DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Modelando problemas reales con grafos

Algoritmos y Estructuras de Datos III

Segundo Cuatrimestre de 2018

Integrante	LU	Correo electrónico
Bernardo Tuso	792/14	btuso.95@gmail.com
Celeste Rodriguez	639/16	cmrodriguez997@gmail.com
Facundo Linlaud	561/16	facundolinlaud@gmail.com
Philip Garrett	318/14	garrett.phg@gmail.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 a la problemática

Dada una lista de puntos distribuídos en un eje cartesiano, se nos pide encontrar una *clusterización* de los puntos válida. En este contexto, llamaremos una *clusterización* a una distribución de los nodos de un mapa en conjuntos. Cada conjunto albergará a todos los nodos cercanos entre sí, donde la cercanía es decidida a través de la evaluación de la distancia Euclidiana entre dos nodos y no pueden quedar nodos sin un conjunto asignado ni un nodo asignado a más de un conjunto.

En este documento se modelarán los clusters a través del uso de grafos. Específicamente, cada cluster corresponderá a una componente conexa. Es por esto que los árboles generadores mínimos serán un recurso muy importante a lo largo de este documento, 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 claramente si el largo de este eje en cuestión es demasiado grande o no en comparación con el resto de los ejes del grafo. Profundizaremos esta idea más adelante.

Por otro lado, la validez de una disposición depende del usuario. Es posible que existan múltiples distribuciones de nodos en *clusters* que resulten 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. Algoritmos propuestos para el cálculo del Árbol Generador Mínimo

Una manera de atacar el problema de la *clusterización* es a través de **árboles generadores mínimos**. Para hacer esto, 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. Existen varios algoritmos conocidos para calcular este tipo de árbol dado un grafo y en este proyecto utilizamos dos de ellos:

- El algoritmo de Kruskal (con y sin la variante Path Compression)
- El algoritmo de Prim

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. Este tema será profundizado en el futuro.

1.2. Implementaciones

1.2.1. Kruskal

Este es uno de los dos algoritmos utilizados para calcular el árbol generador mínimo de un grafo, que toma los siguientes parámetros de entrada:

- n: la cantidad de nodos en el grafo
- $edges$: todos los ejes del grafo con sus respectivos pesos

A continuación se muestra el pseudo-código del algoritmo:

Algorithm 1 Find Minimum Spanning Tree

```
1: function KRUSKAL(Entero points, Lista de ejes ejes)        ▷ O(n + m * log(n))
2:     agm = ∅

3:     Inicializar DisjointSetUnion
4:     Sort(ejes)
5:     for e ∈ ejes do
6:         if DisjointSetUnion.find(e.v) ≠ DisjointSetUnion.find(e.u) then
7:             agm ← agm ∪ e
8:             DisjointSetUnion.union(e.v, e.u)
9:         end if
10:    end for

11:    return agm
12: end function
```

(*) La complejidad del algoritmo se asume utilizando Disjoint Set Union con Path Compression

(**) Sin Path Compression ni la optimización de *padres de vértices*, una operación **find** o **union** puede tomar $\mathcal{O}(n)$.

(***) Si bien $m = n * (n - 1)$ porque se trabaja con un grafo completo, podemos concluir que $\mathcal{O}(n + m * log(m)) = \mathcal{O}(n + m * log(n^2)) = \mathcal{O}(n + m * 2 * log(n)) = \mathcal{O}(n + m * log(n))$ y esta cota obtenida es más preferible pues es más ajustada.

En primer lugar, se procede a ordenar la lista $edges$ por el peso de los ejes y de manera creciente. Este ordenamiento dispone de una complejidad $\mathcal{O}(n * log(n))$. A continuación procederemos a construir el árbol generador mínimo, por lo tanto es necesario comenzar por los ejes de menor peso para poder asegurar la **condición mínima** de nuestro árbol generador. Los ejes de este último se guardarán en la lista $min\_spanning\_tree$. Ahora, un eje del grafo G formará parte del árbol generador mínimo si al agregarlo a $min\_spanning\_tree$, **no forma un ciclo** entre los ejes que ya estaban ahí. Esto levanta una nueva incógnita: ¿Cómo decidimos si un eje forma un ciclo con otros? Para resolver esta pregunta, se dispone de una estructura de datos llamada **Disjoint Set Union**.

1.2.1.1 Disjoint Set Union

El propósito de esta estructura es llevar la cuenta de a qué componentes conexas pertenece cada vértice en el grafo. En otras palabras, es una estructura que nos permite contabilizar conjuntos disjuntos de vértices a través de la designación de **representantes** en las componentes conexas. Esta estructura nos provee la siguiente interfaz:

- find(x): Dado un elemento x, retorna el representante del vértice. Es decir, un identificador de la componente conexa a la cual pertenece x.
- union(x, y): Une a x e y y por transitividad a las componentes conexas de ambos.

Además, esta estructura cuenta con una variante llamada **Path Compression** que consiste en recordar en un arreglo el representante de un nodo buscado a través de $find$ para poder accederlo en $\mathcal{O}(\alpha(m))$ en la próxima consulta, con α la función inversa de Ackermann.

1.2.2. Prim

Este algoritmo es el segundo utilizado para la generación de un árbol generador mínimo de un grafo. Existen dos tipos de implementaciones para el algoritmo, una que utiliza una cola de prioridad y la otra

no, en este caso se optó por la implementación con cola de prioridad. El mismo toma los siguientes parámetros de entrada:

- n: la cantidad de nodos en el grafo
- $ejes$: todos los ejes del grafo con sus respectivos pesos

A continuación se muestra el pseudo-código del algoritmo:

Algorithm 2 Find Minimum Spanning Tree

```
1:  function PRIM(Entero points, Lista de ejes ejes)                    ▷ O(n + m * log(n))
2:      for 0 ≤ i < points do
3:          _visitado[i] ← false
4:          _distancia[i] ← INF
5:          _padre[i] ← i
6:      end for

7:      p ← 0                          ▷ Nodo cualquiera del grafo, se toma el nodo 0 a modo de ejemplo
8:      _distancia[p] ← 0
9:      _padre[p] ← 0
10:     _colaMinPrioridad.push(p, _distancia[p])

11:     while _colaMinPrioridad ≠ ∅ do
12:         p ← _colaMinPrioridad.extraerMinimo()
13:         if ¬_visitado[p] then
14:             _visitado[p] = true
15:             for 0 ≤ i < points do
16:                 if i ≠ p ∧ ∃e ∈ ejes, e.u = p ∧ e.v = i then
17:                     if ¬_visitado[i] ∧ _distancia[i] > peso(p, i) then
18:                         _distancia[i] = peso(p, i)
19:                         _padre[i] = p
20:                         _colaMinPrioridad.push(i, _distancia[i])
21:                     end if
22:                 end if
23:             end for
24:         end if
25:     end while

26:     return _padre
27: end function
```

En primer lugar, inicializan tres arreglos de longitud n, representando cada posición en cada arreglo información sobre cada punto, lo que conlleva una complejidad de $\mathcal{O}(n)$. El arreglo booleano $\_visitado$ indica si cada nodo fue visitado o no. El arreglo $\_distancia$ indica la distancia de cada nodo al subárbol ya explorado por Prim; para cada posición, su valor indicará la longitud (o peso) de menor costo conocido desde el subárbol explorado hasta el nodo i. Por último, el arreglo $\_padre$ permite la reconstrucción del árbol generador mínimo, el eje representado entre i y $\_padre[i]$ es el eje de menor peso explorado, por lo tanto mantiene la propiedad de ser mínimo.

¿Cómo assegurar que no se forma un ciclo en alguna iteración? Sea p un nodo perteneciente a un grafo cualquiera G. Al comenzar a recorrer la cola de prioridad (línea 11), primero se pregunta si p fue visitado previamente. En caso afirmativo, se ignora el nodo y se prosigue con el siguiente. Caso contrario, primero se marca al nodo como visitado, luego se buscan todos los nodos adyacentes al mismo. Sea i un nodo adyacente a p, si i fue visitado no se realiza ninguna acción sobre él. De esta manera, se evita el caso de generar dos caminos simples que conduzcan a i, manteniendo así la propiedad de árbol en G. Si no el nodo i no fue visitado, se evalúa si la distancia entre i y p es menor a la distancia del AGM al nodo

i almacenada. Si así lo fuera, se actualiza la distancia del AGM al nodo i, se marca como padre del nodo i a p y se agrega al nodo i a la cola de prioridad para explorar sus nodos cercanos. Al recorrer los ejes del grafo G y usar la cola mínima de prioridad, cuyas operaciones son logarítmicas, para formar el AGM, la complejidad final es de $\mathcal{O}(mlog(n))$

1.2.2.1 Recorte de ejes

El algoritmo de recorte recibe cuatro parámetros:

- Un árbol generador mínimo estructurado en listas de adyacencia
- Un valor σ 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 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 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 la definición de consistencia de Zahn *(Graph-Theoretical Methods for Detecting and Describing Gestalt Clusters, 1971:72)*.

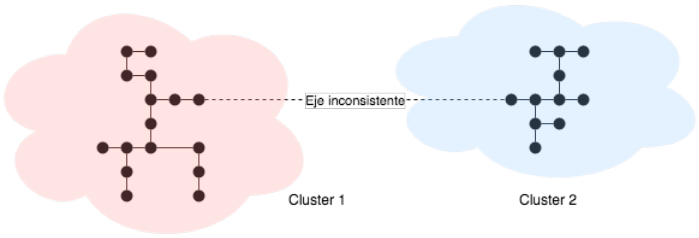


Figura 1: Un arbol generador mínimo donde se pueden reconocer dos clusters conectados a través de un eje inconsistente (marcado en rojo) y nueve ejes consistentes (marcados en verde).

En síntesis, dado que nos encontramos manipulando un árbol generador mínimo, eliminar un eje de esta estructura generaría una componente conexa nueva; es decir, separaría el grafo en dos representando 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** y el algoritmo las eliminará inmediatamente a través de la función **delete_edge**. A continuación se presenta el pseudo-código del algoritmo de recorte:

Algorithm 3 Algoritmo de recorte ▷ $\mathcal{O}(|E| * (|V| + \triangle(grafo)))$

```
1: function PRUNE_EDGES(Lista de adyacencia grafo, Punto flotante σ, Punto flotante f, Entero d)
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
```

1.2.2.2 Determinación de la consistencia de un eje

Este algoritmo tiene la función de, dado un eje perteneciente a una lista de adyacencias, la lista en sí y tres parámetros de exploración, determinar si la arista en cuestión es consistente. En otras palabras, si debe o no ser removida del grafo para generar un nuevo cluster.

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.

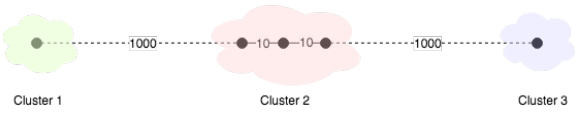


Figura 2: Recorte correcto llevado a cabo por las condiciones **(a)**, **(b)** y **(c)**.

Figura 3: Recorte incorrecto llevado a cabo solo por las condiciones **(a)** y **(b)**.

Se puede apreciar como en la figura 2 el algoritmo hace buen uso de la tercera condición para detectar la inconsistencia de los dos ejes de peso 1000 a diferencia de la figura 3, donde no se realiza una buena categorización sobre la consistencia de los ejes anormales. A continuación se presenta el pseudo-código de la función en cuestión:

Algorithm 4 Determinación de la consistencia de un eje ▷ $\mathcal{O}(|grafo|)$

1: **function** EDGE_IS_INCONSISTENT(Lista de adyacencia $grafo$, Eje e, Punto flotante σ, Punto flotante f, Entero d)
2: **Conjunto** $v\_pesos \leftarrow$ **recolectar_pesos**$(grafo, e.v, e.u, d)$ ▷ $\Theta(|V| + |E|)$
3: **Conjunto** $u\_pesos \leftarrow$ **recolectar_pesos**$(grafo, e.v, e.u, d)$ ▷ $\Theta(|V| + |E|)$

4: **Punto flotante** $v\_promedio \leftarrow$ **calcular_promedio**(v_pesos) ▷ $\mathcal{O}(|V|)$
5: **Punto flotante** $u\_promedio \leftarrow$ **calcular_promedio**(u_pesos) ▷ $\mathcal{O}(|V|)$

6: **Punto flotante** $v\_stdev \leftarrow$ **calcular_stdev**(v_pesos) ▷ $\mathcal{O}(|V|)$
7: **Punto flotante** $u\_stdev \leftarrow$ **calcular_stdev**(u_pesos) ▷ $\mathcal{O}(|V|)$

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

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

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

1.2.2.3 Recolector 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 5 Recolección de ejes de un vecindario ▷ $\Theta(|V| + |E|)$

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

5:     Conjunto de puntos flotante pesos ← ∅
6:     for Entero hijo ← 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 ∪ collect_edge_weights(grafo, siguiente, nodo, d − 1)
13:            end if
14:        end if
15:    end for

16:    return pesos
17: end function
```

1.2.2.4 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 6 Remoción de eje ▷ $\mathcal{O}(|ejes|)$

```
1: function DELETE_EDGE(Lista de ejes 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
```

1.2.2.5 Algoritmos de promedio y desviación estándar

Estos algoritmos son utilizados a la hora de calcular efectivamente el promedio y la desviación estándar del vecindario de un vértice v calculado mediante el algoritmo de **recolección de ejes** previamente explicado. Las complejidades de estos algoritmos son lineales en el tamaño de los vecindarios. Desde un punto de vista más macro, 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. Es sencillo entonces concluir que, 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)

Por lo tanto 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 7 Cálculo de desviación estándar ▷ $\Theta(|pesos|)$

```
1: function CALCULATE_STDEV(Lista de puntos flotantes pesos)
2:     if pesos es vacío then
3:         return 0
4:     end if

5:     Punto flotante promedio ← calcular_promedio(pesos)          ▷ Θ(|pesos|)
6:     Punto flotante varianza ← 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 8 Cálculo de promedio ▷ $\Theta(|poblacion|)$

```
1: function CALCULATE_MEAN(Lista de puntos flotantes poblacion)
2:     if poblacion es vacío then
3:         return 0
4:     end if

5:     Punto flotante promedio ← 0

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

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

1.2.3. Algoritmo de clusterización

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

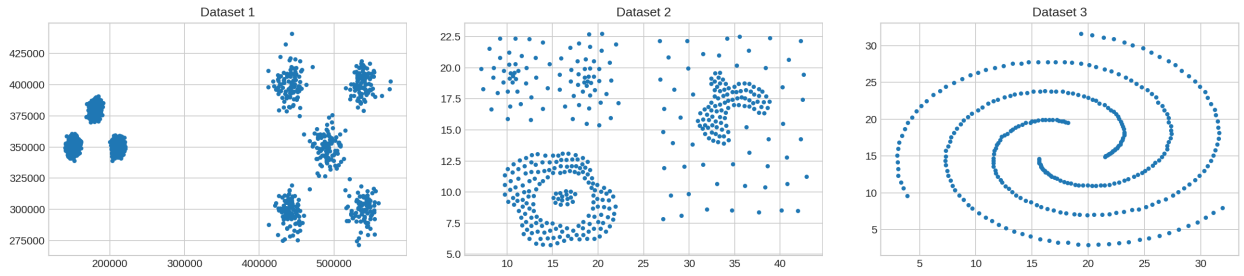
1.3.1. Criterios para evaluación de ejes

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

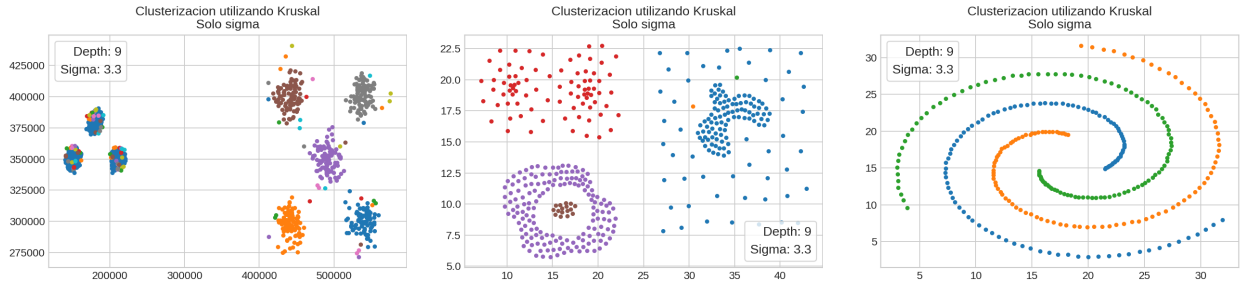
Para realizar las comparaciones vamos clusterizar los siguientes 3 grafos, definiendo como mas optimo al criterio que mejor arme clusters en todos los escenarios utilizando la menor cantidad de esfuerzo en elección de parámetros posible.



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

1.3.1.1 Solo desvío estándar

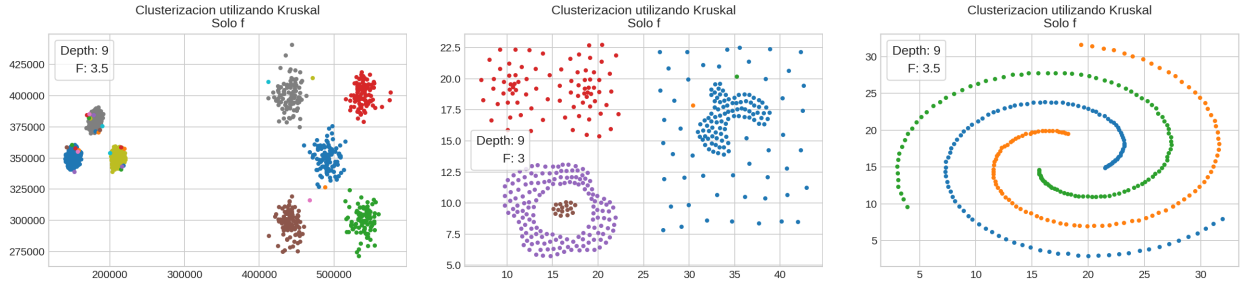
En este caso, vamos a descartar únicamente los ejes que son declarados inconsistentes al exceder en más de σ veces el desvío estándar aplicado sobre el tamaño de eje promedio del vecindario.



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

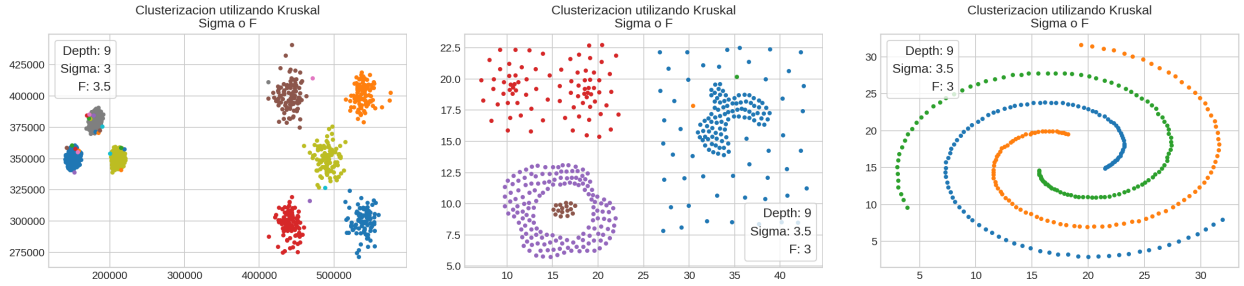
En este caso, vamos a descartar únicamente los ejes que son declarados inconsistentes al ser F veces más grandes que el tamaño de eje promedio del vecindario.



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

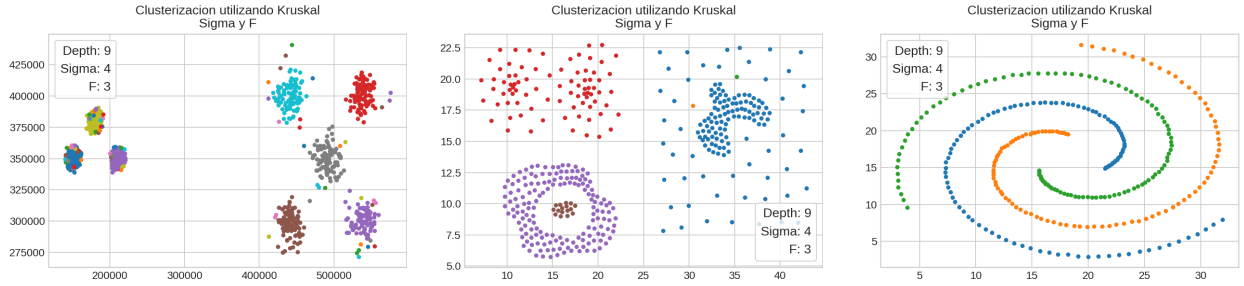
Para este caso, los ejes inconsistentes van a ser los que cumplan con alguno de los dos criterios anteriores



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.

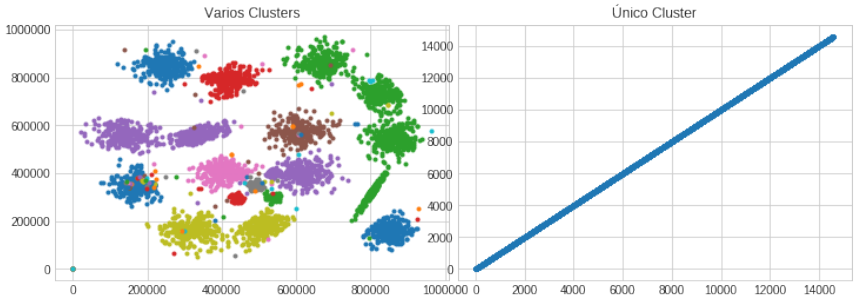
1.3.1.5 Conclusiones

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 Path Compression

Tal como fue presentado en la implementación del algoritmo de Kruskal, el mismo cuenta con una variante conocida como **Path Compression** que consiste en recordar el representante de un nodo buscado para poder accederlo en $\mathcal{O}(1)$ en la próxima consulta.

Es por ello, que surge entonces la inquietud de comparar el desempeño del algoritmo en ambos casos. Para poder llevar a cabo dicha comparación, los escenarios planteados son representados con los siguientes grafos con 14600 puntos cada uno:



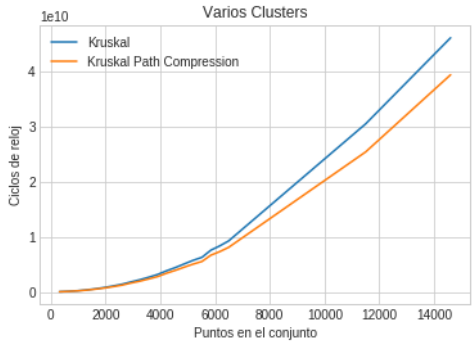
> **Nota**: Los puntos de la curva representan el promedio de ciclos de reloj de 10 ejecuciones por punto. Para cada grafo, desde 0 hasta 6500 se generaron datos en intervalos de 325 puntos, luego se genero información para 11500 puntos y por último para 14600 puntos.

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

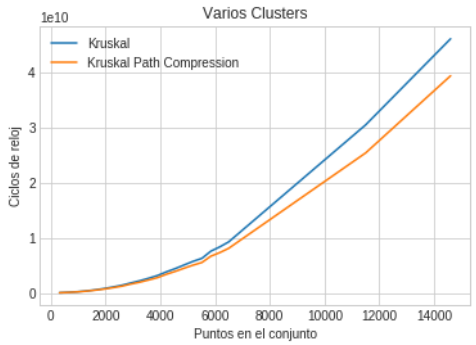
Al utilizar un grafo de 14600 puntos repartido en varios clusters como se pudo observar en la figura, se puede ver que cada cluster no contiene muchos puntos. Si bien, la alternativa de kruskal con path compression permite acceder al representante de un cluster en $\mathcal{O}(1)$, al tratarse de clusters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

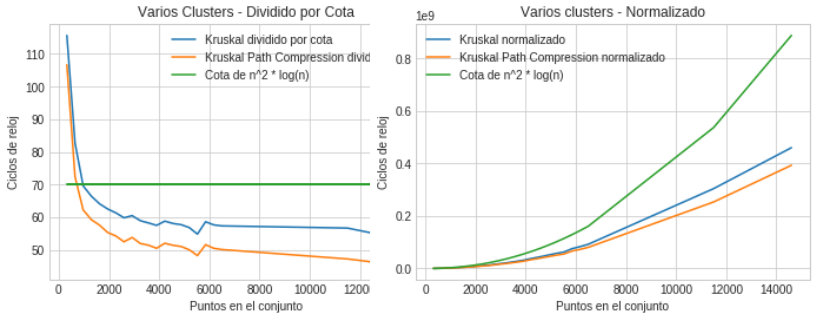
Al utilizar el grafo de 14600 puntos repartido en un único cluster se espera que la optimización de path compression sea más visible y haya una mayor diferencia en la ejecución de cada algoritmo.



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

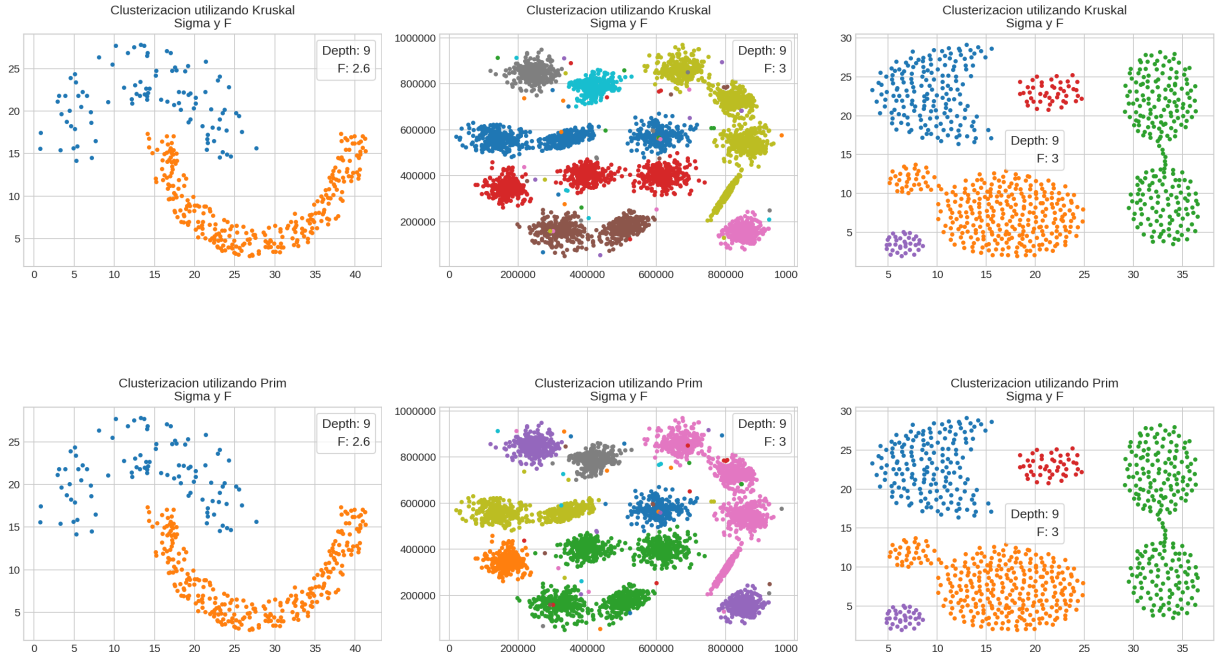
La complejidad de ambas implementaciones es la misma, por eso se espera que ante la cota tomada, ambos pasen a estar por debajo de la misma a partir de un cierto valor.



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

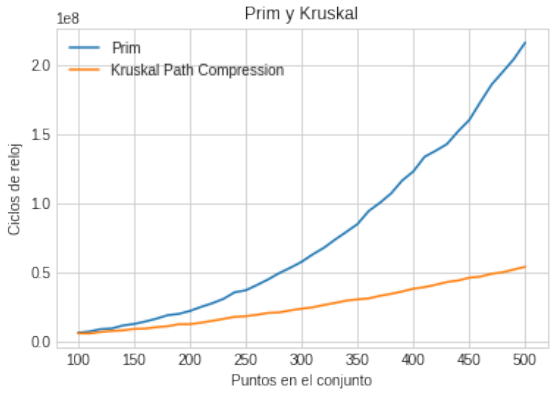
1.3.3. Comparacion de Kruskal contra Prim

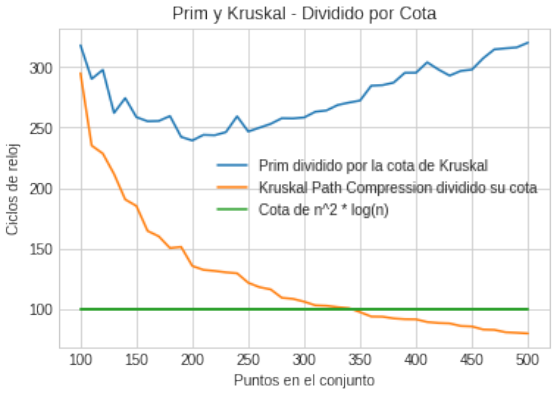
Al comparar Prim y Kruskal a la hora de clusterizar, podemos ver que ambos producen resultados similares. La diferencia entre ambos existe a la hora de generar el árbol generador mínimo, sobre el cual luego se descartan los ejes inconsistentes.



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 tomar ventaja del desequilibrio de precios en diferentes mercados comprando un recurso para luego venderlo y así generar una ganancia. Supongamos que existe un recurso R disponible para la compra y venta en dos distintos mercados, M1 y M2.

En M1, se puede comprar R por 42 pesos y vender por 41.8.

En M2, 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 M1 se entera de este suceso: actualizará su cotización. De esta forma, ahora M1 compra por 42,8 pesos el recurso y lo vende por 43.5. Hay oportunidad de arbitraje ya que M2 no actualizó sus precios: por cada unidad comprada en M2 y vendida en M1 se obtiene 42,8-42,5 = 0,3 pesos.

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, reales..). Para ello, contamos con la tasa de cambio para cada par. 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 $n-1$.

Entrada de ejemplo	Posible salida
4 1 0.1 5 0.125 10 1 0.5 $0.\overline{3}$ 0.2 2 1 2 8 3 0.5 1	2 3 1 2

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, un ciclo es $\{2, 3, 1, 2\}$ ya que c_{23} x c_{31} x c_{12} = 2 x 3 x 0.5 = 3 > 1.

2.2. Análisis del problema

2.2.1. Traduciendo a grafos

En primer lugar, transladaremos el problema al contexto de grafos de la materia. Supongamos que tenemos al grafo dirigido $G = (V, E)$ y la función p tal que p : $E \to \mathbb{R}$ con pesos en las aristas. Además, llamaremos al eje $e = \{i, j\}$ como $e_{i,j}$.

- V representará al conjunto de divisas disponibles por lo que es de tamaño n
- El peso de cada arista en E representará la tasa cambiaria entre cada par de monedas. Es decir, $\forall e \in E \wedge e = \{i, j\} \Rightarrow p(e) = c_{i,j}$

En consecuencia, encontrar la secuencia S mencionada anteriormente es encontrar S tal que

$$S \subset V \wedge S = \{d_a, d_b, d_c, ..., d_k\} \wedge p(e_{a,b}) \text{ x } p(e_{b,c}) \text{ x ... x } p(e_{k,a}) > 1$$

Cabe aclarar que el grafo es completo. Como entre todo par de divisas existe un multiplicador para transformar de una a la otra, en la afirmación anterior no hace falta mencionar que la arista tiene que existir para poder aplicar la función p (cuyo dominio es E). Para esto, definiremos a nuestro grafo incluyendo el eje $e_{i,i}$ cuyo peso será 1 para toda moneda i ya que para convertir de i a i hay que multiplicar por 1.

Por último, concluiremos que la secuencia buscada se condice con un circuito (ya que empieza y termina en el mismo nodo) que cumple la propiedad que al multiplicar el peso de sus aristas, da mayor que uno. Por lo tanto buscar S es buscar $C = \{e_{a,b}, e_{b,c}, ..., e_{k,a}\}$.

2.2.2. Buscando el ciclo negativo

Para poder resolver el problema con Bellman-Ford o Floyd, debimos hacer algunas modificaciones. Para ello, analizamos a que podemos aspirar al aplicar estos dos algoritmos. Destacamos las dos motivaciones más importantes:

- Encontrar un camino mínimo en un grafo con pesos en sus aristas, los cuales pueden ser negativos.
- Detectar ciclos negativos.

En consecuencia, procederemos a convertir el problema de arbitraje en encontrar un ciclo negativo. Es decir, encontrar un circuito cuya suma del peso total sea negativa. En principio el circuito C que buscamos cumple:

$$p(e_{a,b}) \text{ x } p(e_{b,c}) \text{ x ... x } p(e_{k,a}) > 1$$

Debemos buscar una función que modifique el peso para ajustar el problema. Para ello, aplicaremos $\log_{10}$ cuyo dominio es $\mathbb{R}_+ - \{0\}$. Como el multiplicador no puede ser cero y son todos positivos, todos los pesos están en el dominio. Además, al elegir la base del logaritmo positiva, la función es creciente y no altera la desigualdad al ser aplicada:

$$\log_{10}(p(e_{a,b}) \text{ x } p(e_{b,c}) \text{ x ... x } p(e_{k,a})) > \log_{10}(1)$$

Recordando la propiedad de los logaritmos: $log_{10}(a \text{ x } b \text{ x ... x } k) = log_{10}(a) + log_{10}(b) + ... + log_{10}(k)$ y siendo $log_{10}(1) = 0$ entonces:

$$\log_{10}(p(e_{a,b})) + \log_{10}(p(e_{b,c})) + \ ... \ + \log_{10}(p(e_{k,a})) > 0$$

Multiplicando por -1 ambos lados, se da vuelta la desigualdad:

$$+(-\log_{10}(p(e_{a,b}))) + (-\log_{10}(p(e_{b,c}))) + \ ... \ + (-\log_{10}(p(e_{k,a}))) < 0$$

Definiendo la función p' : $E \to \mathbb{R}$ como $p'(e) = -\log_{10}(p(e))$ entonces:

$$p'(e_{a,b}) \ + \ p'(e_{b,c}) + \ ... \ + p'(e_{k,a}) < 0$$

Finalmente, esto se condice con buscar un ciclo de aristas cuya suma de pesos sea menor a cero, es decir, un ciclo negativo. En este nuevo grafo con los pesos actualizados mediante la función p' ahora sí podemos aplicar los algoritmos de camino mínimo que detectan estos ciclos como Bellman o Floyd.

2.3. Bellman-Ford

2.3.0.1 Resolución

Para aplicar Bellman, utilizaremos la representación de grafos como vector de aristas. Cada una será a su vez representada mediante el struct $Edge$ de tres componentes: dos enteros (el nodo v será el inicio y u el final) y un peso.

Como el grafo es completo y estamos incluyendo las aristas (i, i) , para un grafo de n nodos habrá n^2 ejes.

Además cabe mencionar que hay que asegurarse que el ciclo negativo sea alcanzable desde el nodo inicial elegido: como el grafo es completo, será alcanzable desde cualquier nodo seleccionado. Si no lo fuera, habría que agregar un nodo inicial nuevo conectado a todos los demás a través de una arista de peso igual a uno y correr el algoritmo con este nuevo grafo y con el nuevo inicial.

Bellman actualizará en cada una de sus $n - 1$ iteraciones el vector de distancias $distancias$. En cada paso k, su invariante será:

$$d_k(i) = \text{camino mínimo del nodo inicial a i que usa a lo sumo k aristas}$$

Por lo tanto, en la última iteración obtendremos los caminos mínimos que usan a lo sumo $n-1$ aristas. Si no hay ciclos negativos, serán los mínimos posibles ya que a lo sumo cada uno pasa por todos los nodos del grafo una vez. Entonces, si al realizar una iteración más logro mejorar alguna distancia, la noción de camino mínimo se indefine y significa que encontré un ciclo negativo.

2.3.0.2 Implementación

Algorithm 9 Busqueda de ciclos negativos con Bellman

```
1: function BELMAN(vector<Edge> aristas, Entero n, Entero inicial)                ▷ O(n^3)
2:     actualizar_pesos(aristas)
3:     vector<Enteros> predecesores(n, null)                                      ▷ O(n^2)
4:     vector<Reales> distancias(n, INF)                                          ▷ O(n^2)
5:     distancias[inicial] ← 0

6:     Entero i ← 1
7:     Bool sigueMejorando ← true
8:     while i < n ∧ sigueMejorando do                                            ▷ O(n^3)
9:         sigueMejorando ← false
10:        for e ∈ aristas do
11:            if distancias[e.u] > distancias[e.v] + e.distance then
12:                distancias[e.u] ← distancias[e.v] + e.distance
13:                predecesores[e.u] ← e.v
14:                sigueMejorando ← true
15:            end if
16:        end for
17:    end while

18:    if ¬sigueMejorando then
19:        return ∅
20:    end if

21:    for e ∈ aristas do                                                         ▷ O(n^2 + n) = O(n^2)
22:        if distancias[e.u] > distancias[e.v] + e.distance then
23:            return reconstruir_ciclo(e, predecesores)
24:        end if
25:    end for
26:    return ∅
27: end function
```

Inicialización

- El primer paso será realizar la transformación de los pesos de las aristas como explicamos en la sección 2.2.2 mediante 10.
- $predecesores$ será inicializado con $null$: para cada posición i representará desde que nodo se llega a i en el camino actual.
- $distancias$ será inicializado con infinito para todos los nodos excepto para el inicial ya que la distancia de un nodo a sí mismo es cero. Como es completo, hay arista entre el inicial y cualquier nodo: su peso será mayor o igual al del camino mínimo resultante. Por eso, si elegimos infinito como la arista de mayor peso del grafo, seguro será mayor o igual que cualquier posible camino mínimo.
- Tendremos un booleano $sigueMejorando$ que chequeará en cada iteración si alguna distancia mejoró.

Algorithm 10 Actualizar pesos

```
1: function ACTUALIZAR_PESOS(vector<Edge> aristas)
2:     for e ∈ aristas do
3:         e.distance ← −log10(e.distance)
4:     end for
5: end function
```

Ciclo

- Por cada una de las aristas $e = (v, u)$ chequeamos si esta mejora la distancia hasta u o no. En caso afirmativo, la nueva distancia hasta u será la de v mas el peso de la arista. Como ahora se alcanza a u a partir de v, el nuevo predecesor de u será v.
- Cuando en una iteración no se mejore ninguna distancia, $sigueMejorando$ lo detectará y el ciclo terminará.

En el caso en que las distancias dejen de mejorar, seguro que no hay ciclo negativo por lo explicado en 2.3.0.1. De ser el caso, retorna vector vacío.

Recontrucción ciclo negativo

Si llegamos a este punto, tendremos dos casos:

- Encontrar los caminos mínimos tomó exactamente $n - 1$ iteraciones. Por lo tanto, no hay ciclos negativos y al realizar una iteración más no seguirá mejorando. Retornaremos vector vacío.
- Al realizar otra iteración, descubriremos que las distancias mejoran y a través del vector de $predecesores$ reconstruiremos el ciclo buscado.

Algorithm 11 Reconstrucción del ciclo en Bellman

```
1: function RECONSTRUIR_CICLO(Edge arista, vector<Entero> predecesores)        ▷ O(n)
2:     vector<Reales> res ← ∅
3:     Entero v ← arista.v
4:     agregar_atras(res,arista.u)
5:     agregar_atras(res,v)

6:     while v no este repetido en res do
7:         v ← predecesores[v]
8:         agregar_atras(res,v)
9:     end while

10:    borrar_hasta(res, v)
11:    reverse(res)

12:    return res
13: end function
```

Como encontramos a una $arista = (v, u)$ que mejora una distancia, recorriendo a través de los predecesores del origen de la arista volveremos al mismo u o nos toparemos con otro ciclo. En consecuencia, la idea es ir agregando los predecesores a res hasta haber añadido al mismo nodo dos veces. Al finalizar, tendremos el ciclo desde el final del vector orientado hacia la primera aparición del nodo que está repetido. Por lo tanto, eliminaremos todos los nodos hasta su primera aparición y daremos vuelta el vector (en el código son funciones que nos provee la std mediante iteradores).

2.3.0.3 Análisis de complejidad

Espacial

- $predecesores$ y $distancias$, ambos de tamaño n
- En **reconstruir_ciclo** se van agregando nodos hasta que hay un nodo que aparece dos veces. En consecuencia, a lo sumo corta cuando agrega todos los nodos: $n + 1$ (el inicial aparece dos veces).

En conclusión, la complejidad espacial es $\mathcal{O}(n)$.

Temporal

En primer lugar, **actualizar_pesos** recorre los n^2 ejes aplicandole a los respectivos pesos la función logaritmo. Si asumo que su complejidad pertenece a $\mathcal{O}(n)$, entonces la complejidad total de **actualizar_pesos** seguro que está acotada burdamente por $\mathcal{O}(n^3)$. Incluso, hay procesadores que pueden computar el logaritmo en una sóla instrucción.

En segundo lugar, contamos con las inicializaciones de los vectores en $\mathcal{O}(n)$.

Luego, el ciclo hace $n - 1$ iteraciones y en cada una recorre el vector de n^2 aristas, realizando operaciones en tiempo constante. Si bien puede o no cortar antes, a priori la complejidad total del ciclo es $\mathcal{O}(n^3)$.

Por último recorre una vez más los ejes y cuando encuentra una arista que mejora la distancia, reconstruye el ciclo. Otra vez recorrer $aristas$ es $\mathcal{O}(n^2)$, mientras que reconstruir el ciclo pernece a $\mathcal{O}(n)$. Esta última función auxiliar realiza todas operaciones sobre un vector de a lo sumo tamaño n. Hacer un $reverse$ sobre este pertenece a $\mathcal{O}(n/2) = \mathcal{O}(n)$. Asimísmo, para **borrar_hasta** se utilizó la función $erase$ que es lineal en la cantidad de elementos eliminados mas la cantidad de elementos que deben ser movidos, por lo tanto también pernece a $\mathcal{O}(n)$.

En conclusión, la complejidad temporal total del algoritmo es $\mathcal{O}(n^3)$.

2.4. Floyd-Warshall

2.4.0.1 Resolución

Para aplicar Floyd, utilizaremos la representación de grafos como matriz de pesos. Cada posición $[i][j]$ representará el peso de la arista que va de i a j. Tendrá n^2 posiciones, lo cual se condice con la cantidad de ejes del grafo. Por otro lado, tendremos la matriz $distancias$ del mismo tamaño en la que cada posición $[i][j]$ representará la distancia del camino de i a j. Por último, tenemos la matriz $siguientes$ donde el nodo en la posición $[i][j]$ es el nodo siguiente a i en el camino que va desde i hasta j.

Floyd actualizará en cada una de sus n iteraciones el vector de distancias. Sea $distancias^k$ la matriz del paso k (como k va desde 0 hasta $n - 1$ con -1 representaremos la inicialización):

$$distancias^{-1}[i][i] = 0 \; y \; para \; i \neq j \; distancias^{-1}[i][j] = p(e_{i,j})$$
$$distancias^{k+1}[i][j] = min(distancias^k[i][j], distancias^k[i][k+1] + distancias^k[k+1][j])$$

En cada paso k calcula el camino mínimo de i a j con vértices intermedios en el conjunto $\{0, 1, ..., k\}$. Por lo tanto, en la última iteración cuando $k = n - 1$ será calculado el camino mínimo con vértices intermedios en el conjunto $\{0, 1, ..., n - 1\}$, es decir, todos los nodos del grafo.

Como la distancia de un nodo a sí mismo es cero y se inicializa de esta forma, si $distancias[i][i]$ en algún paso es menor a cero, significa que hay un ciclo negativo que empieza y termina en el nodo i. Al encontrarlo, reconstruiremos el mismo a partir del camino en $siguientes$ y lo retornaremos.

2.4.0.2 Implementación

Algorithm 12 Busqueda de ciclos negativos con Floyd

```
1:  function FLOYD(Matriz<Reales> M, Entero n)                          ▷ O(n^3)
2:      Matriz<Reales> distancias(n^2, 0)                               ▷ O(n^2)
3:      Matriz<Enteros> siguientes(n^2, 0)                              ▷ O(n^2)
4:      actualizar_pesos(M, distancias)
5:      inicializar_siguientes(siguientes)                              ▷ O(n^2)

6:      for k ← 0 hasta n − 1 do                                        ▷ O(n^3 + n) = O(n^3)
7:          for i ← 0 hasta n − 1 do
8:              for j ← 0 hasta n − 1 do
9:                  if distancias[i][j] > distance[i][k] + distancias[k][j] then
10:                     distance[i][j] ← distance[i][k] + distancias[k][j]
11:                     siguientes[i][j] ← siguientes[i][k]
12:                 end if
13:             end for
14:             if distancias[i][i] < 0 then
15:                 return reconstruir_ciclo(i, siguientes)             ▷ O(n)
16:             end if
17:         end for
18:     end for

19:     return ∅
20: end function
```

Inicialización

- El primer paso será inicializar la matriz de $distancias$ con los pesos actualizados de las aristas del grafo mediante 13, según lo explicado en 2.2.2. Para cada posicion $[i][j]$ de la matriz como el eje $e_{i,j}$ existe, la distancia inicial de i a j será el peso de esa arista.
- De nuevo como para cualquier par de nodos del grafo la arista $e_{i,j}$ existe, el $siguientes$ de i en el camino que va a j es j. El pseudo-código en 14.

Algorithm 13 Actualizar pesos

```
1: function ACTUALIZAR_PESOS(Matriz<Reales> M, Matriz<Reales> distancias)
2:     for i ← 0 hasta n − 1 do
3:         for j ← 0 hasta n − 1 do
4:             distancias[i][j] ← −log10(M[i][j])
5:         end for
6:     end for
7: end function
```

Algorithm 14 Inicializar siguientes

```
1: function INICIALIZAR_SIGUIENTES(Matriz<Reales> siguientes)
2:     for i ← 0 hasta n − 1 do
3:         for j ← 0 hasta n − 1 do
4:             siguientes[i][j] ← j
5:         end for
6:     end for
7: end function
```

Ciclo

- En cada iteración k se chequea para cada par de nodos (i, j) si utilizar el nodo intermedio k mejora la distancia desde i hasta j. En caso afirmativo, la nueva distancia de i a j será la de i a k mas la de k a j. Como ahora se alcanza a j utilizando el nodo intermedio k, el nuevo siguiente de i en el camino a j será el mismo que el siguiente a i en el camino a k.
- En cada iteración, dado un nodo i luego de actualizar todos las distancias desde este al resto de los nodos, chequearemos por lo explicado en 2.4.0.1 si se mejoró el camino desde i a i. Por lo tanto, si la distancia pasó a ser negativa entonces encontré un primer ciclo negativo.

Si no se encontró ciclo negativo, el ciclo termina y retorna vector vacío.

Recontrucción ciclo negativo

Para reconstruirlo, utilizamos la función auxiliar **reconstruir_ciclo**. El ciclo será $\{primero, ..., primero\}$ donde el segundo vértice $nodo$ será el siguiente en el camino de $primero$ a $primero$, es decir, $siguientes[primero][primero]$. Asimísmo, el tercero será el siguiente en el camino desde $nodo$ hasta $primero$ y así hasta que $nodo = primero$.

Algorithm 15 Reconstrucción del ciclo en Floyd

```
1:  function RECONSTRUIR_CICLO(Entero primero, Matriz<Enteros> siguientes)        ▷ O(n)
2:      vector<Reales> res ← ∅
3:      Entero nodo ← siguientes[primero][primero]
4:      agregar_atras(res,primero)
5:      while nodo! = primero do
6:          agregar_atras(res,nodo)
7:          nodo ← siguientes[nodo][primero];
8:      end while
9:      agregar_atras(res,primero)

10:     return res
11: end function
```

2.4.0.3 Análisis de complejidad

Espacial

- matrices $siguientes$ y $distancias$, ambas de tamaño n^2
- En **reconstruir_ciclo** se van agregando nodos hasta llegar al primer nodo. En consecuencia, a lo sumo corta cuando agrega todos: $n + 1$ (el inicial aparece dos veces).

En conclusión, la complejidad espacial es $\mathcal{O}(n^2)$.

Temporal

En primer lugar, contamos con las inicializaciones de las matrices $siguientes$ y $distancias$ en $\mathcal{O}(n^2)$.

En segundo lugar, **actualizar_pesos** recorre la matriz de n^2 aplicando el logaritmo. Si asumo que su complejidad pertenece a $\mathcal{O}(n)$, entonces la complejidad total de **actualizar_pesos** seguro que está acotada por $\mathcal{O}(n^3)$.

Luego son tres fors anidados de n iteraciones $\implies$ $\mathcal{O}(n^3)$. Cuando no hay arbitraje el ciclo no corta antes. En caso de encontrar el ciclo, lo reconstruye y lo retorna. Como en **reconstruir_ciclo** sólo se realizan asignaciones, comparaciones y se agregan a lo sumo $n + 1$ nodos al vector resultado, la complejidad resulta ser $\mathcal{O}(n)$

En conclusión, la complejidad temporal del algoritmo es de $\mathcal{O}(n^2 + n^3 + n)$, es decir $\mathcal{O}(n^3)$.

2.5. Experimentación

2.5.1. Experimento genérico

El primer experimento que realizamos sobre los algoritmos para buscar ciclos negativos fue uno genérico. Para este, fuimos moviendo n desde 10 hasta 1400 y medimos los tiempos de ejecución de ambas implementaciones para las siguientes instancias:

- 5 instancias donde no hay arbitraje
- 5 instancias distintas donde hay un solo ciclo de longitud random (que también utiliza nodos random)
- 5 instancias donde puede o no haber arbitraje asignandole pesos random a las aristas

A cada una de ellas se la corrió 5 veces y se calculó la mediana para eliminar outliers.

Hipótesis

La idea es que al utilizar entradas variadas y como ambos algoritmos tienen la misma complejidad temporal, estos se comporten de manera similar. En general creemos que instancias donde hay arbitraje podría llegar a tener ventaja Floyd ya que Bellman realizará siempre las $n - 1$ iteraciones del ciclo (no puede cortar antes porque hay ciclo negativo). Sin embargo, si el ciclo está posicionado de manera que Floyd tarde en encontrarlo, no habrá demasiada diferencia entre ambas performances. Esto motivo al experimento 2.5.3. Por otro lado, también agregamos instancias donde no hay arbitraje ya que esto implica una ventaja para Bellman, el cual cuando termine de calcular los caminos mínimos, dejará de iterar.

Resultados

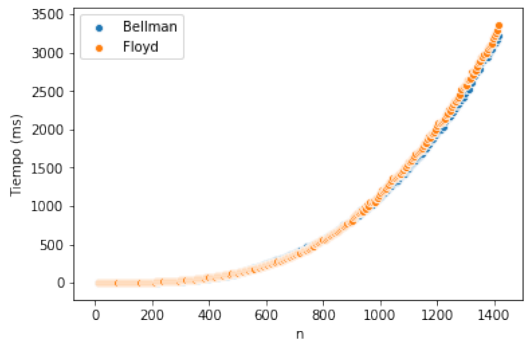


Figura 4: Tiempo consumido en función de una entrada de tamaño n

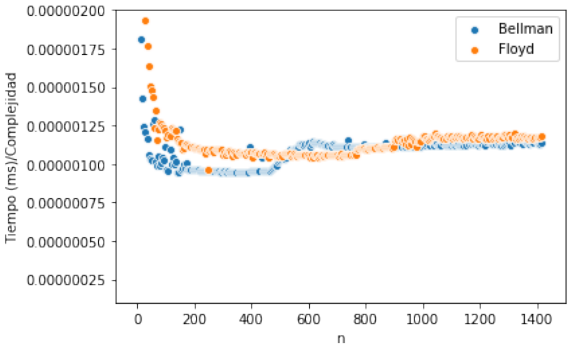


Figura 5: Tiempo consumido/complejidad esperada en función de una entrada de tamaño n

En la figura 4 podemos observar que efectivamente ambos algoritmos se comportan de manera similar. Sin embargo, para cierto rango de valores, Bellman le ganó a Floyd y viceversa. Para $450 \leq n \leq 850$ Floyd parece ser más rápido que Bellman mientras que para el resto de los valores, Bellman tuvo una mejor performance que Floyd.

En la figura 5 lo que hicimos fue dividir el tiempo de ejecución por la complejidad de los algoritmos, es decir n^3. Como la complejidad teórica de ambos es la misma, esperamos que quede graficada una constante. Efectivamente, aquí se observa que a partir de cierto n ambas funciones resultantes quedan acotadas por una constante. Si bien no es una demostración formal, es una prueba de que la complejidad es la esperada. De otra forma, los puntos no convergerían alrededor de un valor constante.

2.5.2. Variando la longitud del ciclo

Pareció interesante realizar una experimentación basada en ver como se comportan los algoritmos al aumentar la longitud del ciclo negativo. Para ello, elegimos cuatro tamaños de grafos para fijar e ir variando la longitud del ciclo para cada uno de ellos. Los valores fueron $\{300, 500, 800, 1000\}$ mientras que la longitud del ciclo para cada uno de los cuatro n fijos va desde 10 hasta n. El ciclo se arma de manera tal que sea el único del grafo, eligiendo vértices random. Para cada valor de longitud de ciclo, se crearon 5 instancias distintas y se corrió a cada una 5 veces. Para eliminar outliers, se tomó la mediana de estos.

Hipótesis

Como en todos estos casos hay arbitraje, Bellman-Ford no podrá cortar el ciclo antes por lo que sí o sí hace las primeras $n - 1$ iteraciones. Luego, sólo queda encontrar una arista que mejore la distancia y reconstruir el ciclo, todo en tiempo $\mathcal{O}(n^2)$. Por lo tanto, como sólo depende de encontrar esa última arista, mientras que reconstruirlo está acotado por la cantidad de nodos del grafo: esperamos que la longitud del ciclo no afecte el tiempo de ejecución del algoritmo, que se mantenga de cierta forma constante.

Por otro lado, tenemos el caso de Floyd. No creemos que su tiempo de ejecución dependa directamente de la longitud del ciclo mas si pensamos que depende de donde esté ubicado. Como en esta experimentación no manejamos la ubicación del ciclo ya que los vértices se eligen de forma random, esperamos que los tiempos sean variados pero que no aumente conforme aumenta la longitud del ciclo. Con respecto a su ubicación, hicimos una experimentación teniendola en cuenta en 2.5.3.

Resultados

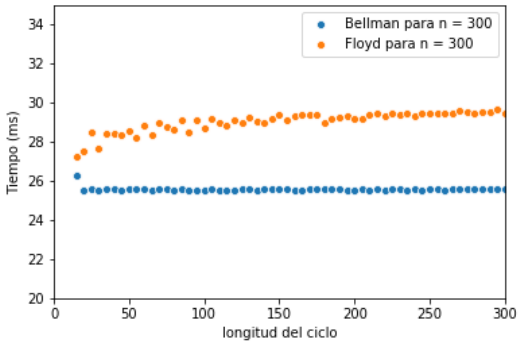


Figura 6: Tiempo consumido en función de la longitud del ciclo para n = 300

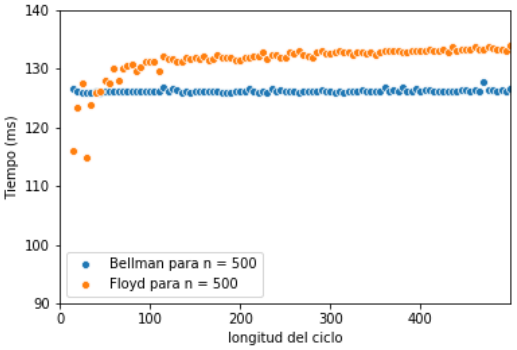


Figura 7: Tiempo consumido en función de la longitud del ciclo para n = 500

Entre las figuras 6 y 7 se observa un comportamiento similar para ambos algoritmos. Como era de esperarse, Bellman ronda los mismos valores conforme aumenta la longitud del ciclo, de esto podemos concluir que no depende ni de este ni de su ubicación. Con estos dos n, Floyd se comporta de manera similar que Bellman. Se observa que si bien crece en pequeña medida, converge.

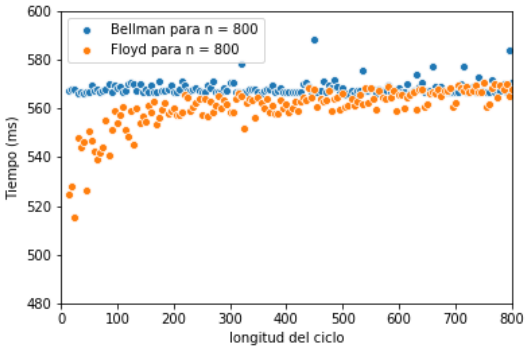


Figura 8: Tiempo consumido en función de la longitud del ciclo para n = 800

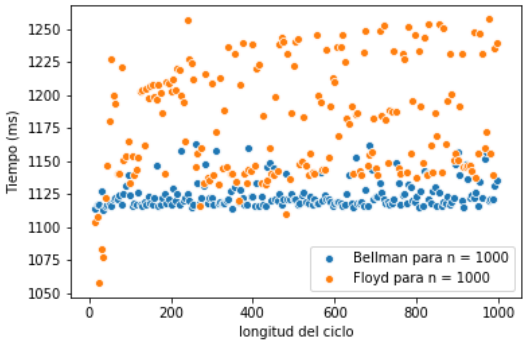


Figura 9: Tiempo consumido en función de la longitud del ciclo para n = 1000

Luego, al fijar n en 800 y en 1000 acá también se observa que Bellman no depende de la longitud del ciclo. Además, al igual que en 2.5.1, con $n = 800$ Floyd parece tener una mejor performance que Bellman.

En la figura 8 se ve una diferencia entre n menores a 200 y los mayores. Cuando es chico parece que al aumentar la longitud del ciclo aumenta el tiempo de ejecución mientras que cuando pasa los 200, tiende a converger como en el resto de los casos. Por último en 9 es donde mas variabilidad de resultados se encontró para Floyd: hay valores con longitud de ciclo chica que dieron lo mismo que instancias con longitud de ciclo grande. De nuevo, de todas formas todos los valores están acotados ya que la complejidad del algoritmo depende sólo de n.

2.5.3. Buscando buenas instancias

Como bien mencionamos en los experimentos anteriores, es interesante buscar buenas instancias para un algoritmo que no sean tan buenas para el otro. Las posibles buenas instancias para Bellman en principio son sin arbitraje: sin ciclos negativos. Bellman cortará el ciclo cuando termine de relajar todas las aristas: la idea es que no sean necesarias las $n - 1$ iteraciones (recordemos que de todas maneras puede hacer $n - 1$ iteraciones e igual no haber arbitraje). Un primer experimento contó con armar estas instancias sin arbitraje y comparar ambos algoritmos. Para estas, Floyd no encontrará que la diagonal de $distancias$ cambie, por lo que iterará n^3 veces siempre. Para este, creamos instancias con n variable entre 10 y 1000, donde no hay arbitraje y los pesos van desde 0.3 hasta 1.0.

Por otro lado, en cada paso k Floyd calcula el camino mínimo de i a j con vértices intermedios en el conjunto $\{0, ..., k\}$. Por lo tanto, si el ciclo está conformado por los primeros nodos, el k que incluya todos será menor y terminará más rápido. Además, como Bellman no corta antes no será un buen caso para este. La instancia elegida para el experimento fue para cada n variable entre 10 y 1000 crear un grafo de n nodos con exactamente un ciclo de $n/4$ formado por los primeros $n/4$ vértices. Para ambos experimentos correremos cada caso 5 veces y calcularemos la mediana para eliminar outliers.

Hipótesis

Como ya analizamos, esperamos que en el primer experimento Bellman tenga una mejor performance que Floyd y en el segundo al revés.

Resultados

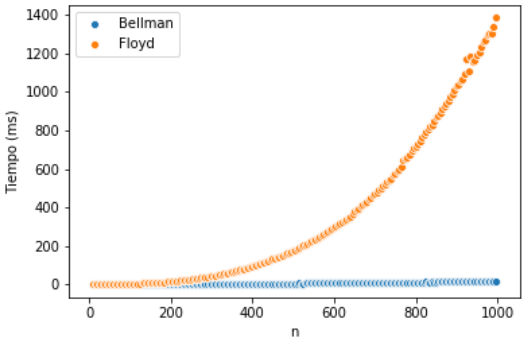


Figura 10: Tiempo consumido en función de una entrada de tamaño n

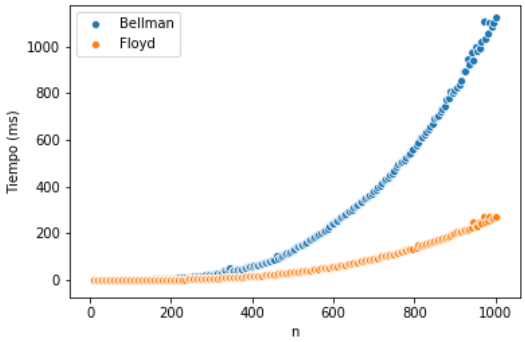


Figura 11: Tiempo consumido en función de una entrada de tamaño n

En la figura 10 se observa como elegimos buenas instancias para Bellman al punto de que comparado con Floyd, el tiempo de ejecución es ínfimo. Asimísmo, en 11 vemos que los data sets elegidos especialmente para que corran rápido con Floyd también fueron acertados. Podemos concluir que nuestras hipótesis efectivamente fueron correctas.

2.5.4. Variando el logaritmo

Como último, agregamos un mini experimento que consiste en comparar entre logaritmo natural, logaritmo base 10 y logaritmo base 2 el tiempo de ejecución de Bellman. La idea es ver si utilizar una u otra base es más o menos complejo. Si fueran enteros, esperaría que una implementación del logaritmo base 2 pueda ser más rápida que las otras. Sin embargo, estamos trabajando con reales (floats).

Para esta experimentación, corrimos Bellman con instancias donde hay arbitraje y donde no de tamaño n con este entre 5 y 300. Como en toda la experimentación, realizamos varias iteraciones y tomamos la mediana. En la figura 12 no se observa diferencia entre utilizar un logaritmo o el otro, excepto en unos pocos casos. De esto podemos concluir que cuando hablamos de $floats$ la base entre estas tres no parece importar.

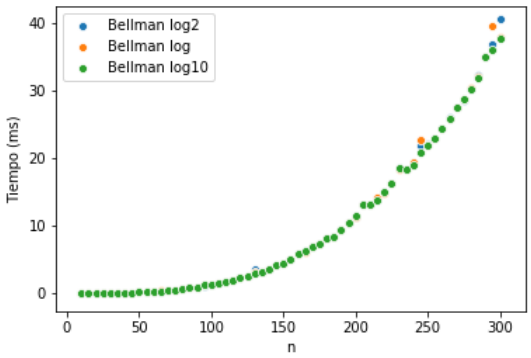


Figura 12: Tiempo consumido en función de una entrada de tamaño n

2.6. Conclusiones

Para resolver el problema de arbitraje de divisas debimos investigar los distintos algoritmos de camino mínimo conocidos. Entre los tres vistos en la teórica, rápidamente descartamos Dijkstra porque justamente hay aristas negativas: los dos restantes sí fueron funcionales para el problema a resolver. Entre los algoritmos de Bellman y Floyd, si bien ambos detectan ciclos negativos, observamos las siguientes diferencias:

1. Bellman-Ford es un algoritmo de single-source shortest path con complejidad $\mathcal{O}(nm)$. Para calcular el camino mínimo entre todo par de nodos, se debe correr el algoritmo n veces: una vez para cada nodo como posible source resultando en una complejidad $\mathcal{O}(n^2m)$. También hay otra alternativa que explicamos en 2.3.0.1 que consiste en agregar un nodo ficticio source conectado a todos los nodos del grafo, de manera que el peso de todas esas aristas agregadas sea cero (o uno, en el problema inicial de arbitraje). En la práctica demostramos que esta solución no agrega complejidad al algoritmo.

2. Floyd-Warshall es un algoritmo de all-pairs shortest path con complejidad $\mathcal{O}(n^3)$. Calcula los caminos mínimos entre todo par de nodos.

Lo que podemos concluir de estas diferencias es que Bellman depende de la cantidad de aristas en el grafo, mientras que Floyd no. Por lo tanto, cuando se trate de un grafo disperso Bellman tendrá mejor performance que cuando es denso. Por otro lado, a Floyd sólo le importa la cantidad de nodos: si el grafo es denso, la complejidad no se verá afectada pero sí si el grafo tiene muchos nodos. En consecuencia, según como esté conformado el grafo, dependerá que algoritmo conviene escoger.

En el caso de arbitraje, como el grafo es completo la cantidad de aristas es de orden n^2 por lo que ambos tendrán la misma complejidad y no podremos a priori decir que algoritmo tendrá mejor performance. Sin embargo, como buscamos ciclos negativos y ambos los encuentran de forma distinta, sí pudimos observar diferencias entre ambos rendimientos. Como vimos en los experimentos, encontramos buenas instancias para uno u otro algoritmo dependiendo si había arbitraje o no, de la ubicación del ciclo, etc.