunque la complejidad teórica sea igual, Bellman-Ford tiene constantes mucho más bajas que Floyd.

Para ello utilizaremos instancias que contengan una cantidad de divisas en el rango $\{5, 10, 15, \ldots, 400\}$. Los factores de cambio pertenecientes al conjunto de monedas serán elegidos de acuerdo a las distintas instancias que definiremos.

La primera comparación entre los algoritmos se hará sobre instancias en donde no haya arbitraje, es decir, que siempre se genere pérdida. Estas instancias se construyen tomando factores de cambio en las cuales la compra-venta de una moneda genera pérdidas, es decir, siempre se pierda dinero al pasar de una moneda a otra. En particular, se generan instancias calculando un factor de cambio cualquiera para un par de monedas i, j y restando un valor de *spread* ([2]), que es la diferencia entre el precio de compra y el de venta de un activo financiero. Esto hará que cualquier cambio que se realice genere una pérdida, por lo tanto, no habrá arbitraje.

Al utilizar estas instancias, los caminos mínimos entre cualquier par de nodos u y v será el eje que los une, ya que utilizando cualquier otro eje tendríamos un camino de mayor peso. Entonces, la hipótesis será que el algoritmo de Bellman-Ford se comportará de forma más eficiente que Floyd, ya que Bellman-Ford, si no detecta cambios en una iteración, finaliza su ejecución. Esto generará que Bellman-Ford únicamente consulte cada eje una única vez. Por otro lado, Floyd siempre tendrá que ejecutar sus tres ciclos de manera que su tiempo de ejecución será igual o mayor que el tiemp ode ejecución de Bellman-Ford con dichas instancias.

La segunda comparación entre los algoritmos se hará sobre instancias en donde haya arbitraje en todo ciclo del grafo, por lo tanto, todo intercambio que comienza y termina en la misma moneda genera ganancias.

Aquí Bellman-Ford no finalizará tan *rápidamente* ya que en las n iteraciones que hace habrá cambios.

La hipótesis es que, en la primera comparación, Bellman-Ford sera más eficiente, dado que detecta antes la ausencia de ciclos negativos, seguida por la segunda, dado que todo ciclo presenta arbitraje, todos los valores del arreglo de distancias son actualizados en cada iteración de Bellman-Ford.

En la figura 13 podemos ver los resultados de las instancias que no contienen arbitraje. Claramente se ve que, al no haber arbitraje, Bellman-Ford se comporta de manera mucho más eficiente que Floyd.

Este rendimiento muy superior de Bellman-Ford se explica por lo mencionado en la hipótesis, en la que que Bellman-Ford, si no detecta cambios en una iteración, finaliza su ejecución, y como, con este set de instancias, se generan pérdidas para todas las aristas entonces no se actualiza ningún camino en la primer iteración, el algoritmo efectivamente finaliza.

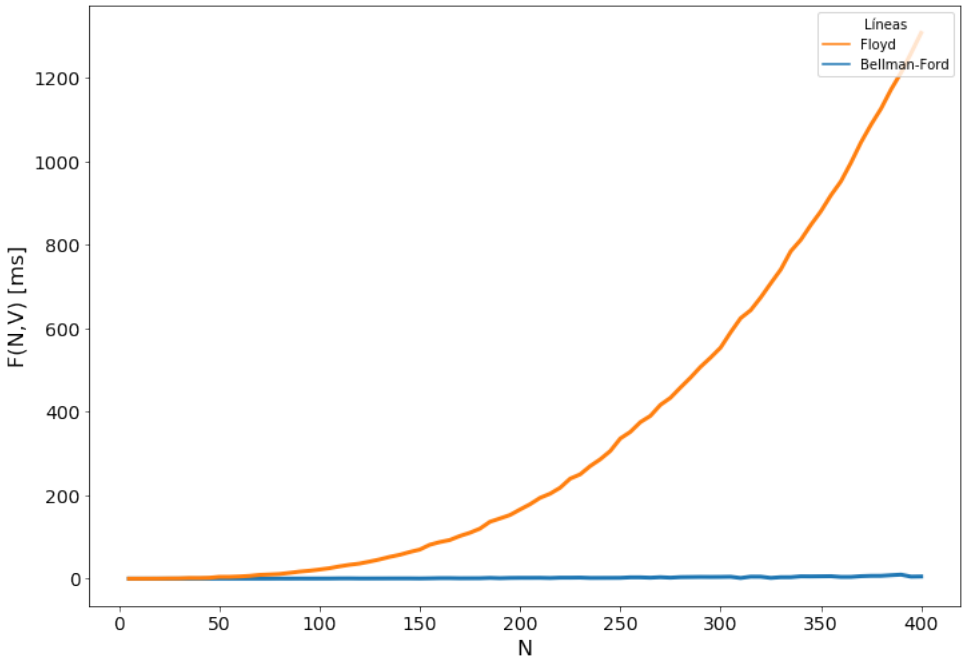

Figura 13: Comparación instancias sin arbitraje.

En el gráfico de la Figura 14 podemos ver los resultados de las instancias que contienen al menos una forma de realizar arbitraje. Claramente se ve que, al compararlo con el gráfico de la Figura 13, Bellman-Ford sigue comportándose mejor que Floyd, aunque que su eficiencia se acerca a la de este último. El *empeoramiento* del rendimiento se debe a que, por cómo es Bellman-Ford, éste debe iterar siempre n veces para lograr detectar que hay un ciclo negativo. La diferencia con respecto al caso anterior es que dicho caso, Bellman-Ford únicamente iteraba una vez, con lo cual era mucho más eficiente.

Por otro lado, Floyd también ha disminuido su eficiencia debido a que, al haber arbitraje, debe cambiar las distancias de los caminos mínimos de todos los pares de vértices. En el caso anterior, donde había arbitraje, Floyd nunca debía modificar las distancias.

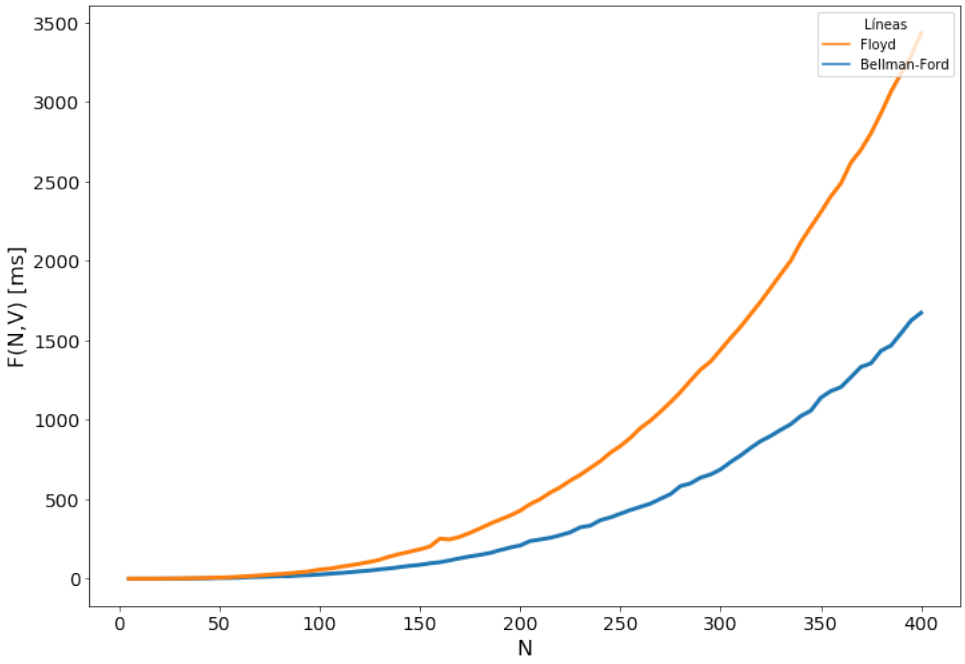

Figura 14: Comparación instancias con arbitraje

Como conclusión para el problema de arbitraje, tal como vimos en las Figuras 14 y 13, la utilización de Bellman-Ford siempre resulta más rápida para detectar si hay arbitraje o no. Por lo tanto, si tuviésemos que utilizar un algoritmo para detectar arbitraje en la vida real utilizaríamos Bellman-Ford.

Referencias

[1] C.T. Zahn, Graph-Theoretical Methods for Detecting and Describing Gestalt Clusters. *IEEE Trans. on Computers*, C-20(1):68-86, jan 1971

[2] `https://es.wikipedia.org/wiki/Spread`