DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Algoritmos y Estructuras de Datos III
Segundo Cuatrimestre de 2019

Integrante	LU	Correo electrónico
Agustín Vicente Dorda Recalde	314/15	agustindorda@gmail.com
Nicolás Pedro Casaballe	78/15	ncasaballe@dc.uba.ar
Philip Garrett	318/14	garrett.phg@gmail.com
Werner Busch		w.g.busch@hotmail.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. Separando la paja del trigo

1.1. Introducción

En *aprendizaje por computadora* uno de los problemas más fundamentales es el de *clusterizar*. Clusterizar significa agrupar un conjunto de objetos en diferentes subconjuntos por alguna nocion de similaridad intraconjunto (o disimiliradad interconjunto). Para esta aplicación, tenemos una nube de puntos en un espacio métrico, y nos gustaría realizar divisiones (*clusters*) que agrupen a los conjuntos de puntos que se encuentran juntos según la distancia Euclidiana, no deberían poder quedar nodos sin un conjunto y además los conjuntos deben formar una partición.

Para resolver el problema de la *clusterización* se modelarán los puntos con grafos, donde cada cluster corresponderá a una componente conexa. Como muestra *Charles Zahn*, los árboles generadores mínimos ”preservan”la clusterización, serán un recurso muy importante a lo largo de este trabajo, pues la idea principal detrás de este proyecto es utilizar un árbol generador mínimo para obtener un camino mínimo a través de todos los puntos del eje cartesiano y luego recortar los ejes correspondientes que separen un cluster de otro, de esta manera generando cada componente conexa relacionada con cada cluster, donde el criterio que decida si un eje debe ser cortado o no será si el largo de este eje en cuestión es demasiado grande o no en comparación con el resto de los ejes del grafo. Esta es la noción de *eje inconsistente* que presenta *Zahn*

Sin embargo, al no haber definición formal de *cluster* la validez de una disposición depende del uso que se le dé. Se estudiarán diversos criterios para armar *clusters* que serán *coherentes*. Motivados por esto, intentaremos analizar y experimentar sobre las posibles soluciones para cada problema de este tipo y cómo fluctuan en función de los algoritmos utilizados y sus parámetros de entrada.

1.1.1. Sobre el AGM

Como se menciono en la subsección anterior, el *árbol generador mínimo* de un grafo *conexo* preservar los clusters. Se modela la lista de puntos a *clusterizar* como un grafo completo, donde cada punto está conectado con todos y cada eje entre dos nodos u, v tiene un peso asignado que denota la distancia Euclidiana entre el nodo u y el nodo v. Dado este grafo G, se procede a generar un árbol generador mínimo T asociado a G. De esta manera nos quedaremos sólo con las distancias más cortas entre los nodos de nuestro problema de tal manera que todos los nodos permanezcan conectados a través de un camino. Ya existen varios algoritmos que resuelven el problema de hallar el AGM en tiempo razonable, Prim y Kruskal, y ambos seran utilizados para darlo. Una vez calculado el árbol generador mínimo T de G, la idea principal para la construcción de una *clusterización válida* de una lista de puntos consiste en **recortar los ejes inconsistentes** del árbol T, es decir, remover aquellos ejes que dispongan de un peso atípicamente alto en comparación con el de los ejes que lo rodean. Realizar esta tarea no es un paso trivial y el algoritmo encargado del recorte de estos ejes sobresalientes funciona con una heurística que depende de tres parámetros regulables.

1.2. Resolviendo el problema

1.2.1. Armando un grafo completo

Armar un grafo completo, dada una lista de nodos, consistirá en dar $\forall v, w \in lista$ el eje v, w

Algorithm 1 Armando ejes de grafo completo

```
1:  function CONSTRUIREJES(Vector points)                          ▷ O(n^2)
2:      Lista ejes                                                 ▷ O(1)
3:      for v < |points| do                                        ▷ O(n)
4:          for u ← v + 1 < |points| do                            ▷ O(n)
5:              Point v ← points[v]                                ▷ O(1)
6:              Point u ← points[u]                                ▷ O(1)
7:              float d ← distanciaEuclidiana(v, u)                ▷ O(1)
8:              ejes.push_back((u, v, d))                          ▷ O(1)
9:          end for
10:     end for
11:     return _ejes
12: end function
```

1.2.2. Algoritmos de AGM

Para aplicar los algoritmos de Kruskal y Prim, se debe proveer un grafo conexo con peso asignado a las aristas. En nuestro caso, como construimos un grafo completo, que es conexo porque cada vértice es adyacente con todos los demás, y además el peso son las distancias euclidianas de los puntos, entonces se verifica la precondición de Kruskal y Prim.
Para la implementación de Kruskal, se implementará un Union Disjoint Set sobre un árbol para separar las componentes conexas durante la iteración, con *path compression* y sin, y el ordenamiento de los ejes será con un algoritmo de orden $O(n + mlogn)$, por lo que se espera la complejidad de la ejecución de Kruskal sea $O(n + mlogn)$ siendo m el cardinal de las aristas y n el de los vértices.
Para la implementación de Prim, se dará la una cola de prioridad sobre *heap*, se espera la complejidad de Prim sea $O(n + mlogn)$ siendo n la cantidad de aristas

1.2.3. Recortando ejes

El algoritmo de recorte recibe cuatro parámetros:

- Un árbol generador mínimo
- Un valor $\sigma \in \mathbf{R}$ que refleja la cantidad máxima de veces que debe ser superada la desviación estándar de un vecindario por un eje para que este último pueda llegar a ser considerado inconsistente
- Un valor $f \in \mathbf{R}$ que especifica el valor máximo de veces que debe ser superado el promedio de la distancia de los ejes del vecindario por otro para que este último sea considerado inconsistente siempre y cuando cumpla también los requerimientos del punto anterior
- Un valor de profundidad $d \in \mathbf{Z}$ que delimita la distancia máxima a explorar del vecindario de un eje

Nuestro proceso comienza iterando por sobre todas las listas de adyacencia de primer nivel del grafo. Luego, se itera sobre todos los ejes que se encuentren dentro de cada una de ellas. A continuación se decidirá si cada eje es o no consistente según el criterio de consistencia que describe el $paper$ de Zahn *(Graph-Theoretical Methods for Detecting and Describing Gestalt Clusters, 1971:72)*.

Dado que nos encontramos manipulando un árbol, eliminar un eje de esta estructura generaría un bosque con 2 componentes conexas; es decir, esta nueva componente representa la existencia de un cluster distinto. Por esta razón, nos resulta conveniente eliminar aquellos ejes del árbol que sean anormalmente largos en comparación con el resto de los ejes que lo rodean, pues esto sugiere la presencia de un posible *salto entre clusters*. Estas aristas son las consideradas **inconsistentes** según el paper de **Charles Zahn** y el algoritmo las eliminará.
A continuación se presenta el pseudo-código del algoritmo de recorte:

Algorithm 2 Algoritmo de recorte ▷ $\mathcal{O}(|E| * (|V| + \triangle(grafo)))$

```
1: function PRUNEEDGES(grafo,σ ∈ R,f ∈ R,d ∈ Z)
2:     for eje e ∈ grafo do                                  ▷ O(|E| * (|V| + △(grafo)))
3:         if El eje e es inconsistente (utilizando σ, f y d) then   ▷ O(|V|)
4:             Eliminar el eje e de grafo                    ▷ O(△(grafo))
5:         end if
6:     end for

7:     return grafo
8: end function
```

1.2.3.1 Estudiando los ejes inconsistentes

Este algoritmo tiene la función de, dado un eje, un grafo implementado sobre lista de adyacencias y tres parámetros de exploración, determinar si la arista en cuestión es consistente.

El algoritmo logra esto tomando los dos vértices u, v que componen al eje aportado como parámetro y explorando sus vecindarios por separado; es decir, todos los ejes que están cerca de u y luego todos los ejes que están cerca de v (sin contar el mismo eje que los compone) a una profundidad de no más de d aristas. El rejunte de estos ejes es realizado por otro algoritmo que llamaremos **el recolector de pesos**. De esta manera, se generan dos conjuntos de ejes: el vecindario de u y el vecindario de v. A continuación, se calcula la media y la desviación estándar de los pesos de las aristas de ambos conjuntos por separados y proponemos el siguiente criterio: un eje es consistente pertenece al vecindario de u o al vecindario de v. Un eje pertenece a un vecindario si cumple las siguientes condiciones al mismo tiempo:

a) Ninguno de los dos vecindario es vacío

b) La distancia del eje es menor o igual al promedio de distancias del vecindario sumadas con su desviación estándar multiplicadas por un factor σ

c) La distancia del eje es menor o igual al promedio de distancias de vecindario multiplicadas por el factor f

Si ambos vecindarios de un eje son vacíos, significa que el eje en cuestión está uniendo dos vértices que no poseen información de referencia para juzgar su consistencia. En este caso, se decide eliminar el eje y separar ambos vértices en distintos clusters. Por otro lado, las condiciones **(b)** y **(c)** utilizan las distancias de los ejes como principal elemento comparativo. El punto **(b)** sigue el enfoque propuesto por **Charles Zahn** que consiste en delimitar las dimensiones de un eje a través de la desviación estandar de uno de sus vecindarios multiplicado por el factor σ (que generalmente es 3) mas el promedio de distancias del mismo vecindario. En un principio, nuestro algoritmo sólo disponía de las condiciones **(a)** y **(b)**, hasta que comenzaron a aparecer casos bordes donde ejes muy largos sobrevivían el recorte cuando no debían. La razón se encontraba en la cantidad de ejes inconsistentes dentro del vecindario de un eje. De existir un vecino lo suficientemente grande como para opacar la inconsistencia de la arista en análisis, se podían llegar a producir falsos negativos a la hora de concluir la inconsistencia del eje en cuestión. Es por esto que además se decidió agregar la condición **(c)** para atajar estos casos.

A continuación se presenta el pseudo-código de la función en cuestión:

Algorithm 3 Consistencia de un eje $\triangleright\ \mathcal{O}(|grafo|)$

1: **function** EDGEISINCONSISTENT(Grafo $grafo$, Eje e,$\sigma \in \mathbf{R}$, $f \in \mathbf{R}$,$d \in \mathbf{Z}$)
2: $v\_pesos \leftarrow$ **recolectarPesos**$(grafo, e.v, e.u, d)$ $\triangleright\ \Theta(|V| + |E|)$
3: $u\_pesos \leftarrow$ **recolectarPesos**$(grafo, e.v, e.u, d)$ $\triangleright\ \Theta(|V| + |E|)$

4: $v\_promedio \in \mathbf{R} \leftarrow$ **calcularPromedio**(v_pesos) $\triangleright\ \mathcal{O}(|V|)$
5: $u\_promedio \in \mathbf{R} \leftarrow$ **calcularPromedio**(u_pesos) $\triangleright\ \mathcal{O}(|V|)$

6: $v\_stdev \in \mathbf{R} \leftarrow$ **calcularDesviacionEstandar**(v_pesos) $\triangleright\ \mathcal{O}(|V|)$
7: $u\_stdev \in \mathbf{R} \leftarrow$ **calcularDesviacionEstandar**(u_pesos) $\triangleright\ \mathcal{O}(|V|)$

8: **bool** $v\_esConsistente \leftarrow v\_pesos \neq \varnothing \wedge e.distance \leq v\_promedio + \sigma * v\_stdev \wedge e.distance \leq v\_promedio * f$
9: **bool** $u\_esConsistente \leftarrow v\_pesos \neq \varnothing \wedge e.distance \leq u\_promedio + \sigma * u\_stdev \wedge e.distance \leq u\_promedio * f$

10: **if** $v\_pesos = \varnothing \wedge u\_pesos = \varnothing$ **then**
11: **return** $false$
12: **end if**

13: **return** $\neg(v\_esConsistente \vee u\_esConsistente)$
14: **end function**

1.2.3.2 Recolección de pesos

El algoritmo de recolección de pesos es el encargado de realizar la exploración sobre el vecindario de un vértice con una profundidad parametrizada d y agregar el peso de cada eje encontrado a una lista para luego retornarla. El método es invocado con la lista de adyacencias, un nodo v, un nodo padre p y la profundidad d. Como puede ser apreciado en el pseudo-código de la función de cálculo de la consistencia de un eje e, este recolector es invocado dos veces, o sea una para cada vértice del eje e. La primera invocación lo hace con el parámetro $v = e.v$ y $p = e.u$. El algoritmo itera sobre cada vértice u vecino al nodo v y siempre y cuando $u \neq p$, agrega el peso del eje u–v a la lista de acumulación. A continuación, si $d \neq 1$ – o sea si debo seguir profundizando la exploración del vecindario –, uno al conjunto de pesos la recursión invocada con el mismo grafo, el siguiente nodo s definido como el vértice extremo a v, el nodo v como padre y la profundidad d decrementada en 1 como parámetros.

De esta manera, se logran obtener los pesos de todos ejes a una profundidad d de un vértice v sin contar el peso del eje formado por los nodos v–p. La complejidad de este algoritmo es fácilmente demostrable, pues en esencia es un **Depth First Search** que inicia con el nodo parámetro p como **vértice cubierto** y la condición de cobertura es justamente que ningún vértice sea el padre p para contabilizar un eje. Además, como el grafo se trata de un árbol, no es necesario mantener un conjunto de vértices cubiertos pues no hay ciclos por lo tanto sobre cada vértice sólo va a haber uno que ya haya sido visitado y este es el ya conocido vértice padre p. En conclusión, las complejidades de este algoritmo son las mismas que la de **DFS**: $\Theta(|V| + |E|)$ temporal y $\mathcal{O}(|V|)$ espacial.

Algorithm 4 Recolección de ejes de un vecindario ▷ $\Theta(|V| + |E|)$

```
1: function COLLECTEDGEWEIGHTS(Grafo grafo, Nodo nodo, Nodo parent, Entero d)
2:     if |grafo[nodo]| = 1 then
3:         return ∅
4:     end if

5:     Lista < R > pesos ← ∅
6:     for hijo ∈ Z ← 0 hasta |grafo[nodo]| do
7:         Eje eje ← grafo[nodo][hijo]

8:         if eje.u ≠ parent ∧ eje.v ≠ parent then
9:             pesos ← pesos ∪ |eje|

10:            if d ≠ 1 then
11:                Nodo siguiente ← (if nodo = eje.u then eje.v Else eje.u)
12:                pesos ← pesos ∪ collectEdgeWeights(grafo, siguiente, nodo, d − 1)
13:            end if
14:        end if
15:    end for

16:    return pesos
17: end function
```

El grafo está implementado sobre lista de adyacencias

1.2.3.3 Eliminación de ejes inconsistentes

La remoción de las aristas inconsistentes se hace a través de la función **delete_edge** que toma una lista de ejes (que corresponde a un vértice en una lista de adyacencias) y el eje a eliminar en cuestión. El funcionamiento del algoritmo es bastante sencillo, pues esencialmente es una iteración sobre todos los ejes de la lista, es decir, es lineal sobre la cantidad de ejes en la secuencia de ejes y luego una remoción por índice que en la implementación también es ejecutada linealmente sobre la cantidad de ejes que le siguen en la lista al elemento a eliminar, dando una complejidad total de $\mathcal{O}(|ejes|)$. Se puede concluir que, para cualquier lista de ejes del grafo G, la complejidad de la función será $\mathcal{O}(\triangle(G))$, con $\triangle(G)$ siendo el grado máximo de G, es decir, la cantidad máxima de ejes que hay conectados a un vértice.

Algorithm 5 Eliminación de eje ▷ $\mathcal{O}(|ejes|)$

```
1: function DELETEEDGE(Lista < Eje > ejes, Eje e)
2:     for Eje k en ejes do                                   ▷ O(|ejes|)
3:         if ejes[k] = e then
4:             Remover k de ejes
5:         end if
6:     end for
7: end function
```

1.2.3.4 Algoritmos de promedio y desviación estándar

Estos algoritmos son utilizados a la hora de calcular el promedio y la desviación estándar del vecindario de un vértice v calculado mediante el algoritmo de **recolección de ejes**. Las complejidades de estos algoritmos son lineales en el tamaño de los vecindarios. El tamaño de un vecindario está acotado superiormente por su profundidad y el grado máximo del grafo en cuestión, sabemos que como mucho un vértice tendrá $\triangle(G)$ vértices conectados y lo mismo se cumple para cada uno de ellos, por lo tanto, como mucho el tamaño de un vecindario V' cualquiera de un grafo $G = (V, E)$ será acotado por:

$$|V'| \leq \sum_{i=1}^{d} \triangle(G)^i = \frac{1 - \triangle(G)^{d+1}}{1 - \triangle(G)} - 1 \leq |V|$$

(por propiedades de series geométricas)

Se puede concluir que la complejidad de los algoritmos de cálculo del promedio y desviación estándar pertenecen a la clase $\mathcal{O}(|V|)$. A continuación se presentan los pseudo-códigos de las respectivas soluciones:

Algorithm 6 Cálculo de desviación estándar ▷ $\Theta(|pesos|)$

```
1: function CALCULATESTDEV(Lista < R > pesos)
2:     if pesos es vacío then
3:         return 0
4:     end if

5:     promedio ∈ R ← calcularPromedio(pesos)                ▷ Θ(|pesos|)
6:     varianza ∈ R ← 0

7:     for x ∈ pesos do                                      ▷ Θ(|pesos|)
8:         varianza ← varianza + (x − promedio)^2
9:     end for

10:    return sqrt(varianza / |pesos|)
11: end function
```

Algorithm 7 Cálculo de promedio ▷ $\Theta(|poblacion|)$

```
1: function CALCULATEMEAN(Lista < R > poblacion)
2:     if poblacion es vacío then
3:         return 0
4:     end if

5:     promedio ∈ R ← 0

6:     for x ∈ poblacion do                                  ▷ Θ(|poblacion|)
7:         promedio ← promedio + x
8:     end for

9:     return promedio / |poblacion|
10: end function
```

1.2.4. Algoritmo de clusterización

El algoritmo final de clusterizacion tendria esta pinta:

Algorithm 8 Clusterización de puntos cartesianos

```
1: function CLUSTERIZAR(Vector points)                       ▷ O(n^2)
2:     Lista ejes ← construirEjes(points)                    ▷ O(n^2)
3:     Grafo AGM ← calcularAGM(points)                       ▷ O(n + mlog(n))
4:     prune_edges(AGM)                                      ▷ O(m * (n + △(G)))
5:     Lista clusters ← findClusters(AGM)                    ▷ O(n)
6:     return _clusters
7: end function
```

Donde findClusters es una función que simplemente recorre las componentes conexas con union disjoint set, como hace el algoritmo de Kruskal.

La complejidad del algoritmo de clusterización a nivel macro es dada por la suma de las siguientes subcomplejidades:

- El algoritmo de búsqueda de un Árbol Generador Mínimo (Kruskal o Prim) en $\mathcal{O}(n + m * log(n))$
- El recorte de ejes (que se realiza dos veces con diferentes parámetros para mejorar la precisión de la clusterización) en $\mathcal{O}(m * (n + \triangle(G)))$, con $\triangle(G)$ acotable por n
- La búsqueda de componentes conexas sobre el resultado del algoritmo de recorte en $\mathcal{O}(n)$

Sea C el algoritmo de clusterización, y teniendo en cuenta que $m = \mathcal{O}(n^2)$ se concluye que la complejidad de C es:

$$
\begin{aligned}
C &= \mathcal{O}(n + m * log(n)) + \mathcal{O}(m * (n + \triangle(G))) + \mathcal{O}(n) && (1)\\
&= \mathcal{O}(n) + \mathcal{O}(m * log(n)) + \mathcal{O}(m * (n + \triangle(G))) + \mathcal{O}(n) && (2)\\
&= 2 * \mathcal{O}(n) + \mathcal{O}(m * log(n)) + \mathcal{O}(m * (n + \triangle(G))) && (3)\\
&= 2 * \mathcal{O}(n) + \mathcal{O}(m * log(n)) + \mathcal{O}(m * (2n)) && (4)\\
&= \mathcal{O}(m * (2n)) && (5)\\
&= \mathcal{O}(m * n) && (6)\\
&= \mathcal{O}(n^2 * n) && (7)\\
&= \mathcal{O}(n^3) && (8)\\
& && (9)
\end{aligned}
$$

1.3. Experimentación

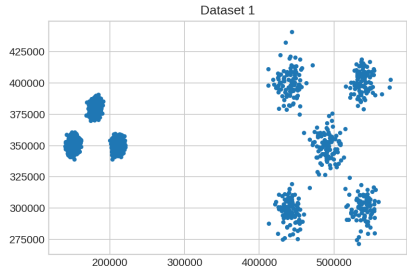
1.3.1. Criterios para evaluación de ejes

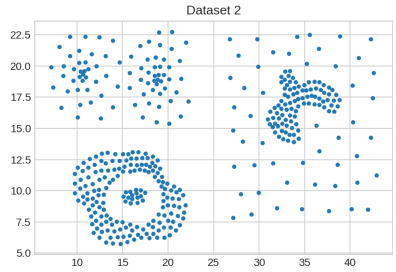
Al momento de decidir que ejes van a ser considerados inconsistentes (y por ende descartados), es necesario utilizar un criterio que se adapte a los diversos escenarios posibles. **Charles Zahn** propone utilizar el desvío estándar y la relación al tamaño de eje promedio para esto.

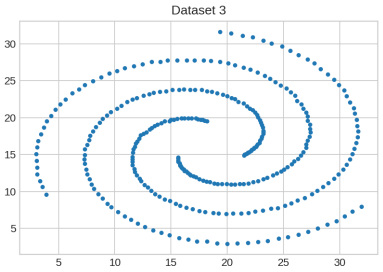
En esta sección vamos a explorar cuál composición de criterios se adaptan mejor a los distintos tipos de clusters. Las opciones a comparar son:

- Utilizar solo el desvío estándar
- Utilizar solo la relación al tamaño de eje promedio
- Utilizar el desvío estándar O la relación al eje promedio
- Utilizar el desvío estándar Y la relación al eje promedio

Para realizar las comparaciones vamos clusterizar los siguientes 3 grafos, definiendo como mas optimo al criterio que mejor arme clusters en todos los escenarios utilizando la menor cantidad de esfuerzo en elección de parámetros posible.



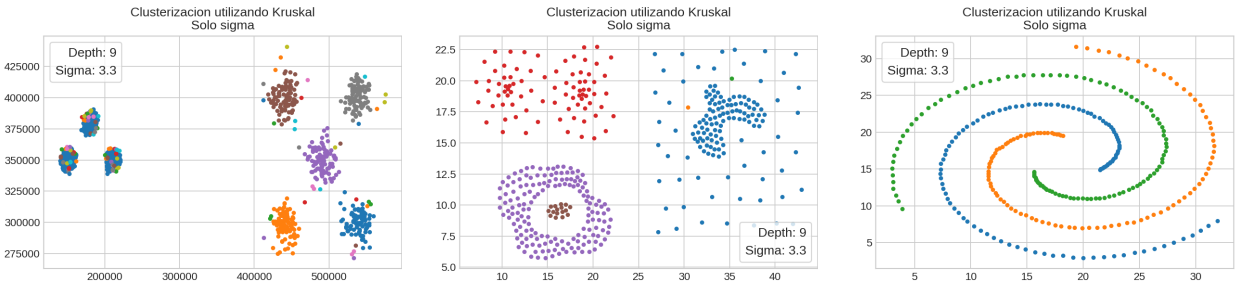




> **Nota**: Para todos los criterios vamos a utilizar una profundidad de exploración fija de 9 ejes adyacentes.

1.3.1.1 Solo desvío estándar

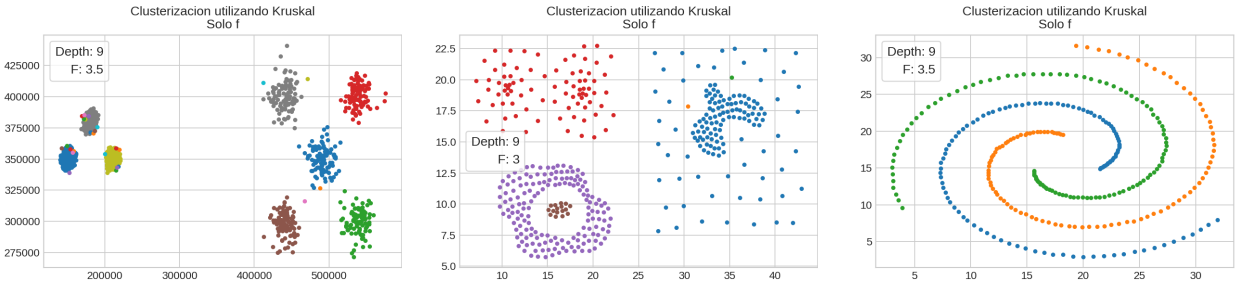
En este caso, vamos a descartar únicamente los ejes que son declarados inconsistentes al exceder en más de σ veces el desvío estándar aplicado sobre el tamaño de eje promedio del vecindario.



Como podemos ver, utilizar solo el desvío estándar sirve para algunos casos, sin embargo tiene problemas para el *dataset-1*, dado que en los clusters más chicos la desviación estándar entre los puntos es más alta, al estar todos más juntos. No podemos lograr un consenso de clusters sobre estos sin perjudicar la clusterización para los grupos de la derecha.

1.3.1.2 Solo relación sobre el eje promedio

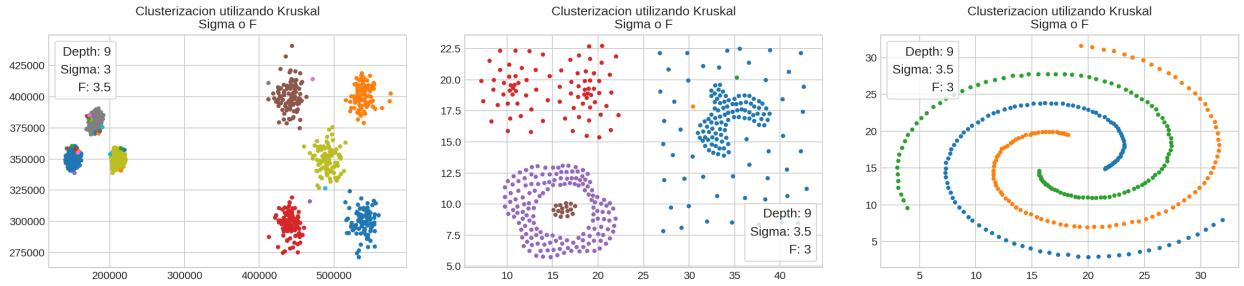
En este caso, vamos a descartar únicamente los ejes que son declarados inconsistentes al ser F veces más grandes que el tamaño de eje promedio del vecindario.



En este caso podemos ver que el *dataset-1* no es problema para este criterio, aunque hay que notar que para el caso del *dataset-2*, hubo que elegir un f distinto, ya que los clusters de lado izquierdo eran agrupados con el valor utilizado para los otros datasets.

1.3.1.3 Desvío estándar o relación de eje promedio

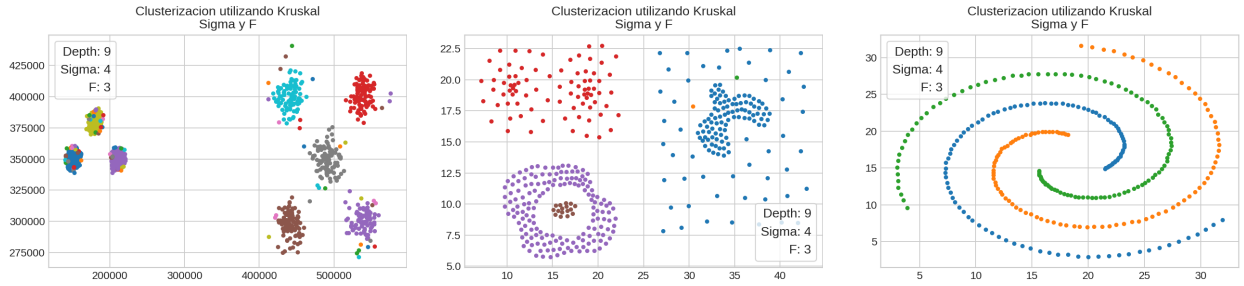
Para este caso, los ejes inconsistentes van a ser los que cumplan con alguno de los dos criterios anteriores



Al utilizar cualquiera de los dos criterios anteriores para descartar nodos, podemos ver que en general toma precedencia el criterio por 'f' (relación de tamaño con el eje promedio). Por lo que no observamos beneficios al incluir el desvío estándar en la comparación.

1.3.1.4 Desvío estándar y relación de eje promedio

Para este último caso, vamos a descartar únicamente los ejes que sean declarados inconsistentes tanto por el criterio del desvío estándar como por el criterio de relación de eje.



Siguiendo el caso anterior, podemos ver que usar la conjunción sólo resulta en peores resultados, dado que para forzar la clusterización en algunos grupos, debemos usar un valor alto para sigma, lo cual resulta en agrupamientos indeseados en otras partes del grafo.

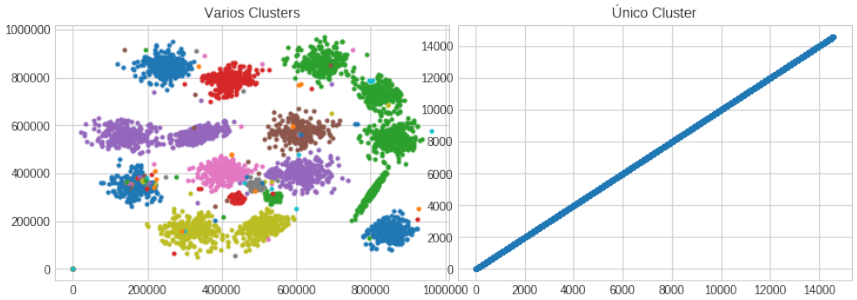
1.3.1.5 Conclusiones

Para los grafos elegidos para el experimento, resulta más conveniente utilizar únicamente la relación del eje contra el tamaño promedio de ejes. Como experimentos para profundizar, se deberían elegir grafos en los cuales la desviación estándar sea mayor al tamaño de eje promedio, en cuyo caso podríamos sacar provecho de combinar ambos criterios.

1.3.2. Kruskal vs Kruskal con path compression

Tal como fue presentado en la implementación del algoritmo de Kruskal, el mismo se puede implementar con un DSU con **path compression** que consiste en recordar el representante de un nodo buscado para poder accederlo en $\mathcal{O}(1)$ en la próxima consulta.

Para poder llevar a cabo dicha comparación, los escenarios planteados son representados con los siguientes grafos con 14600 puntos cada uno:



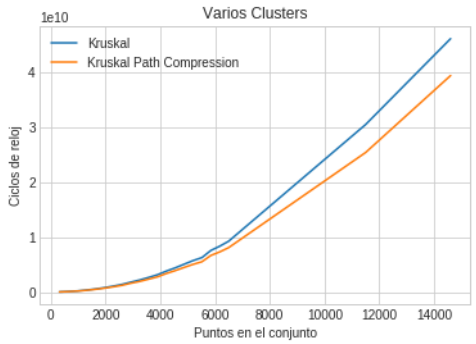
Nota: Los puntos de la curva representan el promedio de ciclos de reloj de 10 ejecuciones por punto. Para cada grafo, desde 0 hasta 6500 se generaron datos en intervalos de 325 puntos, luego se genero información para 11500 puntos y por último para 14600 puntos.

Los puntos a evaluar son:

- Performance entre Kruskal y Kruskal con path compression con varios clusters.
- Performance entre Kruskal y Kruskal con path compression con un único cluster.
- Evaluación de Kruskal con cota.

1.3.2.1 Performance con varios clusters

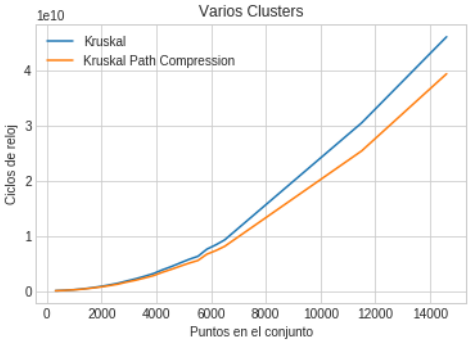
Al utilizar un grafo de 14600 puntos repartido en varios clusters como se pudo observar en la figura, se puede ver que cada cluster no contiene muchos puntos. Si bien, la alternativa de kruskal con path compression permite acceder al representante de un cluster en $\mathcal{O}(1)$, al tratarse de clusters con pocos puntos, se espera que haya una diferencia mínima en los ciclos de reloj.



Si bien la cantidad de ciclos de reloj entre una implementación y la otra comienzan a distanciarse a medida que se incrementan los puntos, la diferencia es mínima y se puede comprobar que la implementación con path compression se resuelve en menos ciclos.

1.3.2.2 Performance con un único cluster

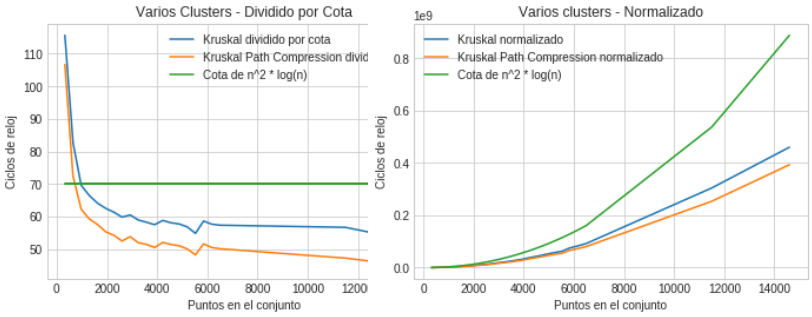
Al utilizar el grafo de 14600 puntos repartido en un único cluster se espera que la optimización de path compression sea más visible y haya una mayor diferencia en la ejecución de cada algoritmo.



Como se puede observar en la figura, los ciclos de reloj para cada algoritmo se distancian con una mayor diferencia que el caso anterior. Esto se debe a que la implementación normal de kruskal debe recorrer todo el arreglo de representantes, equivalente a recorrer todos los puntos ya analizados, para obtener el representante del cluster, que es único. En cambio, en el grafo anterior, los clusters contenían menos puntos, por ello la diferencia no era tan notable. En la última parte del grafo es posible ver como la pendiente de la ejecución para kruskal crece más rapidamente que para kruskal con path compression.

1.3.2.3 Evaluación de cota

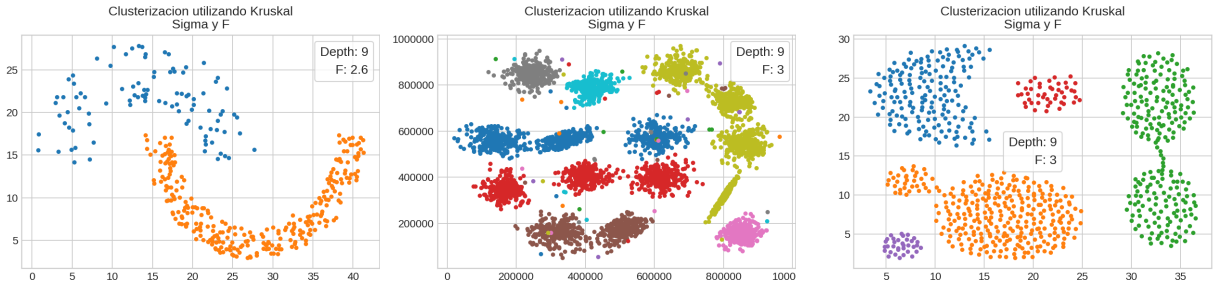
La complejidad de ambas implementaciones es la misma, por eso se espera que ante la cota tomada, ambos pasen a estar por debajo de la misma a partir de un cierto valor.

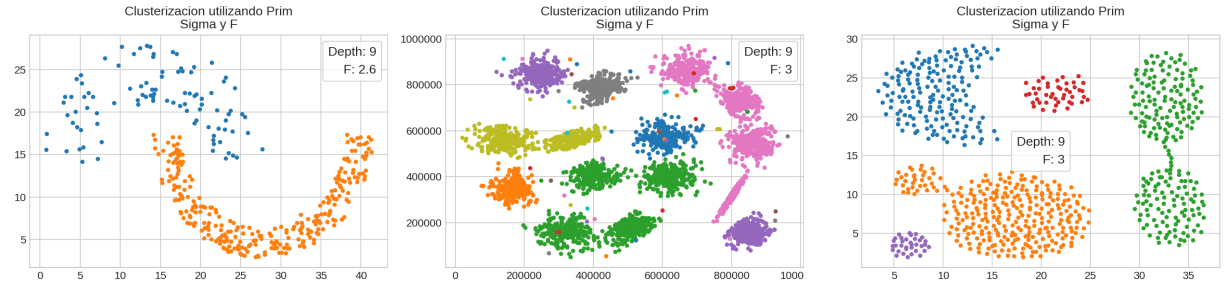


Como podemos ver, la funcion converge a un valor constante al dividirla por la complejidad planteada, lo cual demuestra que cumple con dicha cota.

1.3.3. Comparacion de Kruskal contra Prim

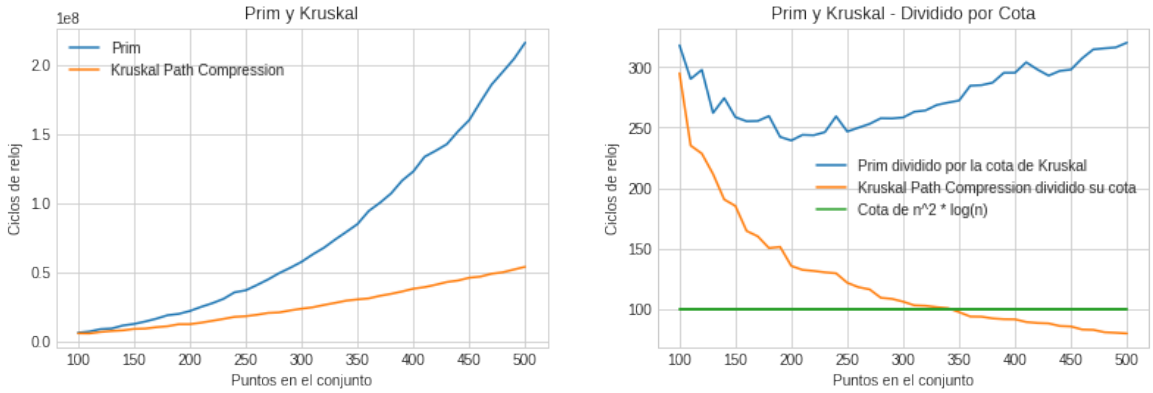
Al comparar Prim y Kruskal a la hora de clusterizar, podemos ver que ambos producen resultados similares. La diferencia entre ambos existe a la hora de generar el árbol generador mínimo, sobre el cual luego se descartan los ejes inconsistentes.





1.3.3.1 Conclusiones

Sin embargo, cabe destacar para Kruskal -al ser un algoritmo bottom up- armar los clusters una vez recortado el árbol generador mínimo es una tarea trivial, ya que el algoritmo se basa en formar clusters desconectados. En este aspecto, debemos realizar una solución Ad-Hoc para Prim, lo cual nos fuerza a obtener una complejidad mayor, a fuerza de no implementar Kruskal para clusterizar dentro de Prim.



2. Arbitraje

2.1. Introducción a la problemática

2.1.1. Noción de arbitraje

En finanzas, se denomina arbitraje a la práctica de comprar y vender un recurso de manera simultanea para generar una ganancia, aprovechandose de los desbalances de los precios en diferentes mercados. Supongamos que existe un recurso R disponible para la compra y venta en dos distintos mercados, M_1 y M_2.

En M_1, se puede comprar R por 42 pesos y vender por 41.8.

En M_2, se puede comprar R por 42,5 pesos y vender por 40,9.

Comprar en el segundo mercado para venderlo en el primero, o viceversa, no genera ganancia. Sin embargo, si el recurso se valoriza y M_1 se entera de este suceso: actualizará su cotización. De esta forma, ahora M_1 compra por 42,8 pesos el recurso y lo vende por 43,5. Hay oportunidad de arbitraje ya que M_2 no actualizó sus precios: por cada unidad comprada en M_2 y vendida en M_1 se obtiene 43,5-42,5 = 1 peso.

2.1.2. Arbitraje de divisas

El problema a resolver en este trabajo es analizar si existe oportunidad de arbitraje entre varias divisas (pesos, dolares, euros, etc). Para ello, contamos con la tasa de cambio para cada divisa.

La entrada consistirá de un primer entero n que corresponde a la cantidad de divisas a tener en cuenta. Luego, habrá n líneas. Cada una de ellas tendrá n numeros reales $c_{i,j}$, representando el multiplicador que

se debe aplicar a una unidad de la divisa i al cambiar a la divisa j. Si el arbitraje existe, la salida debe ser un posible ciclo de divisas, donde cada numero representará desde la divisa 0 a la $n-1$.

A continuación se presentan dos ejemplos, uno donde existe arbitraje y otro donde no.

	Entrada	de	ejemplo	Posible salida
4				3 2 1 3
1	0.1	5	0.125	
10	1	0.5	0.3	
0.2	2	1	2	
8	3	0.5	1	

	Entrada	de	ejemplo	Salida
4				NO
1	0.1	0.05	0.125	
0.1	1	0.5	0.3	
0.2	0.02	1	0.2	
0.8	0.3	0.5	1	

Por lo tanto, debemos hallar una secuencia de divisas tal que al convertir una unidad de la divisa inicial a través de las monedas, se genere una ganancia. Esto se traduce a encontrar una secuencia S

$$S = \{d_a, d_b, d_c, ..., d_k\} \quad \text{tal que} \quad c_{a,b} \text{ x } c_{b,c} \text{ x ... x } c_{k,a} > 1$$

En la entrada de ejemplo con solucion, un posible ciclo es $\{3, 2, 1, 3\}$ ya que c_{32} x c_{21} x c_{13} = 2 x 10 x 5 = $100 > 1$.

2.2. Análisis del problema

2.2.1. Traduciendo a grafos

2.3. Bellman-Ford

2.3.0.1 Resolución

Para resolver arbitraje usando Bellman Ford (**BF**) utilizamos una variación (**BF'**) del algoritmo clásico. **BF'** considera la *longitud* de un camino como el *producto* de sus aristas, y busca *maximizar* (y no minimizar) la longitud. Utilizamos esta variación ya que es la más natural dada el input que recibimos: si queremos pasar de un activo i a un activo j, tenemos que multiplicar por $c_{i,j}$ y no sumar. En esta variación del algoritmo, un arbitraje será un ciclo tal que el producto de sus aristas sea estrictamente mayor a 1.

Algorithm 9 Find arbitrage

```
1: function BELLMAN-FORD'(vector < Nodo > nodos, vector < Eje > ejes, Nodo inicial)
2:     Inicializar
3:     π = vector < int >[number_of_nodes]
4:     predecesor = vector < int >[number_of_nodes]
5:     for u ∈ nodos do
6:         π(u) = 0
7:         predecesor(u) = −1
8:     end for
9:     π(inicial) = 1

10:    for i = 0; i ≤ number_of_nodes; i + + do
11:        for e ∈ ejes do
12:            if π(e.terminal) < π(e.comienzo) ∗ e.peso then
13:                π(e.terminal) = e.comienzo ∗ e.peso
14:                predecesor(e.terminal) = e.comienzo
15:            end if
16:        end for
17:    end for

18:    for e ∈ ejes do
19:        if π(e.terminal) < π(e.comienzo) ∗ e.peso then
20:            vector < int > ciclo = reconstruir_arbitraje
21:            return ciclo
22:        end if
23:    end for

24:    return NO
25: end function
```

2.3.0.2 Justificacion

Podemos ensayar una justificación de porqué el algoritmo funciona apoyándonos en el homeomorfismo monótono decreciente $e^{-x} : (\mathbb{R}, +) \longrightarrow (\mathbb{R}_{>0}, .)$ y su inversa $-log(x) : (\mathbb{R}_{>0}, .) \longrightarrow (\mathbb{R}, +)$:

Sea (G, I) tal que $I(w, u) > 0 \forall (w, u)$. Este es el problema original que recibiremos, donde los pesos de las aristas es cuanto cuesta el cambio de un activo a otro. Defino $\tilde{G} = (G, -log(I))$, osea aplico $-log$ a los pesos. Sea $\tilde{\pi}$ una solución clásica de BF. Notar que la condición de que $\tilde{G}$ no tenga ciclos de longitud negativa alcanzables desde v es equivalente a que G no tenga ciclos de *longitud* (productoria) estrictamente mayor a 1 alcanzables desde 1 (es decir, que haya un arbitraje).

Sea ahora C un camino de v a u_0. Entonces por def. de solución de BF,

$$\tilde{\pi}(u_0) \leq l(C) = \sum \tilde{I}(a_i, a_{i+1}) = -\sum log(I(a_i, a_{i+1}))$$

Aplicando e^{-x},

$$e^{-\tilde{\pi}(u_0)} \geq e^{-(-\sum logI(a_i, a_{i+1}))} = e^{\sum logI(a_i, a_{i+1})}$$

Si defino π como $e^{-\tilde{\pi}}$, tengo que

$$\pi(u_0) \geq \prod I(a_i, a_{i+1})$$

Es decir, obtenemos una solución que maximiza los productos. Esto demuestra que pudimos haber hecho BF tradicional en $(G, -log(I))$, y al resultado aplicarle e^{-x} (coordenada a coordenada) para obtener una solución del problema de maximizar con el producto. Por otra parte, si tomamos el algoritmo de BF' como aparece en el pseudo-código y "aplicamos" la función $-log$, usando que "$-log(0)" = +\infty$ y que el resultado de aplicar $-log$ a

$$\pi(e.terminal) < \pi(e.comienzo) * e.peso$$

es

$$-log(\pi(e.terminal)) \geq -log(\pi(e.comienzo)) + -log(e.peso) \quad (10)$$

$$\tilde{\pi}(e.terminal) \geq \tilde{\pi}(e.comienzo) + \tilde{\pi}(e.peso) \quad (11)$$

Es decir, intuitivamente, podemos transformar el algoritmo modificado (BF') y obtener el BF tradicional, que sabemos que funciona (porque ya lo demostramos). Pero también podemos volver con $e^{-}x$ y obtener el algoritmo original (BF'). Por lo tanto BF' también funciona. Si bien intuitivamente es claro que la correspondencia entre $-log(x)ye^{-}x$ hace que la modificación del algoritmo funcione, no es formal. Una demostración formal podría hacerse por ejemplo copiando la demostración de BF clásica y reemplazando $+$ por ., $<$ por $\geq$ y haciendo los cambios correspondientes.

2.4. Floyd-Warshall

2.4.0.1 Resolución

El algoritmo de Floyd-Warshall es un algoritmo matricial que calcula la distancia entre todos los distintos nodos. Dado que la entrada del ejercicio es la matriz de tipos de cambio, podemos tomar a la misma como la matriz de entrada para el algoritmo, llamemosla L. Donde cada 'i', 'j' de la matriz representa el peso de la arista desde el nodo 'i' a 'j'. Sea l(ij) la funcion que define este peso.

Como parte del ejercicio es devolver el ciclo de arbitraje encontrado, debemos tener una manera de reconstruir dicho ciclo. Para ello, utilizaremos la matriz "next", una matriz de sucesores, que tiene el mismo tamaño que la matriz de entrada. La matriz "next.[es] inicializada con cada nodo indicando a si mismo como su sucesor.

Una vez que tenemos la matriz de sucesores, procedemos a resolver el camino máximo del grafo. Para ello, el algoritmo de Floyd-Warshall recurre a tres ciclos anidados donde recorremos la matriz. La idea es trabajar con una matriz "virtualrepresentada en la iteracion del ciclo principal, que llamaremos 'k'. Los otros dos ciclos recorren el valor de 'i' en el primer caso y el de 'j' en el ciclo interno. Por cuestiones de complejidad temporal, Floyd-Warshall nos permite siempre trabajar sobre la matriz inicial, sin necesidad de crear una nueva para cada paso. En cada paso del ciclo 'k', que itera el valor de 'k' entre 0 y 'n', el algoritmo calcula el camino máximo de un vertice 'i' a otro 'j' con vertices intermedios en el conjunto "0,....,k".

L^k es la matriz del paso 'k'. Lo cual nos permite la siguiente definicion:

- $L^0[i][j] = 0$ y para ineqj, $L^0[i][j]$ = l(ij) si ij pertence a los ejes del grafo. En este caso, el grafo es completo asi que el eje siempre existe.

- $L^{k+1}[i][j] = \max L^k[i][j], L^k[i][k] * L^k[k][j]$. Esta funcion no es la original del algoritmo de Floyd-Warshall, ya que aquí buscamos maximizar la longitud del camino. Esto es válido de la misma forma que la variante de Bellman-Ford presente en este trabajo práctico. Podemos encontrar la justificación de la utilización del producto en la sección 2.2.0.2, es decir, la misma justificación que Bellman-Ford. En este paso, tambien actualizamos el valor de la matriz de sucesores en caso de que el valor maximo sea el del segundo argumento. En dicho caso, como el camino del nodo 'i' a 'j' necesita pasar primero por el nodo 'k', podemos decir que el camino es igual al camino de 'i' a 'k'.

La matriz buscada es L^n.

Al final de cada iteracion del ciclo 'j', el algoritmo verifica si el elemento en la posicion $[i][i]$ es mayor a 1. Esto es equivalente a buscar si se formo el ciclo de arbitraje. Esto se hace al final de cada iteracion de 'j' porque significa que ya actualizamos la distancia del nodo 'i' a todos los demas, por eso queremos ver si en esta modificacion obtuvimos un ciclo de arbitraje. Al encontrar dicho ciclo pasamos a reconstruir el mismo, en caso de que no se encuentre, el algortimo habra iterado los tres ciclos por completo. Como 'k', 'i' y 'j' iteran entre 0 y 'n', iterar los tres por completo nos presenta la complejidad $\mathcal{O}(n^3)$ del algoritmo.

Reconstruir el ciclo solo nos presenta complejidad $\mathcal{O}(n)$. El mismo recorre el camino formado en la matriz de sucesores. Tomamos como punto de partida el nodo $next[i][i]$. Obteniendo ese valor, que llamaremos 'suc', lo que queremos es pedir el siguiente del nodo $[suc][i]$, para reconstruir el camino desde el sucesor ('suc') hasta 'i' nuevamente.

Algorithm 10 arbitraje(Matriz cotizaciones, int n) res: vectorıint¿cicloArbitraje

```
1: vector < int > cicloArbitraje                                   ▷ O(n)
2: Matriz < int, int > sucesores ← getNextElementMatrix(n)         ▷ O(n^2)
3: for k = 0, n do                                                 ▷ O(n^3 + n)
4:     for i = 0, n do
5:         for j = 0, n do
6:             distanciaCalculada ← cotizaciones[i][k] * cotizaciones[k][j]
7:             if cotizaciones[i][j] < distanciaCalculada then
8:                 cotizaciones[i][j] ← distanciaCalculada
9:                 sucesores[i][j] ← sucesores[i][k]
10:            end if
11:        end for
12:        if cotizaciones[i][j] > 1 then
13:            return computeArbitrajeCycle(sucesores, i)
14:        end if
15:    end for
16: end for
17: return cicloArbitraje
```

Algorithm 11 getNextElementMatrix(int n) res: Matriz sucesores

```
1: Matriz < int, int > sucesores
2: for i = 0, n do                                                 ▷ O(n^2)
3:     for j = 0, n do
4:         sucesores[i][j] ← j
5:     end for
6: end for
7: return sucesores
```

Algorithm 12 computeArbitrajeCycle(Matriz sucesores, int i) res: Matriz sucesores

```
1: vector < int > cicloArbitraje
2: suc ← sucesores[i][i]
3: cicloArbitraje.push_back(i)
4: while suc ≠ i do                                                ▷ O(n)
5:     cicloArbitraje.push_back(suc)
6:     suc ← sucesores[r][i]
7: end while
8: cicloArbitraje.push_back(i)
9: return sucesores
```

2.5. Experimentación

Para la experimentación llevada a cabo, hay dos categorias principales. La primera platea experimentos “genericos”, entendidos como experimentos donde no hubo mucha lógica detrás de ellos. La segunda es la categoria de casos favorables para uno u otro algoritmo. Ambas categorias son explicadas en detalle en su respectiva sección.

En los dos casos, la cantidad de ‘n’ nodos era $10 \leq n \leq 800$, incrementando el valor de ‘n’ en 10 para cada salto. Se realizaron 5 ejecuciones para cada juego de datos y se tomo el promedio de las mismas para representarlo en el gráfico.

No se tomaron casos de ‘n’ mayor a 800 ya que el tiempo de ejecución de los mismos ponía en peligro la presentación del trabajo práctico.

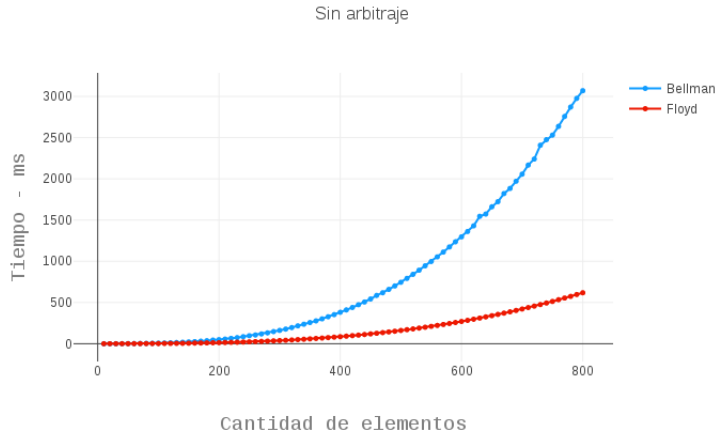
2.5.1. Experimento genérico

En el caso de los experimentos genericos, los mismos fueron desarrollados bajo los siguientes casos: una instancia sin arbitraje, otra con un único ciclo de arbitraje de longitud variable.

Tal como fue mencionado en la introducción, estos casos fueron diseñados sin mucha lógica detras. Qué significa esto? Significa que en el caso sin arbitraje solo se completó la matriz de tipos de cambio con elementos menores a 1 para cada 'i', 'j' en la matriz, excepto por la diagonal que contiene 1s. En el caso del único ciclo de longitud variable, se inicializa la matriz con todos valores cercanos a 0, excepto por la diagonal, y luego se cambia una cantidad aleatoria de los elementos cercanos a 0 a un valor un poco mayor a 1, por último, se cambia el valor de un nodo a 2.

2.5.1.1 Sin Arbitraje

Como fue explicado arriba, para este experimento se generó una matriz con valores menores que 1, para que no haya arbitraje. Dadas las complejidades temporales de los algoritmos, se espera en este caso que el tiempo de ejecución sea parecido para ambos algoritmos. Esto principalmente se espera así ya que Bellman-Ford, en cualquier escenario, debe terminar de recorrer el grafo para poder terminar la ejecución. En cambio, el algoritmo de Floyd interrumpe su ejecución al encontrar un ciclo que cumpla. Como no hay arbitraje, ambos algoritmos deben recorrer todos los puntos del grafo.

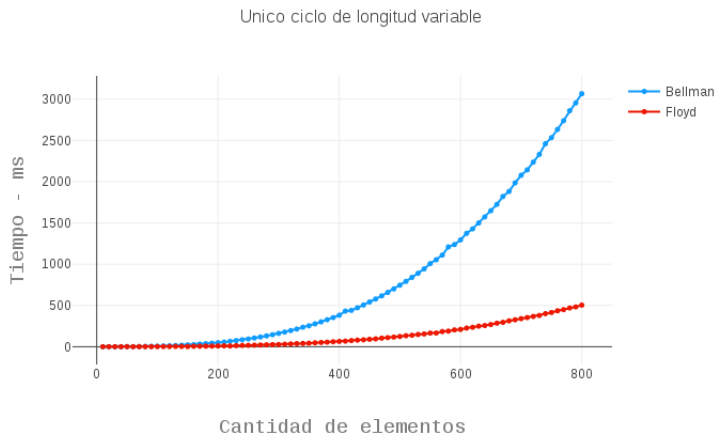


La ejecución no se comporta como fue esperado ya que la diferencia de tiempos entre los dos algoritmos es bastante amplia. Es probable que haya alguna implementación dentro del algoritmo de Bellman-Ford que esté causando la diferencia de tiempos. Sin embargo, si se puede observar que la curva es creciente en ambos casos, lo cual implica que siempre se fue cumpliendo con el peor caso.

2.5.1.2 Único Ciclo de Longitud Variable

Dado que en escencia, la matriz presentada en este caso es muy parecida al caso anterior, se espera una ejecución parecida. Recordamos que la diferencia con la matriz anterior es que esta contiene un ciclo de arbitraje.

Es por ello, que sabiendo que Floyd puede finalizar su ejecución antes si encuentra el ciclo de arbitraje, es posible que la ejecución del mismo sea más rapida que en el caso anterior.



Como era de esperar, la ejecución de ambos algoritmos es parecida al caso anterior, pero con una leve mejora en el caso de Floyd. Es importante tener en cuenta que esta mejora es leve porque existe un único ciclo que puede tener longitud ‘n’, lo cual implica que no necesariamente vaya a ser rapidamente encontrado por el algoritmo.

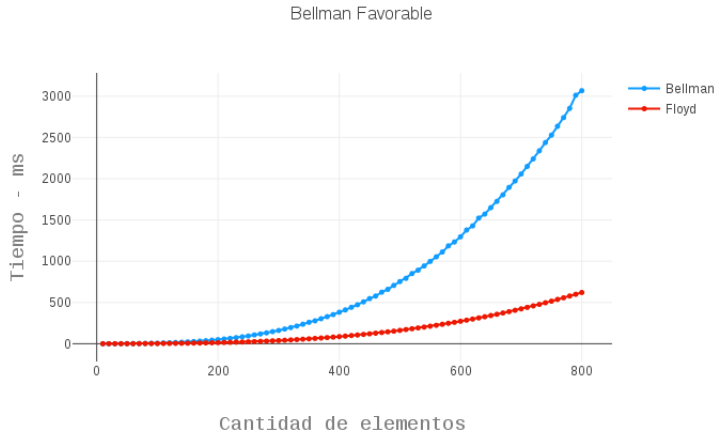
2.5.2. Experimento Favorables por Algoritmo

Para esta categoria se generaron dos instancias principales, una donde la matriz favorezca a Bellman-Ford y la otra que favorezca a Floyd-Warshall.

2.5.2.1 Favorable Bellman-Ford

El caso favorable a Bellman-Ford es aquel en donde no hay arbitraje y algoritmo pudo finalizar su ejecución antes de recorrer toda la matriz. Recordemos que Bellman-Ford finaliza su ejecución cuando termina de relajar todas sus aristas. Recordemos que el que haya arbitraje es equivalente a que haya un ciclo negativo, dicho ciclo se encontraría al iterar ‘n’ veces. Es por ello que como no hay arbitraje, es probable que no sea necesario iterar el grafo por completo, logrando así en teoria una mejor ejecucion temporal que Floyd-Warshall. Este ultimo algoritmo finaliza antes su ejecución si encuentra un ciclo negativo, que en este caso no existe.

Sin embargo, si tenemos en cuenta el resultado del primer experimento, el cual es muy parecido a este, es probable que la ejecución de Floyd-Warshall sea mejor que la de Bellman-Ford.

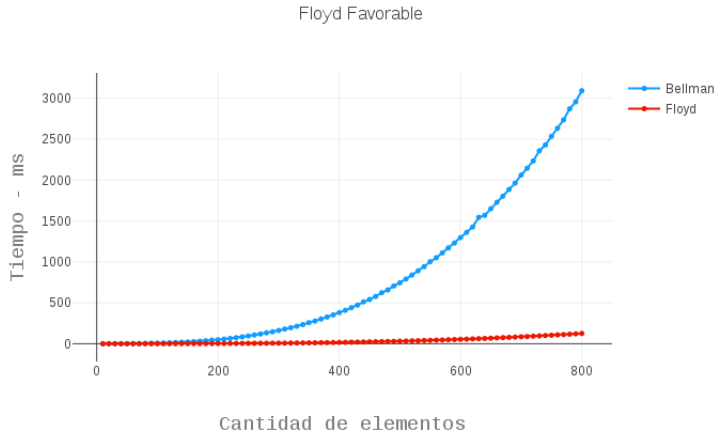


Tal como era de suponer, la ejecución de Floyd-Warshall es mucho mejor que la de Bellman-Ford. Esto llama la atención especialmente por lo mencionado arriba, donde este es un caso en el que Bellman-

Ford fue claramente favorecido. Nuevamente, es probablemente que haya alguna implementación en el algoritmo que no permita llegar a la complejidad temporal esperada.

2.5.2.2 Favorable Floyd-Warshall

Para favorecer la ejecución de Floyd-Warshall, se busco que las instancias a ejecutar contengan un ciclo de arbitraje y que el mismo se encuentre en la primera parte de la matriz, en particular entre los primeros n/4 elementos. De esta manera, es más visible la comparación contra Bellman-Ford dado que este debe ejecutarse por completo y Floyd-Warshall finalizará su ejecución al recorrer los primeros n/4 elementos.



Como se puede observar en el grafico, la diferencia de tiempos es muy grande. Recordamos que la ejecución de Bellman-Ford es probable que no sea consistente con su cota teorica, como dijimos en los experimentos anteriores. Sin embargo hay una gran diferencia entre la ejecución de uno y otro algoritmo.

2.6. Conclusiones

2.6.1. Analisis Comparativo entre los Algoritmos

La problematica de arbitraje fue resuelta con los algoritmos de Bellman-Ford y Floyd-Warshall como mostramos en las secciones anteriores. Sabemos que la complejidad de Bellman-Ford es $\mathcal{O}(nm)$. Si tenemos en cuenta que en este problema de arbitraje necesitamos el camino minimo entre todo par de nodos, debemos ejecutar el algoritmo otras 'n' veces, lo cual resulta en una complejidad de $\mathcal{O}(n^2m)$. Por otro lado, sabemos que la complejidad de Floyd-Warshall $\mathcal{O}(n^3)$.

Esto nos dice que Bellman-Ford no tiene un buen caso en grafos completos, que es el caso del arbitraje. Floyd-Warshall no es afectado por grafos completos, pero si por el tamaño del mismo. Esto tiene relevancia dado que la experimentación ejecutada mostraba al algoritmo de Floyd como un claro ganador en tiempo. Es cierto que la implementación de Bellman-Ford no esté en el $\%100$ de su efectividad, pero es de esperar que en un grafo completo, Floyd-Warshall se comporte mejor.

Con respecto a los casos donde puede convenir uno u otro, vimos que ambos pueden ser utilizados para detectar un ciclo negativo. Floyd-Warshall es superior si el arbitraje se encuentra entre los primeros nodos, Bellman-Ford si no lo hay.