DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 2

Clustering

25 de octubre de 2018 — Algoritmos y Estructura de Datos III

Integrante	LU	Correo electrónico
Buceta, Diego	001/17	diegobuceta35@gmail.com
Jiménez, Gabriel	407/17	gabrielnezzg@gmail.com
Springhart, Gonzalo	308/17	glspringhart@gmail.com

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2610 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (++54 +11) 4576-3300

http://www.exactas.uba.ar

1. Introducción al problema

En este informe informe vamos a ver distintos problemas cuyas posibles soluciones involucran árboles generadores mínimos y algoritmos que los generan. En particular los problemas son el **Clustering** y el **Arbitraje**.

2. Clustering

En el mundo del machine learning, problema de **Clustering** consiste en poder **agrupar datos** en distintos grupos llamados clusters, donde se espera que los datos de un cluster tengan algún tipo de **relación o característica** que los **distinga** de los datos en otros clusters. Este problema es de gran interés ya que tiene muchos usos como por ejemplo:

- Las tomografías por emisión de positrones utilizan análisis de clusters para poder diferenciar tejidos en imagenes tridimensionales.
- La investigación de mercados utilizan el clustering para poder particionar la población de consumidores en grupos para poder entender mejor la relación entre ellos.
- El clustering puede ser utilizado por motores de búsqueda de páginas web para poder agrupar resultados similares.
- La segmentación de imágenes utiliza clustering para poder detectar bordes u objetos en imágenes.

El análisis de Clusters es un problema de **aprendizaje no supervisado**, que puede ser abarcado de **varias maneras**, dependiendo de como uno **interprete que es lo que constituye un cluster** y como se puede encontrar de forma eficiente. Los diversos algortimos que resuelven el problema se pueden categorizar según su interpretación de lo que es cluster. Se puede reformular el problema de Clustering como un problema de **optimización multiobjetivo**, donde el algoritmo y los parámetros (por ejemplo la función para calcular distancias, o la cantidad esperada de clusters) relacionados a este dependen del conjunto de datos a analizar. El análisis de clusters es un **proceso iterativo** que involucra prueba y error, generalmente es necesario modificar parámetros hasta conseguir los resultados satisfactorios.

2.1. Dificultades del clustering, Aprendizaje Supervisado vs No Supervisado

Como se mencionó anteriormente el análisis de clusters es un problema de **aprendizaje no supervisado**, para poder entender como esto afecta al problema vamos a explicar de forma resumida los dos tipos principales de aprendizajes que existen en el mundo del machine learning.

Aprendizaje Supervisado

El Aprendizaje Supervisado es una técnica de machine learning que, a partir de conjuntos de entrenamiento de entradas y salidas esperadas, utiliza un algoritmo que intenta inferir una función que pueda predecir las salidas adecuadas para cualquier entrada nueva. Se llama supervisado ya que a medida que el algoritmo hace predicciones sobre los conjuntos de entrenamiento se van corrigiendo los datos de acuerdo a las salidas esperadas. Los problemas de aprendizaje supervisado se pueden agrupar en problemas de **clasificación**,

cuando las salidas son clases o categorías y problemas de **regresión**, cuando las salidas son valores reales.

Aprendizaje No Supervisado

A diferencia del aprendizaje supervisado, en el **aprendizaje no supervisado** no se sabe nada de la información de entrada **a priori**. A partir de un conjunto de datos de entrada se corren algoritmos que se encargan de descubrir estructuras o características en los datos. Como no hay salidas esperadas, la única forma de saber si los datos resultantes de correr los algoritmos son correctos es realizar evaluaciones sobre los resultados. Pese a esta desventaja, el aprendizaje no supervisado es útil cuando por ejemplo, no se sabe nada sobre los datos a estudiar, por lo que se puede usar un algoritmo de este tipo para obtener información sobre los mismos.

Entonces podemos ver que el problema de análisis de clusters cae en la segunda categoría de aprendizaje, esto lo hace un problema difícil, ya que no existe una solución çorrectaçon la que podríamos revisar los resultados obtenidos del los algoritmos de clustering, además por lo mencionado anteriormente, la definición de cluster en si es ambigua y dependiendo de lo que se decida definir como un cluster, los resultados obtenidos sobre un mismo conjunto de datos de entrada pueden ser muy diferentes.

2.2. Heuristica para resolver el problema

Para resolver el problema utilizamos un algoritmo basado en el método de detección de clustes explicado en el paper "Graph-Theoretical Methods for Detecting and Describing Gestalt Clusters"de Charles T. Zahn, y su forma de calcular los clusters es la siguiente: Consideramos nuestros datos como una serie de puntos en un plano, tomando un grafo completo G usando los puntos como vertices y las distancias entre ellos como pesos en las aristas. Luego calculamos el Árbol Generador Mínimo de G y lo llamamos T. Para poder encontrar los clusters hay que eliminar de T a los ejes **inconsistentes**, según el paper un eje e es inconsistente si su peso supera en cierta cantidad de desviaciones estandard a las medias de los pesos de las aristas que estén a distancia k de los vértices que están en sus puntas. Al remover un eje inconsistente, las componentes conexas formadas son identificadas con un indice, entonces luego de recorrer todas las aristas, todas las componentes conexas van a tener un indice distinto y podemos interpretarlas como los clusters del grafo. Para poder calcular el AGM del grafo G se implementaron dos algoritmos, el algoritmo de Kruskal y el algoritmo de Prim, ambos levemente modificados para utilizar estructuras de Lista de Incidencia y Lista de Adyacencia respectivamente. Además, se implentaron dos versiones de Kruskal, una con path-compression y otra sin, a fin de poder compararlos en la experimentación.

2.3. Principios de la Forma (Gestalt Principles)

El método de detección de clusters esta basado en los *principios de la forma de organización perceptible* que están explicados en el paper anteriormente mencionado, en resumen se intento formular un método matemático que detecte clusters utilizando los principios de tal forma que los clusters formados correspondan con los que una persona podría percibir al ver el grafo. Los diversos principios explican como la percepción humana organiza datos sensoriales visuales, reconoce patrones y simplifica imágenes complejas. El principio fundamental para la detección de clusters es el **principio de la proximidad** que dice: "Los objetos o formas que se encuentren cerca unos de otros aparentan **formar grupos**",

incluso si los objetos o formas son muy diferentes van a parecer formar parte de un grupo si se encuentran **cerca**. En este principio, el sentido de que dos objetos estén cerca no necesariamente es el de que su distancia sea corta, varios objetos pueden ser agrupados siguiendo este principio pero aplicado sobre otras características de los mismos, como sus tamaños, sus formas, sus colores, etc.

3. Justificación teórica

3.1. Arboles Generadores Mínimos

En el método explicado anteriormente los clusters se calculan sobre un AGM del grafo completo de los datos, nos interesaría saber como es que el AGM nos garantiza que se van a calcular todos los clusters que pueda haber en el grafo, para eso vamos a ver primero algunos conceptos. Si tenemos un grafo G podemos pensar que una particion de G es una división en dos subconjuntos no vacíos (P, Q), luego definimos la distancia $p(P, Q)$ entre particiones como el peso más chico entre los ejes que tienen un nodo en P y otro en Q. Luego podemos pensar en dos conjuntos de ejes del grafo $C(P, Q)$ es el conjunto de ejes que tienen un nodo en P y el otro nodo en Q, de estos ejes llamamos **cadena** a los ejes cuyo peso sea igual a $p(P, Q)$ y definimos $\lambda(P, Q)$ como el conjunto de cadenas de P y Q. En un grafo con pesos en los ejes resulta útil definir el costo de un camino en el grafo como el **peso máximo** entre los pesos de los ejes que forman parte del camino, de esta forma podemos definir también a un camino entre dos nodos como **minimax** si es el camino con menor peso entre ellos. Ahora con estos conceptos podemos presentar los siguientes teoremas[1] :

- Teorema 1: Para cada partición (P, Q) de G, cualquier AMG contiene al menos un eje de $\lambda(P, Q)$.
- Teorema 2: Todos los ejes del AGM son ejes cadena de alguna partición de G.
- Teorema 3: Si S son los nodos de G y C es un subconjunto no vacío de S tal que para toda partición (P, Q) de C se cumple que $p(P, Q) < p(C, S - C)$ entonces restringir un AGM a los nodos de C forma un subarbol del AGM.
- Teorema 4: Si T es un AGM de G y X e Y son nodos de G, entonces el camino único en T que va de X a Y es un camino minimax entre X e Y.

De estos teoremas el más importante es el tercero, del cual se puede concluir que los clusters van a ser subarboles del AGM , de los teoremas 1 y 2 vemos que los ejes que conectan a los clusters son los de menor peso y el teorema 4 nos asegura que el camino entre nodos va a ser minimax y entonces los nodos conectados no van a tender a alejarse del cluster donde pertenecen.

4. Algoritmos presentados

4.1. Aclaraciones

- El tipo peso es un alias del tipo float

[1] Estos teoremas están demostrados en el apéndice (para los teoremas 1-3) y en las referencias (para el teorema 4) del paper de C.T. Zahn, que se encuentra en la bibliografía

- El tipo AristaAd consiste de un Nodo llamado **adyacente** y un **peso** funciona a como una tupla
- El tipo Arista contiene dos nodos, **desde** y **hasta**, un **índice** y un **peso**, y comparadores de igualdad y orden (Que se realizan en $O(1)$)

4.2. Kruskal

KruskalSinPathComp($ListaIncidencia : grafoCompleto, cantNodos : entero$)

```
1  padre = vector de enteros de tamaño de cantNodos y cargado con el valor
   de su posición en cada posición
2  AGM = lista de incidencia vacia, de tamaño cantNodos-1
3  OrdenarPorPeso(grafoCompleto)
4  for e:Arista ∈ grafoCompleto
5       if getPadre(indice(e.primerNodo), padre) == getPadre(indice(e.segundoNodo), padre)
6            agregar(e,agm)
7  Devolver AGM
```

KruskalConPathComp($ListaIncidencia : grafoCompleto, altura : entero, cantNodos : entero$)

```
1  padre = vector de enteros de tamaño de cantNodos y cargado con el valor
   de su posición en cada posición
2  AGM = lista de incidencia vacia, de tamaño cantNodos-1
3  OrdenarPorPeso(grafoCompleto)
4  for e:Arista ∈ grafoCompleto
5       if getPadre(indice(e.primerNodo), padre) == getPadre(indice(e.segundoNodo), padre)
6            agregar(e,agm)
7  Devolver AGM
```

getPadre($entero : indice, padre : vectordeenteros$)

```
1  if padre[indice] == indice
2       Devolver indice
3  else
4       getPadre(indice(padre[indice]), padre)
5
```

getPadreConPathComp($entero : indice, padre : vectordeenteros, altura : vectordeenteros, nivelesSu$
$entero$

```
1  if padre[indice] == indice
2       altura[indice] = nivelesSubidos
3       Devolver indice
4  else
5       padre[indice] = getPadreConPathComp(indice(padre[indice]),padre,altura,nivelesSubidos+1)
6       Devolver padre[indice]
```

unirPadres($indiceNodo1 : entero, indiceNodo2 : entero, padre : vectordeenteros$)

```
1  padreNodo1 = getPadre(indiceNodo1, padre)
2  padre[indiceNodo2] = padreNodo1
```

UNIRPADRESCONPATHCOMP($indiceNodo1 : entero, indiceNodo2 : entero, padre : vectordeenteros, altu$
$vectordeenteros, nivelesSubidos : entero$)

```
1  padreNodo1 = getPadreConPathComp(indiceNodo1, padre,altura,0)
2  padreNodo2 = getPadreConPathComp(indiceNodo2, padre,altura,0)
3  padreMenosAltura = min(altura[padreNodo1],altura[padreNodo2])
4  padreMasAltura = max(altura[padreNodo1],altura[padreNodo2])
5  padre[padreMenosAltura] = padreMasAltura
```

4.3. Prim

PRIMCONCOLA($G : ListaAdyacencias$)

```
 1  res = Crear vector de pesos del tamaño de la lista G con valor ∞ en cada pos
 2  padres = Crear vector de Nodos del tamaño de la lista G
 3  visitados = Crear vector de booleanos del tamaño de la lista G con valor False en cada pos
 4  cola = Crear cola de prioridad vacia que tome AristaAd
    y los ordene en orden ascendiente según su peso
 5  Encolar el una AristaAd que tenga como adyacente al primer nodo de G y como peso 0 en cola
 6  while not cola.vacia()
 7        u = cola.siguiente()
 8        cola.pop()
 9        visitados[u.adyacente.indice] = True
10        for AristaAd ad : G[u.adyacente]
11              if not visitados[ad.adyacente.indice]
12                    cola.encolar(ad)
13                    if res[ad.adyacente.indice] > ad.peso
14                          res[ad.adyacente.indice] = ad.peso
15                          padres[ad.adyacente.indice] = u.adyacente
16 Devolver padres
```

PrimSinCola(G : $ListaAdyacencias$)

```
1  // G se pasa por referencia
2  distancia = Crear vector de pesos del tamaño de la lista G con valor ∞ en cada pos
3  padres = Crear vector de Nodos del tamaño de la lista G
4  visitados = Crear vector de booleanos del tamaño de la lista G con valor False en cada pos
5  first = G[0]
6
7  for AristaAd elem : G[first]
8        distacia[elem.adyacente.indice] = elem.peso
9        padres[elem.adyacente.indice] = first
10
11 distacia[first.indice] = 0
12 visitados[first.indice] = true
13
14 while not todosVisitados(visitados)
15       v = menorNoVisitado(G.getNodos(), distacia, visitados)
16       visitados[v.indice] = true
17       for AristaAd elem : G[v]
18             if not visitados[elem.adyacente.indice]
19                   if distacia[elem.adyacente.indice] > elem.peso
20                         distacia[elem.adyacente.indice] = elem.peso
21                         padres[elem.adyacente.indice] = v
22 Devolver padres
```

Nota: todosVisitados devuelve en tiempo lineal si todos los elementos del vector de booleanos que toma son verdaderos y menorNoVisitado devuelve el nodo con menor distancia que no fue visitado aún, también en tiempo lineal.

4.4. Clustering

armarGrafoCompleto($nodos$: $vectordeNodos$)

```
1  listaAristas = inicializar lista de incidencia
2  matrizAristas = inicializar matriz de adyacencia
3
4  for i = 0 to tam(nodos)
5        for j = i + 1 to tam(nodos)
6              armar arista con datos de v[i] y v[j]
7              agregar arista a listaAristas
8
9  armar matriz de adyacencia con la lista de incidencia
10 Devolver Matriz de adyacencia y Lista de incidencia
```

retirarEjesInconsistentes($ListaIncidencials, \sigma_T, profVecindario, f_T, forma, cantidadDeClusters$ $vectordeenteros$)

```
1  la = Crear lista de adyacencia a partir de ls
2  Marcar el representante de todos los nodos con 0
3
4  for Arista e : ls
5      if not e.indice == −1
6          if forma == 1
7              Calcular media y desviación respecto del vecindario de los
               extremos de e, avanzando con profVecindario profundidad
8              if e es inconsistente
9                  Saco a e de la y ls
10                 Recorro en la los nodos alcanzables desde un extremo y marco
                   su padre como cantidadDeClusters
11                 cantidadDeClusters + +
12         else
13             Se realiza el mismo procedimiento que antes pero se utiliza
               el segundo criterio de inconsistencia
```

4.5. Complejidad

Ahora vamos a ver las complejidades de los algortimos presentados, los algoritmos de Kruskal y Prim utilizados son los que se explicaron en clase, y de la forma en la que fueron programados las complejidades no fueron afectadas.

Para generar el AGM de kruskal, tenemos funciones que permiten facilitar el entendimiento del pseudocódigo correspondiente. Básicamente se procesa cada arista de la lista de incidencia del grafo completo, previamente ordenadas (ordenada con el sort de C++ en O($\log m$ * m), donde m $\approx n^2$), y se comprueba si la arista tomada actual (que es la de mínimo peso) genera un ciclo en las ya agregadas. Esto se hace mediante una implementación de funciones de UDS (Union Disjoint Set) que nos permite la manipulación de conjuntos de cosas disjuntas como las componentes conexas de un grafo. Una de estas funciones permite encontrar el representante de un punto/nodo (inicialmente cada nodo es su propio representante), lo que sería equivalente a preguntar a que componente conexa corresponde y en caso de ser de diferentes se agrega la arista, si no se sigue a la siguiente. Si la arista se elige se deben actualizar los representantes. Vamos a representar a los nodos en árboles y cada padre será el representante de un nodo, excepto la raíz que será su propio padre. Para mejorar la eficiencia, vamos a almacenar además del padre de cada nodo su altura. Imaginemos que cada representante sería la raíz de un árbol y al momento de buscar el representante de una de los hijos, 'subiríamos' hacia arriba preguntando por el padre de ese nodo, que en caso de no ser la raíz no será su propio representante y nos enviará hacia su padre. Este padre si no es la raíz, tampoco tendrá como representante a él mismo y nos enviará a su representante, así hasta llegar a la raíz. Durante toda esta búsqueda vamos a hacer dos cosas: se actualizará al nodo actual el representante y además se actualizará la altura del nodo que busquemos su representante, aumentando en 1 cada vez que subíamos un nivel en el árbol. Almacenar estas cosas nos sirve para: en primer lugar, la próxima vez que preguntemos por el representante de algún nodo de ese árbol no será necesario realizar otra vez todo ese recorrido (Path compression) porque estarán actualizados los representantes, y por otro lado al momento de unir las componentes conexas, tenemos podemos 'colgar' el árbol más chico del otro, realizando menos operaciones. Además la complejidad de buscar un representante será a lo sumo la altura del árbol (de a lo sumo n nodos) al que pertenece.

Así, al momento de unir las componentes conexas, simplemente actualizamos el representante del árbol más chico (O(encontrar representante(nodo) $\approx$ O(altura árbol(nodo)) con el más grande. Entonces buscar un representante y unir componentes $\in$ O(log n).
Siguiendo con la explicación, el procesamiento de todas las aristas (donde m $\approx n^2$)sería O(n^2 * log n + log n^2 * n^2) $\approx$ O(log n^2 * n^2) $\approx$ O(log n * n^2).
Para PrimSinCola, el for al principio es $O(n)$ como cada Nodo agregado se marca y hay n nodos, sabemos que el ciclo while a lo sumo se ejecuta n veces, dentro de cada ciclo se calcula el nodo con distancia mínima en $O(n)$, y se acutalizan las distancias de los nodos adyacentes al mínimo en O(cantidad de nodos adyacentes al mínimo) que a lo sumo es n, entonces su complejidad también es $O(n^2)$.
Veamos la complejidad de RetirarEjesInconsistentes, obtener la lista de adyacencia a partir de la de incidencia se puede hacer recorriendo todas las aristas y es $O(m)$, marcar el padre de todos los nodos es $O(n)$. Recorrer todas las aristas de la lista de incidencia es a lo sumo $n - 1$, para buscar ejes inconsistentes vamos a aprovechar las ventajas que nos brindan las implementaciones de listas de adyacencia y incidencia.
La idea es la siguiente: vamos a recorrer todas las aristas, sacándolas de la lista de incidencia y calcular con un método parecido a BFS los promedios de longitud de vecino, desviaciones, etc. Hay m aristas (que por estar trabajando con un AGM son $\approx$ n) y por cada una calcular esas cosas es $O(n + m) \approx O(n)$. En caso de ser considerado consistente vamos a recorrer, también con algo parecido a BFS, los nodos alcanzables desde uno de los extremos de la arista considerada inconsistente y le vamos a modificar su representante, con un vector de n posiciones donde en la posición i-ésima está el representante del nodo i-ésimo. Por temas de implementación, si la arista es inconsistente vamos a eliminarla de la siguiente forma: en la lista de incidencia modificamos su indice a -1, que es un valor que consideramos internamente como arista no disponible, y en la lista de adyacencia sí la eliminamos, accediendo en tiempo constante a la posicion del vector de ese nodo y con un iterador recorriendo sus nodos vecinos hasta encontrar el que corresponde con el otro extremo de la arista. Esto cuesta O(m) $\approx$ O(n), porque para cada nodo se recorre a lo sumo su grado y si se hace para todos los nodos será la suma de los grados de los nodos que es asintóticamente m, que por ser un árbol es O(n). Análogamente se recorren las aristas vecinas si se decide utilizar el segundo criterio de eje inconsistente
Por lo tanto, la complejidad total sería O(n^2). Usar ambas listas nos permite tener los beneficios de recorer y 'deshabilitar' si es necesario las aristas en tiempo constante por la lista de incidencia y utilizar una modificación de BFS para poder calcular lo necesario y reasignar los representantes en caso de ser necesario de forma óptima por la lita de adyacencia.

Veamos la complejidad de ArmarGrafoCompleto, una vez que se almacenan todos los puntos del espacio ingresados, se procesarán uno a uno de la siguiente forma: por cada uno de ellos se recorrerán todos los que vengan después (en orden de ingreso). Esto nos asegura que cada nodo estará conectando con todos los demás (excepto él mismo) y así conseguiremos el grafo completo que tiene como nodos a los puntos ingresados. Esto lo conseguimos con complejidad O(n^2).

5. Experimentación

5.1. Variaciones

Tomaremos los casos de test brindados en http://cs.joensuu.fi/sipu/datasets/ y correremos nuestro algoritmo para poder obtener las clusterizaciones.

Los experimentos estarán centrados en analizar las diferentes tipos clusterizaciones que pueden realizarse variando las definiciones de eje inconsistente. Intentaremos analizar las configuraciones necesarias para que se pueda alcanzar una clusterización lo más cercana a la de la percepción humana y los resultados interesantes al que pueden llegarse. Para cada clusterización utilizaremos como solución previa los AGM obtenidos por los algoritmos de Kruskal, Kruskal con path compression y Prim, y a su vez indicaremos en cada caso las mediciones de tiempo correspondientes. Para ello usaremos la porción de código utilizada en clase para medir tiempos con una cantidad de tres repeticiones de cada uno para una mejor precisión.

Dados los siguientes: f_T multiplicador del promedio, σ_T multiplicador de la desviación, y la profundidad del vecindario de los extremos del eje candidato, W(XY) el peso del eje candidato, y sea X e Y sus nodos extremos, definiremos un eje inconsistente:

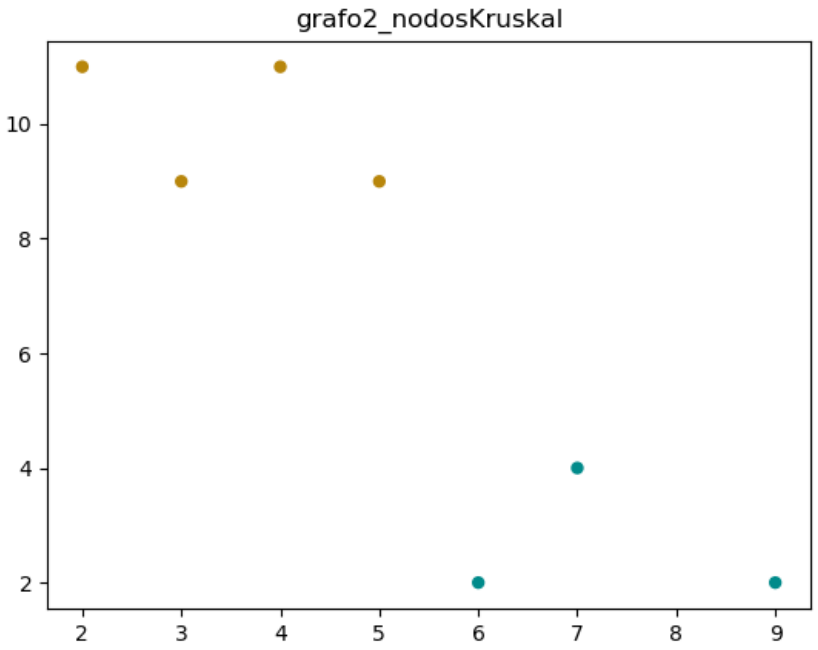
- Forma 1: $\frac{W(XY)}{Promedio(Vecindario(X))} > f_T \; y \; \frac{W(XY)}{Promedio(Vecindario(Y))} > f_T$,
  Es decir, la proporción entre el peso del eje candidato y el promedio de peso del vecindario de sus extremos es mayor al coeficiente dado.
- Forma 2: $W(XY) > Promedio(Vecindario(X)) + \sigma_T * desviacion(Vecindario(X))$ $y \; W(XY) > Promedio(Vecindario(Y)) + \sigma_T * desviacion(Vecindario(Y))$,
  Es decir, que el peso del eje candidato supere al promedio del vecindario de sus extremos por al menos σ_T unidades de la desviación del vecindario del extremo, y además se cumpla la forma 1.

Comencemos recordando que la clusterización no es un problema matemáticamente definido y por ello podemos definirlo de forma tal que se obtenga diferentes resultados para mismas situaciones. En nuestro caso, tenemos datos númericos correspondientes a puntos en el plano y en donde vamos a tomar como relación primaria entre ellos como su distancia euclídea. Experimentaremos con casos ya conocidos de clusterización y evualuaremos fuertemente la habilidad del modelo y sus parámetros para acercarse a los resultados de nuestra percepción, aunque siempre teniendo en mente los principios de agrupación propuestos por la filosofía de Gestalt previamente analizados.

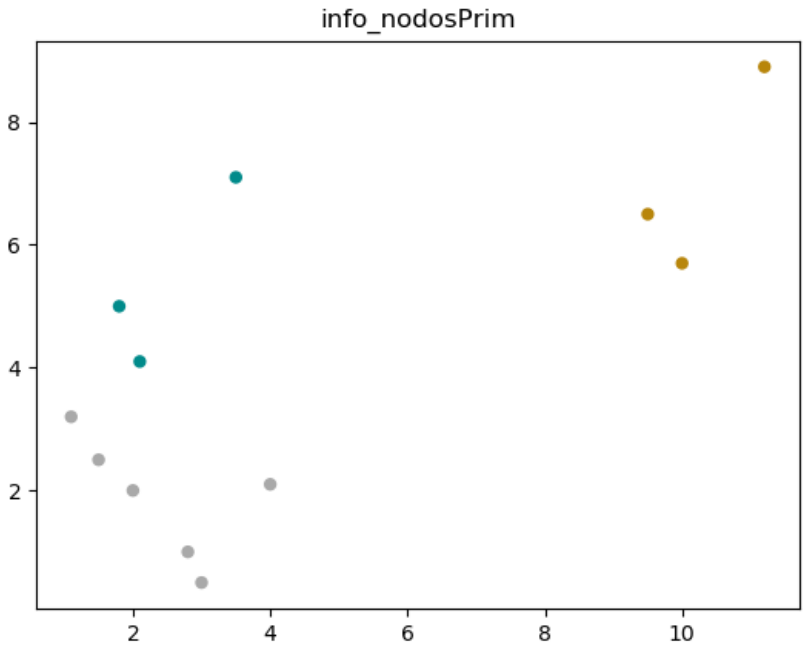
Además, vamos a referirnos al método utilizado para la generación del AGM para referirnos a una determinada clusterización. Vamos a utilizar los datasets provistos por: `http://cs.joensuu.fi/sipu/datasets/`

5.2. Experimentación

Comenzaremos la parte de experimentación exponiendo algunos casos particulares de lo que se trata la clusterización.



Intuitivamente los colores marcan ciertas diferencias entre los conjuntos de puntos. A simple vista los naranja no parecen ser del mismo tipo que los azules, aunque habría que definir que significa eso técnicamente. En este caso estamos suponiendo que dos uniones de puntos que tengan distancia diferente son de diferente tipo. En este caso 'distancia diferente' significa una diferencia del doble de largo. Pero esta claro que en otro contexto dos cosas diferentes necesiten ser 'más' diferentes y la diferencia tenga que ser por ejemplo del triple y entonces acá no habría más que un grupo de puntos. Pero también podría ser lo contrario y considerar que dos puntos separados por una distancia muy chica ya son de diferente tipo y entonces podríamos conseguir que cada punto sea de un tipo diferente. A todo esto podemos involucrar las desviaciones que existen entre vecinos de puntos, porque podríamos querer ser un poco más restrictivos respecto de que separar, puede que en nuestro contexto nos importe cómo están distribuidos los vecinos de un punto más allá de las distancias que los separen.

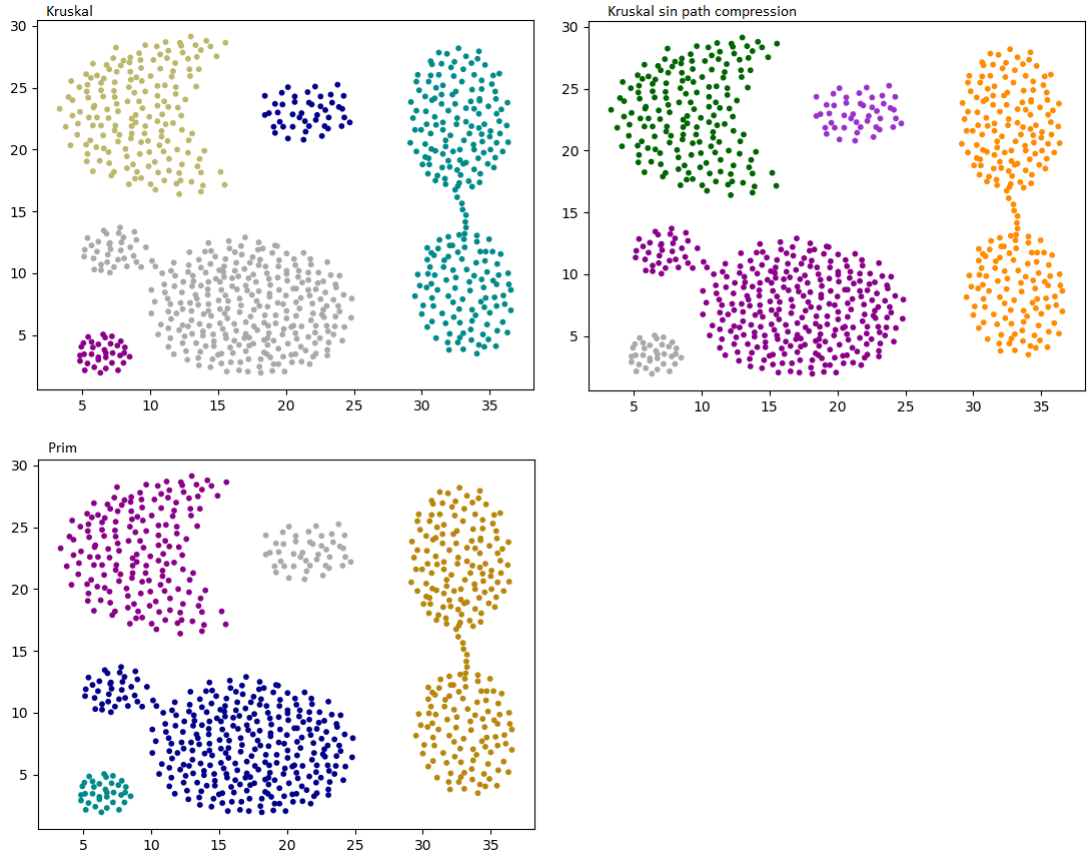


Por ejemplo en este caso, del lado izquierdo tenemos una serie de puntos que no parecen estar lo suficientemente alejados y sin embargo las desviaciones entre sus vecinos nos permiten ser más restrictivos con la medición.

Con esta breve introducción, planteamos algunos interrogantes que pueden surgir y que sería interesante investigar en un análisis más profundo y completo de una mayor variedad de casos.

Cómo conseguir clusters parecidos a nuestra percepción? Es siempre posible? Cómo varían los diferentes métodos de búsqueda de AGM para la clusterización? Como afectan a la clusterización la variación de los parámetros definidos para considerar un eje inconsistente?

5.3. Dataset: Aggregation



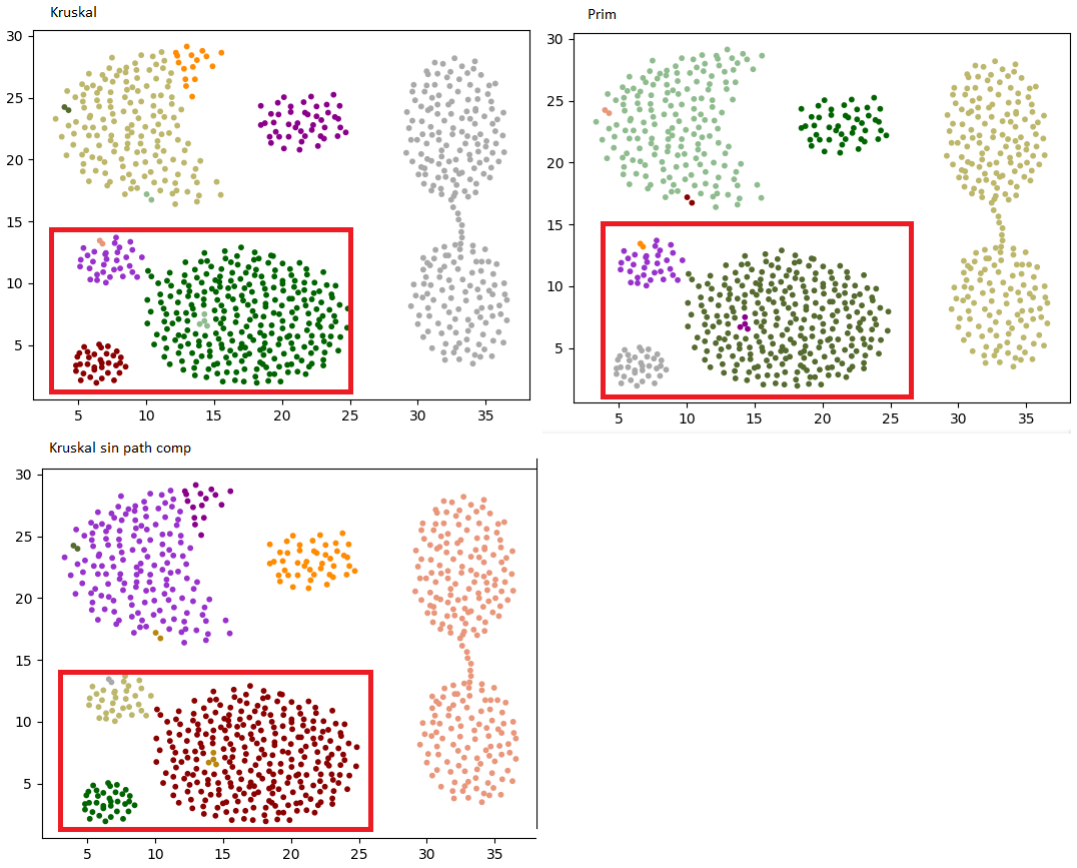
5.3.1. Tiempos

Cuadro 1: Tiempos de ejecución en milisegundos

Cant.Puntos	Kruskal	Kruskal sin path comp	Prim
788	254.66	386.66	151.66

Analizando el siguiente caso podemos ver que generando el AGM con cualquiera de los métodos elegidos obtenemos la misma clusterización. Elegimos como parámetros que un eje inconsistente debe superar en 2.5 al promedio de los vecinos de 2 pasos de sus extremos. La clasificación resulta en 5 clusters. Como primera observación, la percepción humana podría arrojar que el cluster de la esquina derecha en realidad podrían ser dos clusters diferentes, al igual que el cluster de la esquina izquierda. Si intentamos conseguir algo más cercano a esto, podemos intentar clasificar ahora también teniendo en cuenta la desviación. Como todos los puntos tienen una densidad de vecinos parecida, no tiene mucho sentido modificar la distancia del vecindario de los extremos (actualmente dos). Si reducimos el coeficiente del promedio de 2.5 hasta 1.4 y el coeficiente de la desviación lo fijamos en 2, d = 2. Después de varios intentemos, encontramos que en ese punto obtenemos lo

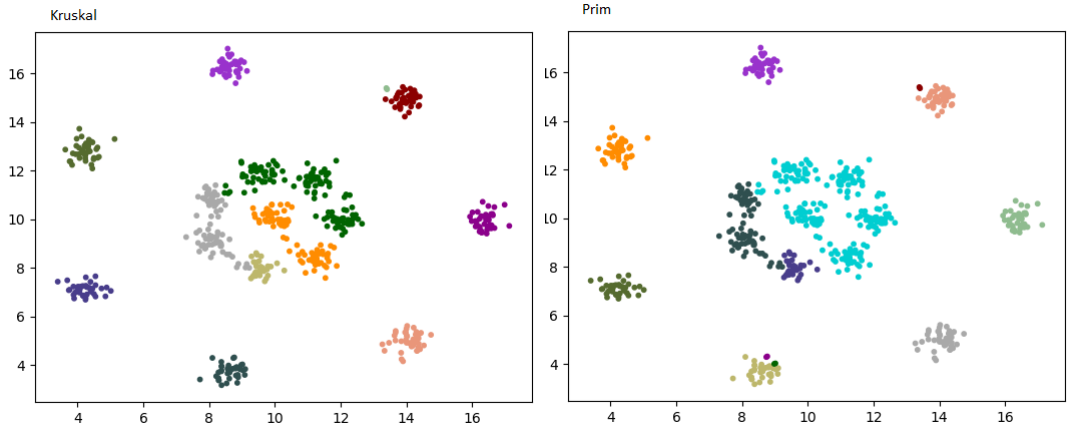
siguiente:



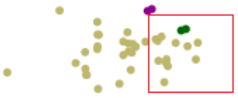
Por un lado, conseguimos auentar los clusters pero las modificaciones generan ciertas inconsistencias en zonas de puntos donde la clusterización estaba bastante cercana a nuestra percepción. Todo esto se da en un contexto en donde buscamos ejes inconsistente que cumplan con los requisitos anteriormente mencionados y el problema es que como se ha visto anteriormente, nuestra percepción suele utilizar en una serie de factores que no necesariamente se corresponden con un método de clusterización en cuestión. Por ello es posible y buscaremos en esta experimentación convencernos lo más posible que la clusterización puede variar lo suficiente con determinados cambios en los requisitos tomados para evaluar y su correspondencia con nuestra percepción puede estar notablemente alejada.
En nuestro gráfico esto genera que nosotros veamos tres clusters bien definidos en la esquina inferior izquierda y nuestro método sólo detecta correctamente dos. En este caso la razón es muy simple, la unión de los conjuntos de puntos marcados en rojo esta formada por un 'cuello' que nuestra percepción ignora y por eso separamos los conjuntos en dos clusters diferentes. Mientras que computacionalmente, ese cuello no cumple nuestra noción de eje inconsistente y en caso de que tratemos de forzarlo comenzamos a tener estas inconsistencias, porque nada impide que otros dos puntos con características similares a este cuello se encuentren dentro de uno de los clusters y en ese caso, por la propia consistencia de nuestro método, también sería considerado inconsistente.
Otro punto interesante a considerar, es que para estos nuevos parámetros conseguimos diferentes clusterizaciones. Particularmente encontramos un mini-cluster en el cluster turquesa de Kruskal sin path compression y en Prim, y ausente en Kruskal. Esto proviene

por como buscamos ejes inconsistentes -recorriendo las aristas del AGM- y cómo se construyeron los AGM. En la sección de algoritmos se explica el funcionamiento y sólo vamos a resaltar que entre Kruskal y Prim la generación es diferente, y mientras en Prim estamos construyendo todo el tiempo sobre el árbol actual, en Kruskal nos limitamos a guardar las aristas de menos peso que no generen circuito con las actuales y esto en algún paso k-ésimo no necesariamente es un árbol, excepto que k sea el último. Por otro lado, Kruskal sin path compression al agregar una nueva arista y unir las componentes conexas lo hace sin intentar optimizar esa acción, y esto puede generar que los referentes de una componente conexa de Kruskal sean diferentes a los de la misma componente conexa en Kruskal sin path compression. Entonces una de las cosas que vamos a ver es si esto es realmente factible y entonces podemos utilizar esta técnica para poder ver a grandes rasgos los clusters y en función de nuestro interes enfocarnos en una zona en particular. Esto nos facilita porque al momento de volver a clasificar esa zona, lo haríamos restringiendo nuestro domino de patrones (anteriormente de todo el conjunto de puntos) hacia uno más pequeño.
Como primeras conclusiones podemos decir que en contextos en donde hay conjuntos de puntos con densidades similares y que existen ciertas uniones con desviaciones y promedios similares (recordemos que la desviación la calculamos entre los vecinos y por más que en ese cuello hay menos vecinos en comparación con algún punto en medio de un cluster en particular, como las distancias entre nodos son similares en ambos casos, la desviación y largo promedio en ejes también lo son) esto hace que no se 'encuentren' ejes inconsistentes.

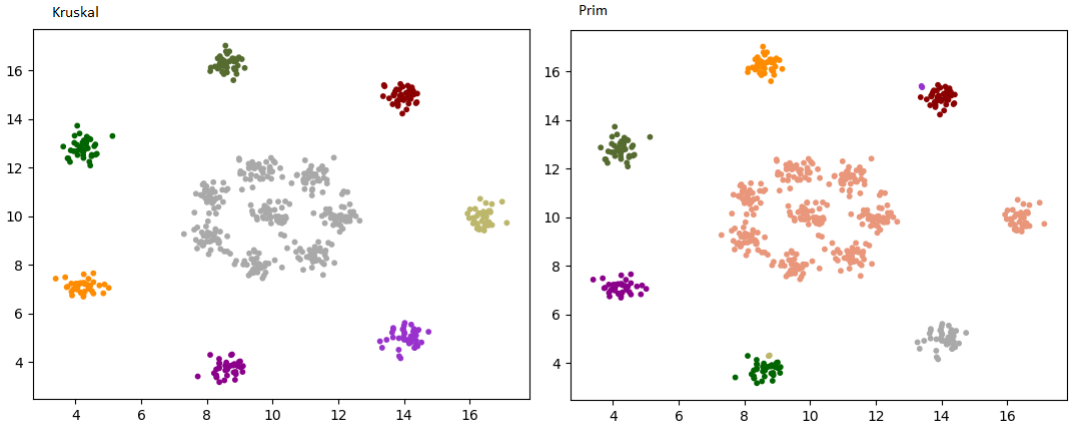
5.4. Grafo r15



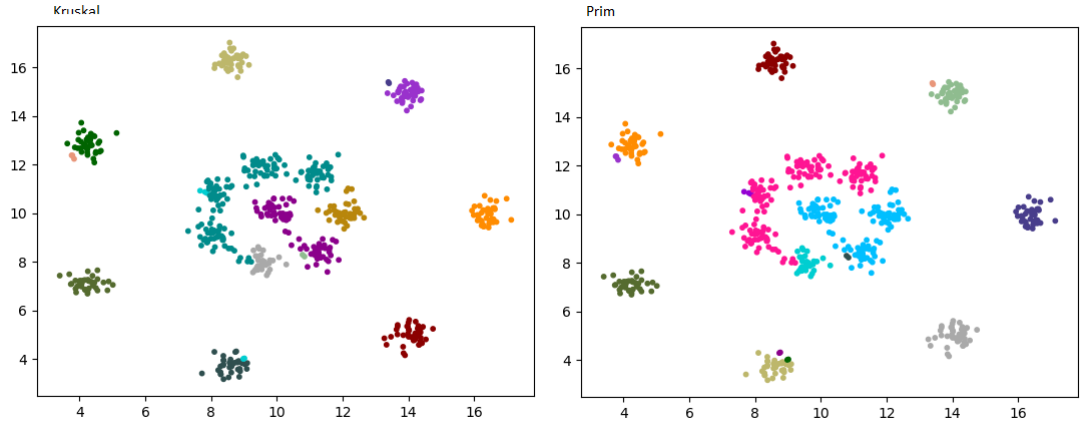
Fijamos $\sigma_T = 3.3$ y $f_T = 2.4$, d = 2. Por razones de espacio evitamos poner a Kruskal sin path compression porque dan exactamente lo mismo. Los valores de los parametros fueron: factor de proporción en 1.8 y la desviación en 3.8. En este caso, pequeñas modificacione en el factor de proporción o desviación no generaban clusters diferentes. Sin embargo se puede apreciar cómo los clusters exteriores están definidos muy parecida a cómo sería esperable mientras que en el centro empiezan a juntarse puntos de -a nuestra percepción- diferente cluster. El problema es que al mirar con atención los posibles ejes que unen en el AGM los nodos que serían de diferente cluster cumplen ciertas cosas:



Si hacemos un zoom a la zona del cluster marrón claro de Prim que tiene como dos clusters adicionales y no intuitivo para nuestra percepción, demarcamos la zona marcada con rojo y obsvervamos el caso del cluster verde es bastante visible que cualquier de los posibles ejes que los une (a los nodos) en el AGM va a cumplir tener mayor factor de proporción de promedio con respecto a los vecinos. Respecto de la desviación podría justificarse de un modo similar. Entonces podemos justificar de forma mas o menos razonable por qué se consideran - y bien- inconsistentes para nuestro modelo.
Modifiquemos ahora un poco la forma de clusterizar. Fijemos d = 3, porque en este caso tiene sentido pensar que nos va a dar un poco de mayor consistencia en las mediciones, y utilicemos sólo la proporción entre ejes con $f_T = 2.6$.



Como era esperado, el centro ahora se convierte en todo un gran cluster. Era algo esperado al agrandar los vecindarios, porque de esa manera por la densidad y distribución de los puntos de centro tiene sentido pensar que al tener en cuenta más vecinos será más complicado superar los 2.6 de proporción de eje. Si llegaramos a reducir el valor, es posible que obtengamos clusters no tan representantivos porque al sacar la desviación de consideración estamos sacando información útil para decidir si separar dos conjuntos de puntos más alla si existe algún eje lo 'suficientemente' largo.



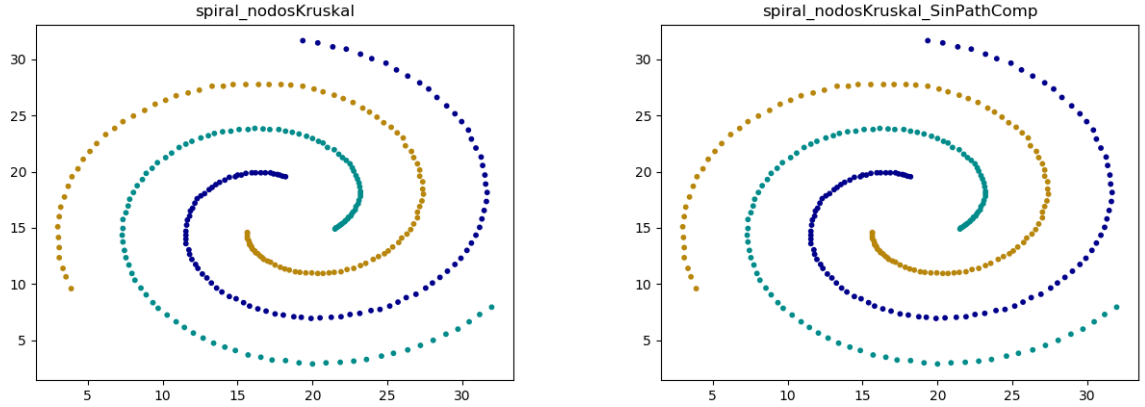
Efectivamente podemos ver que se encuentran más clusters debido a que como dijimos hay menos información con la cuál decidir, es decir, quizá dos nodos están suficientemente alejados pero por la dispersión de vecinos que tienen a nuestros ojos forman parte de lo mismo.
Para cerrar este tipo de contextos, encontramos que al igual que antes, Prim suele tender a generar clusters 'aislados' que intentaremos dar explicar viendo más casos y recopilando más información.

5.5. Dataset: Spiral

5.5.1. Tiempos

Cuadro 2: Tiempos de ejecución en milisegundos

Cant.Puntos	Kruskal	Kruskal sin path comp	Prim
312	33	32	32.66



5.5.2. Gráficos

En este caso fijamos $f_T = 3.3$ y la profundidad del vecindario la fijamos en 2 y 3. Es razonable pensar que ambos AGM serán similares porque no hay demasiadas opciones de ejes para los nodos, porque cada una de las líneas de puntos tiene pocos vecinos cercanos (si pensamos en el grafo completo) con los cuales tiene sentido que se una formando una distancia mínima. Imaginemos el grafo completo y su conversión a AGM y pensemos que entre cada una de las líneas de puntos que vemos como clusters diferente existe una arista que las une (por ser un árbol) y si habría más de una seguro se podría encontrar un árbol generador con menor peso y esto sería absurdo porque Kruskal y Prim nos garantizan que lo consiguen, de modo que tiene sentido pensar que entre cada una de esas líneas existe sólo una arista que las une y que su peso es mucho mayor al de cualquiera de los demás ejes. Entonces al momento de experimentar, pensar en la profundidad de vecinos o proporción de eje inconsistente desde el detalle no tiene gran sentido, dado este caso donde las líneas están bien separadas no hay problemas para conseguir una separación de nodos intuitiva.

5.6. Dataset: Pathbased

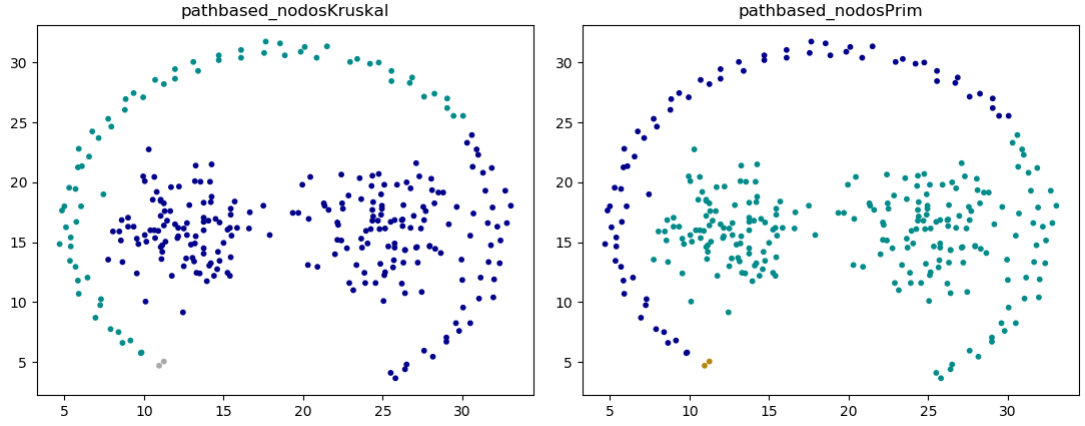
5.6.1. Tiempos

Cuadro 3: Tiempos de ejecución en milisegundos

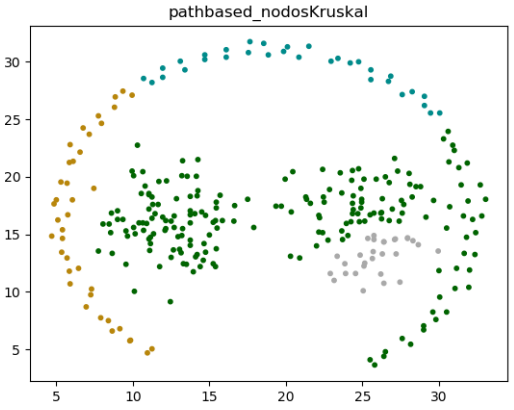
Cant.Puntos	Kruskal	Kruskal sin path comp	Prim
300	32	46	27

5.6.2. Gráficos

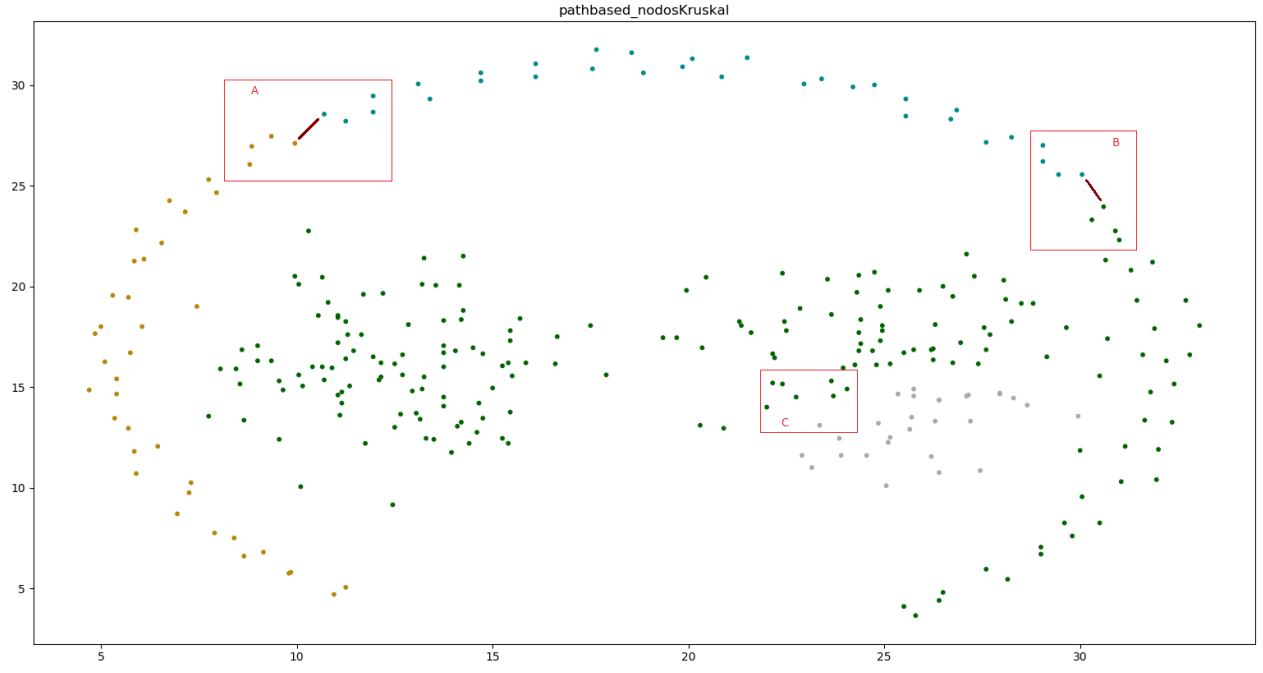
En este caso, tenemos algo similar al anterior pero tenemos una distribución de puntos que nos hace pensar que probablemente sea más dificil de encontrar una clusterización clara y tengamos que ajustar las desviaciones, el vecindario, y las proporciones de peso entre ejes. clusterización clara.



Como vemos, fijando $f_T = 2{,}35$, d = 3, lo que conseguimos no es lo que esperaríamos ver clusterizado. Es otro caso en donde la distribución de los puntos, la cercanía entre los clusters esperables, la relación con los pesos de los ejes del grafo, entre otras cosas, hacen que sea necesario trabajar un poco más para poder conseguir algo mejor.



Incluyendo ahora la desviación y ajustando un poco más los parámetros con $f_T = 1{,}9$, $\sigma_T = 2{,}5$ y d = 2 tenemos algo que nos da más información para analizar. Para que sea más fácil de ver lo señalamos en rojo y con un zoom a la imagen. Siempre es importante tener en la cabeza el posible AGM y a partir de ello analizar las razones por las que hay separaciones inesperadas de cluster.



El tema es claro: reducir el coeficiente de proporción de pesos nos genera clusters no esperados pero que si los analizamos en particular son correctos según nuestra medición. En las zonas señalizadas pusimos un eje bordo que simboliza la posible unión de los nodos en el AGM. Es claro que ese eje en proporción a los vecinos es superior y que para poder 'estabilizarlo' (en términos de evaluarlo y clusterizarlo como nosotros lo vemos) habría que tomar un vecindario más grande, posiblemente con d igual a 4 ó 5. En la zona C pasa algo similar y se le agrega algo: la distribución de puntos es diferente, lo cual genera que este tipo de situaciones sean difíciles de separar.

5.7. Dataset: A1

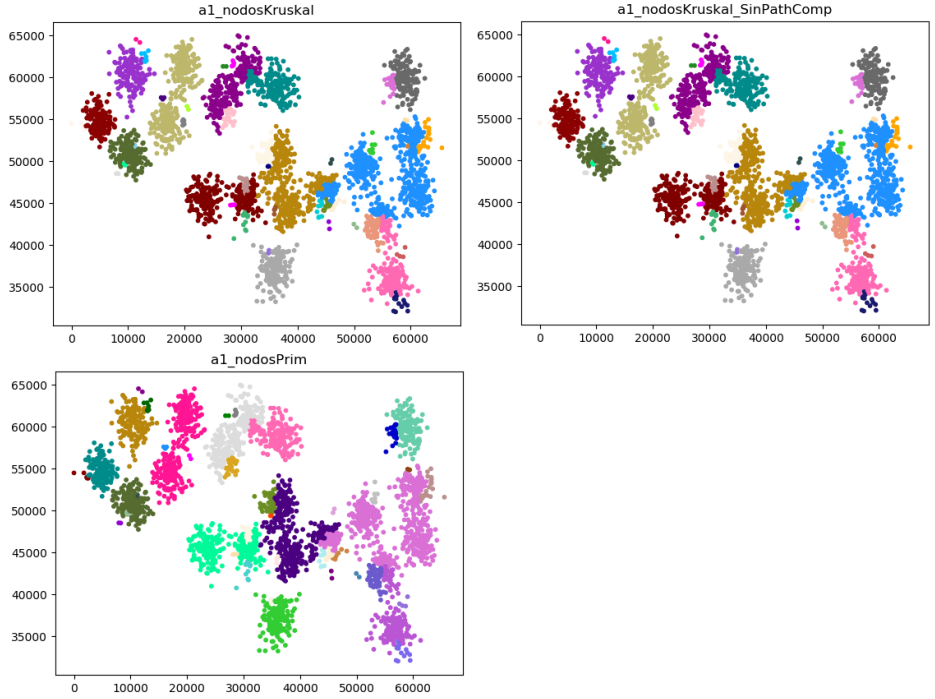
5.7.1. Tiempos

Cuadro 4: Tiempos de ejecución en milisegundos

Cant.Puntos	Kruskal	Kruskal sin path comp	Prim
3000	3839.66	5934.33	1939.33

5.7.2. Gráficos

En este caso tenemos una clusterización bastante evidente a simple vista sin embargo después de variar los parámetros lo más parecido que podemos conseguir es lo siguiente:



Volvemos a encontrar un factor que se viene repitiendo en los distintos datasets: la transición entre lo que a simple vista son diferentes clusters algorítimicamente no suele ser fácil de detectar. Modificando los parámetros podemos llegar a solucionar el problema en alguna zona en particular y al mismo romper lo anteriormente bien -a nuestra percepción- clusterizado. Un posible camino sería subdividir las áreas de puntos que queremos clusterizar y separarlas en función de patrones similares que permitan fijar los parámetros sin romper otras zonas ysa clusterizadas. Sin embargo en este caso en particular no es muy claro qué áreas subidividir.
Veamos otro caso en donde sería un poco más claro cómo dividir en caso de que pase esto.

Cuadro 5: Tiempos de ejecución en milisegundos

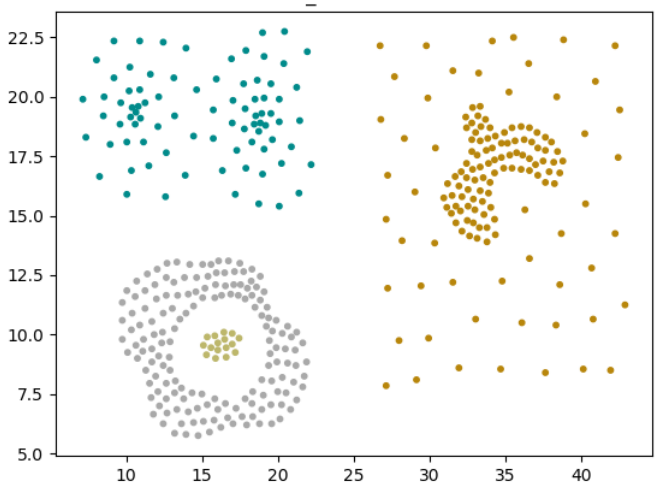
Cant.Puntos	Kruskal	Kruskal sin path comp	Prim
399	61	90.33	46

5.8. Dataset: Compound de Zahn

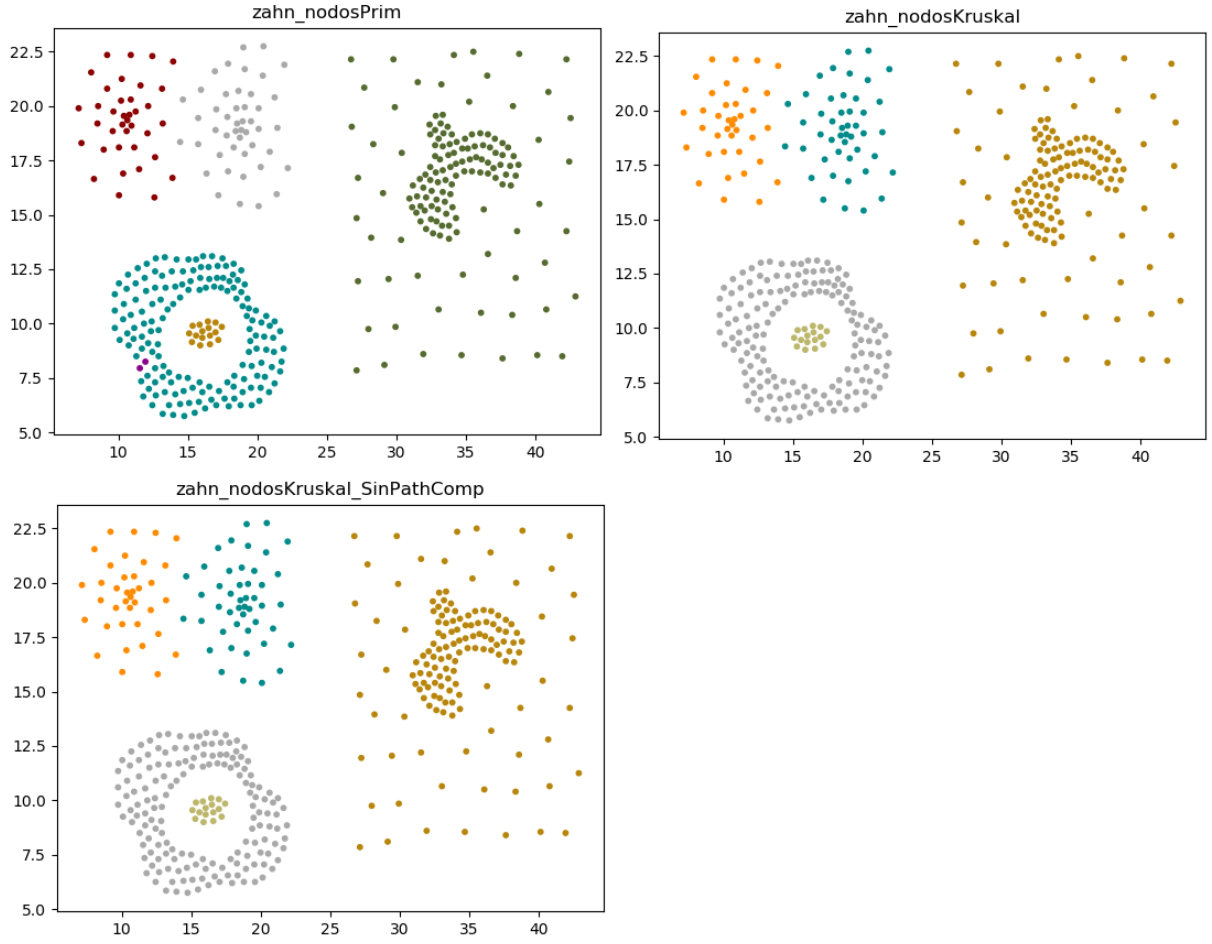
5.8.1. Tiempos

5.8.2. Gráficos

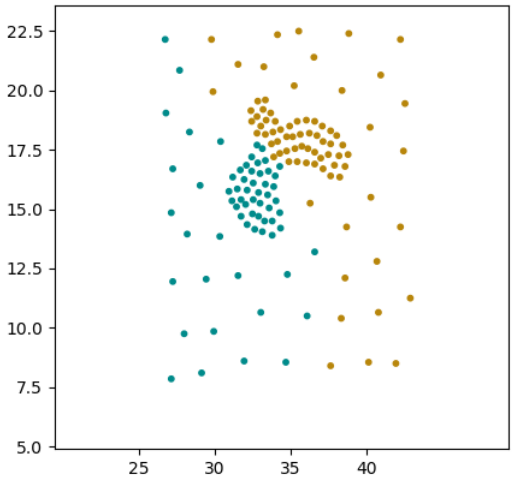
Fijando f_T = 1,19, d = 2, σ = 4.3 encontramos lo siguiente: (por motivos de espacio y que los 3 AGM dan igual sólo ponemos 1)



Es claro que la clusterización obtenida está cerca de la esperada, aunque también sería esperable que los de la esquina superior izquierda sean dos diferentes y que el de la derecha también lo sea. Primero vamos a tratar de modificar los parámetros para obtenerlo y si no podemos vamos a intentar aplicar clusterizar por partes, porque acá sí es más claro que tenemos tres zonas que podemos tratar por separado.



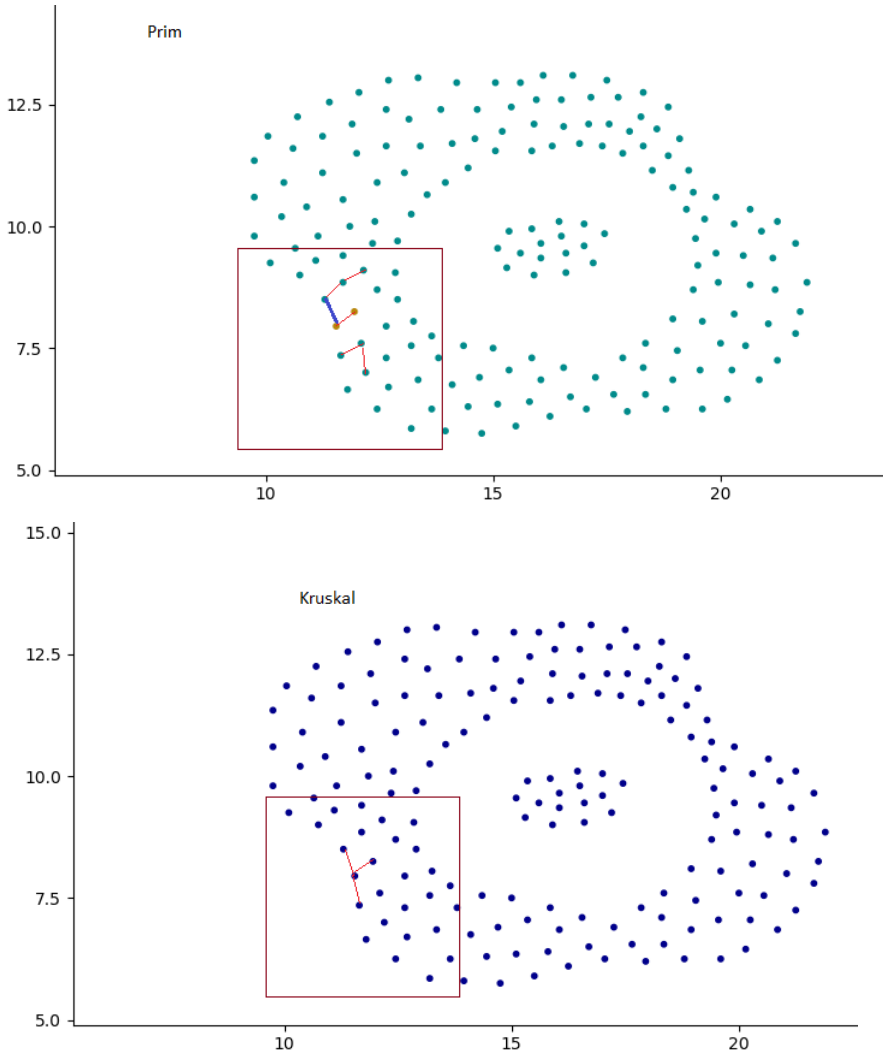
Si miramos con atención, la solución que utiliza Prim generó un cluster adicional, mientras que Kruskal no, aunque ninguno pudo clasificar en dos la parte derecha de la imagen. Intentemos probar ahora clusterizar esa parte sola, tomando a partir de x >= 25 y ver si podemos clusterizarla y en cierto modo terminar clusterizar 'bien' todo el gráfico.



Obtuvimos está nueva clusterización fijando d =1, $\sigma = 1.2$, $f_T = 1.2$, y nuevamente nos encontramos con dificultades para poder separar grupos de puntos que sacándolos del contexto anterior parecía bastante sencillo de separar, los Gestalts principles se hacen pre-

sentes en casi todos los experimentos.

Retomando el gráfico anterior donde Prim generaba un cluster adicional a Kruskal, lo que hicimos fue hacer un zoom en esa región y análisis en detalle, marcando en rojo cuáles son las aristas que unen esos nodos en particular. Obtuvimos lo siguiente:



Ambos métodos tienen diferentes técnicas para generar un AGM, esto es algo que no podemos controlar en profundidad en todos los datasets pero que en este en particular notamos, aunque es posible que en otros también ocurriese y no se notara porque no afectaba a la clasificación de los nodos. En este caso sí y se nota claramente en la imagen. Internamente, la arista en cuestión es la número 280 en Kruskal mientras que Prim la consiguió como arista 115. En Prim imaginamos que esta parte es la especie de 'hojas' del árbol y que la raíz está hacia la derecha. Marcamos con azul la arista que se considera inconsistente. En Kruskal aparentemente sería una situación inversa, estamos donde sería algo similar a la raíz y hacia el otro lado estaría esta situación. En la imagen es apreciable que al cambiar esas relaciones el vecindario también es diferente y en la zona que une Kruskal aumentan levemente los promedios de pesos de ejes y esto impide que sea considerado inconsistente.

5.9. Conclusiones

Durante la experimentación con diferentes tipos de conjuntos pudimos comprobar en persona las discusiones que tuvieron lugar en secciones previas de la introducción teórica. Vimos cómo son afectados los análisis en función de pequeños movimientos de los parámetros y la dificultad para encontrar una configuración que permita una clasificación cercana a nuestra percepción en todo su conjunto. En caso de tener que clusterizar diferentes gráficos se deberá necesariamente contar con el tiempo para una experimentación exhaustiva que permita realizarlo y en muchos casos la subdivisión de los gráficos para el tratado de zonas que presenten patrones de caracterización diferentes y estar preparados para no conseguir los clusters idénticos a lo esperado.
También cabe mencionar las diferentes técnicas de generación de AGM, donde por lo menos a nivel clusterización no hubo una diferencia considerable para alguno en la clusterización de algún conjunto en particular. Sí hay que mencionar las diferencias de performance evaluadas: es claro que más alla de la complejidad teórica Kruskal con path compression en la práctica es mucho mejor y en algunas situaciones es más notable que en otras. Además, en este trabajo en particular se construyó el AGM partiendo del grafo completo, lo que quiere que decir que una técnica como Kruskal en la que es necesario ordenar las aristas se ve altamente perjudicada. Otras forma consisten en calcular un grafo con los k-vecinos más cercanos para cada nodo y de esta forma evitar partir de un grafo completo en donde $m \approx n^2$.
En general, clusterizar en determinados casos requerirá aceptar cierto grado de error, debido a la falta de comprobación exacta de los resultados, y el contexto determinará cuán exacto y exhaustivo, respecto de una percepción esperada, deberán ser los métodos considerados para la clusterización.

6. Arbitraje

En el mundo de la finanzas se denomina *arbitraje* al hecho de comprar y vender un recurso de manera simultánea para generar una ganancia, aprovechóse de los desbalances de los precios en diferentes mercados. La idea es que al ser operaciones simultáneas el abitrajista no corre riesgo en sus transaccciones. En genewral, los desbalances necesarios para darse una situación de arbitraje provienene de desincronizaciones de los mercados.

Supongamos que hay un recurso R disponible para vender y comprar en los mercados M_1 y M_2. En M_1 nos dan 36.80 pesos por vender u na unidad de R u nos cuesta 37.50 pesos comprar un a unidad. En M_2 en cambio nos dan 36.90 pesos por vender una unidad de R y nos cuesta 37.65 pesos comprar una unidad. En este caso no existe oportunidad de arbitraje. No hay gorma con estos dos mercados que vendamos y compremos unidades del recurso R y generemos una ganancia. Sin embargo, en un determinado momento del día sucede un evento que valoriza el recurso R. M_1 se entera de este suceso y actualiza sus cotizaciones. Luego de este cambio nos dan 37.67 pesos por v ender una u nidad de R y nos cuesta 39.95 pesos comprar una unidad. Las noticias llegan más tarde a M_2 por lo que todavía no se enteró del suceso que valorizó a R, por los que sus precios siguen intactos por un período más. En este caso sí existe oportu nidad de arbitraje. Podemos simultaneamente comprar varias unidades del recurso R en M_2 y vender varias unidades en M_1. De esta manera, por cada unidad que se compra/vende estamos obteniendo 0.02 pesos sin ningún tipo de riesgo.

Afortunadamente, este situación, esta situación no es algo que suceda a menudo. De hecho, en el caso que suceda, muchas veces no incluye un único recurso sino que involucra a varios recursos en varios mercados distintos.

En este problema lo que nos gustaría es si existe oportunidad de arbtiraje entre varias divisas (Pesos, Dólares, Libras, Reales, Bitcoins, etc.). Para eso contamos con la tasa de cambio entre cada par de divisas. En caso de existir arbitraje, queremos saber qué divisas son involucradas en el proceso.

A continuación se expondrán algunos ejemplos, cada uno con su respectiva solución en caso de existir, para poder entender mejor el problema.

Ejemplo No. 1:
Para el primer ejemplo tomaremos las divisas:

- d_0 = Dólares estadounidenses.
- d_1 = Pesos argentinos.
- d_2 = Euro.
- d_3 = Libra esterlina.

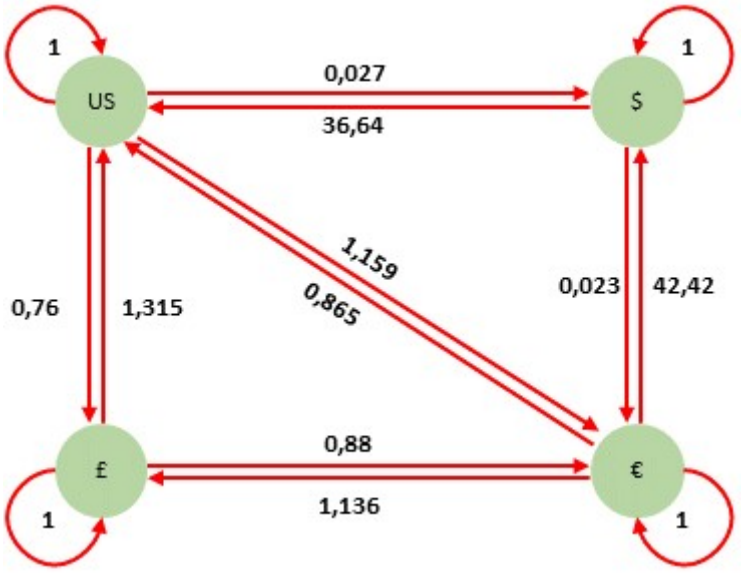
y la "matriz de cambiosçonformada por el peso de los ejes:

$$M = \begin{bmatrix} 1 & 36,64 & 0,863 & 0,760 \\ 0,027 & 1 & 0,023 & 0,020 \\ 1,159 & 42,42 & 1 & 0,88 \\ 1,315 & 48,20 & 1,136 & 1 \end{bmatrix}$$

En la que tomaremos estos pesos como los factores multiplicativos para pasar del valor de un nodo a otro, por eso cada nodo tendrá distancia 1 de si mismo, entonces, si observamos el grafo asociado a la matriz, podemos ver que este será un pseudografo dirigido completo. Será completo ya que cada divisa tendrá que tener un valor de cambio para otra divisa.

En resumen: $m_{i,j}$ será lo que tenga que multiplicar el valor de lo que tenga en la divisa d_i para pasar a la divisa d_j.

El grafo asociado a nuestra matriz será:



Para que nuestro problema tenga solución debemos encontrar al menos un ciclo de divisas tales que al volver a la divisa de la que empezamos haya arbitraje.

Y, si probamos con un ciclo que empieze con el dolar tenemos los siguientes resultados:

- Si el ciclo es d_0,d_1,d_0 es decir (d ólares-pesos-dólares) tenemos que al regresar nuestra divisa valdra 1*36,64*0,027 de lo que valía en un principio, es decir 0,9893 por lo tanto no se producirá arbitraje.
- Si el ciclo es d_0,d_2,d_0 es decir (d ólares-euros-dólares) tenemos que al regresar nuestra divisa valdra 1*0,863*1,159 de lo que valía en un principio, es decir 1,0002: 2 diezmilesimas más que antes, por lo cual aquí si se producirá arbitraje

Como pude encontrar al menos un ciclo de divisas que me generara arbritraje, mi solución será: d_0,d_2,d_0.

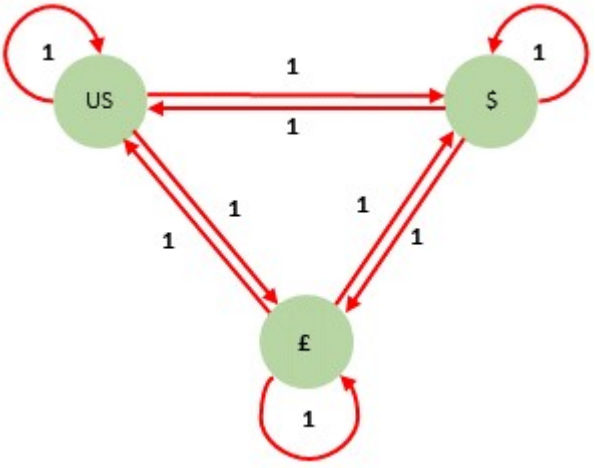
Ejemplo No. 2:

Para el segundo ejemplo se considerará un caso en particular: cuando las divisas tienen el mismo valor de cambio entre todas. Es decir, al pasar de una moneda a otra esta sigue valiendo lo mismo. La matriz correspondiente al caso actual con 3 divisas será

$$M = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

y su pseudografo asociado

En este caso se puede observar que ningún ciclo de divisas podrá generar algún tipo de ganancia, ya que al pasar por cualquier divisa y volver a la divisa inicial esta conservará



su valor original, de manera que no habrá ganancia, pero tampoco pérdida. Por lo tanto podremos concluir que no existirá oportunidad de arbitraje.

Ejemplo No. 3:

Como último ejemplo tendremos en cuenta el grafo en el cual $\forall$ divisa d_i con $0 \leq i \leq 3$ exista un ciclo de divisas que empieza y termina en d_i tal que haya posibilidad de arbitraje. Es decir, si tenemos la matriz

$$M = \begin{bmatrix} 1 & 1{,}41 & 3{,}14 & 1{,}6180 \\ 1{,}01 & 1 & 2,02 & 3,03 \\ 4 & 4 & 1 & 4 \\ 1{,}03 & 1{,}06 & 1{,}09 & 1 \end{bmatrix}$$

Y como podemos notar según lo mencionado previamente, habrá solución posible al problema y será más de una. Es decir, tomando los valores de cambio, cualquier ciclo de la forma C = $\{d_i, d_j, ..., d_i\}$ cumplirá que genere arbitraje ya que todas las divisas ya sea para la ida o para la vuelta tienen una ganancia al realizar cualquier cambio debido a que todas las aristas que unen divisas son mayores a 1. Y como el problema se trata de responder si existe posibilidad y devolver alguno de estos ciclos, podremos devolver cualquiera de ellos y esa será la solución.

7. Algoritmos y soluciones

En la presente sección se explicará la idea detrás de cada algoritmo, como la justificación de la complejidad de cada uno. Se usará pseudocódigo para explicar los mismos.

Para la resolución de nuestro problema utilizaremos las siguientes definiciones:

Definicion: el **camino mínimo** entre dos vértices (o nodos) es aquel camino tal que la suma de los pesos de las aristas que lo constituyen es mínima respecto de otros caminos.

Definicion: el **Circuito Mínimo Multiplicativo** de un nodo v será aquel que cumpla lo siguiente:

Si de avanzar por el camino que inicia en v, multiplicando por los factores multiplicativos(pesos de las aristas), al finalizar , se consiguiera volver a v con un valor menor al inicial, este se considerara un **Camino Mínimo Multiplicativo**

Entonces, en nuestro problema, diremos que habrá solución si al aplicar nuestros algoritmos, al regresar por algún circuito, el resultado del circuito mínimo de un nodo a sí mismo es mayor que 1.

Para la resolución del problema consideraremos dos opciones/variantes, las cuales utilizaran cada una la idea de un algoritmo de camino mínimo conocido.

7.1. Primer Aproximación (Bellman-Ford)

7.1.1. El Algoritmo

Para la presente aproximación del problema se tendrá en cuenta el algoritmo de Bellman-Ford con algunas modificaciones a fin de ser útil con el problema de arbitraje.

Normalmente este algoritmo se usa para calcular los caminos mínimos de un nodo en específico hacia todos los nodos en un grafo. En nuestro caso, no queremos calcular eso sino algún **Camino Mínimo Multiplicativo** que haya en el grafo. Por lo tanto se usarán las herramientas de detección presentes en el algoritmo original y la idea general del mismo para poder devolver el ciclo de divisas que genera el arbitraje. Es decir, como este algoritmo calcula el camino mínimo, si cambiaramos las sumas por multiplicaciones podríamos calcular el camino mínimo multiplicativo. Y en este caso, lo que haremos es cambiar el menor por un mayor y lo que obtendremos por el algoritmo será un camino multiplicativo que "maximice".

Para facilitar la comprensión del mismo se introducirá el pseudo-código del algoritmo que resuelve una parte intermedia del problema.

```
BellmanFordArbitraje(M, s)
    n = rows(M)
    distancias(n, -INFINITO)
    padre(n, -1)
    ciclo = vacio()
    distancias[s] = 1
    cambio = true
    finished = false

    for(int k = 1; k < n+1 && cambio && !finished; k++)
        cambio = false
        for(int i = 0; i < n && !finished; i++)
```

```
            for(int j = 0;j < n && !finished; j++)
                if distancias[i] * M[i][j] > distancias[j]:
                    cambio = true
                    distancias[j] = distancias[i] * M[i][j]
                    padre[j] = i
                    if distancias[s] > 1: finished = true


    if (!cambio || distancias[s] <= 1)
        return ciclo

    comienzoCiclo =  s;
    current = comienzoCiclo
    agrego current a ciclo
    mientras current != comienzoCiclo
        agrego current a ciclo
        current = padre[current]
    agrego current a ciclo
    return ciclo
```

Para explicar el funcionamiento de este algoritmo se tendrá en cuenta la utilidad de las siguientes variables:

- distancias: en la posicion s se guardará la distancia del nodo s a sí mismo por algún camino/ciclo.
- padre: se utilizará para reconstruir el ciclo que involucre a s en caso de exitir uno.
- tanto cambio como finished serán flags que servirán para terminar antes con el triple ciclo.
- ciclo: será el ciclo que genere el arbitraje empezando desde la divisa s.

Con todo lo anteriormente dicho, podemos empezar a explicar el funcionamiento del algoritmo:

Si observamos el triple ciclo for, podemos ver que este se parece un poco al del algoritmo de Bellman-Ford sobre Matrices de Adyacencia, con las siguientes diferencias:

-En la versión original el for más interno de todos solo itera sobre los vecinos de i, pero en particular, como todas las divisas se pueden intercambiar entre si, el grafo será completo y los vecinos de i serán todos los nodos.

-Al igual que en Bellman-Ford usaremos la idea de relajar ejes y reemplazar valores en caso de encontrar un camino mejor utilizando el nodo i como intermediario, la diferencia en este caso será que la distancia de un nodo a sí mismo tambien podrá mejorar, y esto será lo que ocurra en caso de existir posibilidad de arbitraje: Se encontró un camino (en este caso ciclo) que al pasar por cierta/s divisa/s genere un valor mayor al que se encontraba originalmente. Es decir, de relajar ejes también podrá resultar que la distancia de un nodo a si mismo mejore, cosa que no ocurre en el algoritmo original para caminos mínimos, (salvo que hayan ciclos negativos) y con mejor en este caso nos referimos a que aumente. Dicho esto podemos observar que el ciclo de divisas (empezando y terminando en s) que genere arbitraje será identificado en el triple ciclo de nuestra función intermedia mediante el mecanismo de padres. Una vez terminada la ïdentificaciónza sea por que hizo todas las iteraciones o por el valor de alguna flag, se procederá a armar dicho ciclo.

Como se mencionó antes, padre servirá para reconstruir el ciclo que genere arbitraje luego de ejecutar este triple ciclo. Notar que los flags mencionados previamente cumplirán con las siguientes funciones específicas:

- cambio será el encargado de verificar que el vector de distancias cambie por cada iteracion en k, de no ocurrir esto el ciclo terminará. Esto nos garantizará que nuestra implementación sea la optimizada, en la cual no se realizarán iteraciones de más. Si bien esto no altera la complejidad teórica, ayudará en cuanto a la complejidad práctica.
- finished al igual que cambio ayudará a reducir la cantidad de iteraciones en caso de ocurrir lo siguiente: si la distancia del nodo s a si mismo es mayor a 1. ¿Y por qué esto sirve? Simple, esto lo que nos quiere decir es que el algoritmo encontró algún ciclo que empieza y termina en s por lo tanto no es necesario seguir buscando otro ciclo. Al igual que antes, esto no ayuda en cuanto a complejidad teórica pero si en cuanto a práctica.

Por lo tanto podemos decir que el algoritmo servirá para calcular el problema de arbitraje si la divisa pasada por parámetro forma parte de un ciclo que aumente el valor al volver, el cual será el resultado. Es decir, si quisiéramos resolver el problema sin importar desde cuál nodo se comienza el ciclo deberíamos repetir la idea para todos los nodos, lo cual será realizado por la función ArbitrajeBF:

```
problemaArbitrajeBF(M)
    ciclo = vacio()
    n = M.rows()
    for int i = 0; i < n; ++i
        ciclo = bellmanFordArbitraje(M, i)
        Si ciclo no es vacio, salgo del for.

    return ciclo
```

Notar también que nuestro algoritmo devuelve el primer ciclo que encuentre, en caso de haber uno, empezando desde el nodo 0, lo cual puede verse en nuestro pseudo-código en color rojo.

Como efectivamente realizamos esto para cada nodo (divisa) del grafo, y fue probado que resuelve el problema en particular para un nodo, podremos decir que nuestro algoritmo final resuelve el problema de **manera exacta** para cualquier divisa del grafo.

7.1.2. Análisis de Complejidad

Para nuestro análisis de complejidad podremos reutilizar complejidades teóricas de los algoritmos subyacentes de camino mínimo que usamos para resolver los problemas. En particular, podemos ver que nuestro triple ciclo del algoritmo BellmanFordArbitraje tendrá complejidad $\mathcal{O}(n^3)$ como el del algoritmo original sobre matrices de adyacencias y que lo demás (asignaciones, rearmar el ciclo) tendrá a lo sumo complejidad $\mathcal{O}$(n) por lo tanto podemos decir que:

Complejidad Teórica de BellmanFordArbitraje: $\mathcal{O}(n^3)$

Y como realizamos esto una vez por cada nodo en problemaArbitrajeBF, y siendo nuestro peor caso el de que el ciclo no exista o este exista pero esté en el nodo n-1, podemos concluir que la complejidad teórica de nuestra primer aproximación al problema será:

$$\text{T(n)} = \mathcal{O}(n^3)\text{*n} = \mathcal{O}(n^4)$$

7.2. Segunda Aproximación (Floyd-Warshall)

7.2.1. El Algoritmo

La idea inicial es, igual que en el de Bellman Ford, intentar modificar el algoritmo de Floyd para que .en vez de sumar, multiplique".
El algoritmo de Floyd se basa en el concepto de que "si el camino más corto de A a B pasa por C, entonces la distancia de A a B es la distancia de A a C más la distancia de C a B".
Nosotros ahora queremos encontrar el CMM (Camino Mínimo Multiplicativo) por lo tanto queremos demostrar que "si el CMM de A a B pasa por C, entonces la DMM de A a B es la DMM de A a C multiplicada por la DMM de C a B".
Donde DMM = Distancia Mínima Multiplicativa que es la mínima distancia de multiplicando el peso de las aristas del CMM.
Supongamos que el CMM(A,B) pasa por C.
Tenemos el CMM(A,C) con X aristas que llamaremos H_i con $0 \leq i < X$.
Tenemos el CMM(C,B) con Y aristas que llamaremos Q_i con $0 \leq i < Y$.
Tenemos el CMM(A,B) con Z aristas que llamaremos P_i con $0 \leq i < Z$.
DMM(A,C) = $\prod H_i$
DMM(C,B) = $\prod Q_i$
DMM(A,B) = $\prod P_i$
Queremos probar que DMM(A,B) = DMM(A,C) * DMM(C,B).
Esto es equivalente a $\prod P_i = \prod H_i * \prod Q_i$
También es equivalente a $\prod P_i = \prod (H \cup Q)_i$
Por lo tanto queremos probar que $P = H \cup Q$
Suponemos que esto es falso y nos quedan 3 casos:
1) $P \subseteq H \ \& \ P \nsubseteq Q$
2) $P \nsubseteq H \ \& \ P \subseteq Q$
3) $P \nsubseteq H \ \& \ P \nsubseteq Q$
Caso 1:
Si esto es verdad entonces tomamos en el tramo C-B del CMM(A,B) y va a contener una distancia d.
Si $d \leq DMM(C, B)$ es absurdo por definicion de DMM!
Si $d > DMM(C, B)$ entonces se puede reemplazar el tramo C-B del CMM(A,B) por CMM(C,B) y obtendríamos una longitud menor del circuito, lo que probaría, pero esto es absurdo porque este ya era el CMM por lo tanto tenía distancia mínima.
Caso 2:
Es análogo al Caso 1.
Caso 3:
Es como el caso 1 y 2 juntos.

Por lo que queda demostrado por absurdo que $P = H \cup Q$.
Por lo tanto la idea del algoritmo de Floyd puede utilizarse para resolver el problema del CMM.
También vamos a cambiar la comparación para guardarnos los máximos en vez de los mínimos así podemos obtener facilmente el resultado del ejercicio ya que el objetivo es buscar un ciclo positivo.
Sería exactamente lo mismo si invertimos la matriz y ejecutamos el clásico algoritmo de mínimos. Pero esto nos ayuda a evitar errores de precisión al invertir la matriz.

Ahora vamos a generar un pseudocódigo para analizar los cambios que tendrían que ha-

cerse:

```
FloydWarshallModificado(M)
    padres = inicializar_padres()

    Poner diagonal de M en 1

    for k = 0; k < n; k++
        for i = 0; i < n; i++
            for j = 0; j < n; j++
                dt = distancias[i][k] * distancias[k][j]
                if distancias[i][j] < dt
                    distancias[i][j] = dt
                    padres[i][j] = padres[k][j]
                endif
            endfor
        endfor
    endfor

    Recorremos la diagonal para ver si alguno quedo mayor que uno:
        Reconstruimos el ciclo a partir de ese con la matriz de padres
        Si lo logramos: ganamos! encontramos un ciclo positivo.

    Invertimos el ciclo (porque lo reconstruimos al reves)
```

Así es la primera idea de como sería el algoritmo. Es un típico algoritmo de Floyd-Warshall, recorre todos los nodos y para cada uno, se fija para todas las distancias si alguna se puede mejorar haciendo pasar al camino por ese nodo. La única diferencia es que ahora estamos usando multiplicación en vez de suma para poder encontrar las DMM y cambiamos la comparacion para que ahora guarde los mayores asi encontramos el ciclo positivo.
Luego de correr Floyd-Warshall obtenemos la matriz de distancias y la matriz de padres, pero... y el problema?...
Para recordar: Tenemos que obtener el ciclo.
Por lo tanto tenemos que recorrer la matriz de distancias y fijarnos si es que algún elemento de la diagonal dio mayor que 1.
Nosotros empezamos con todos 1s en la diagonal, esto quiere decir que si alguno terminó mayor eso significa que hay un ciclo que lo hizo "mejorar"de su valor original para ir a sí mísmo.
Si encontramos uno de estos podemos recorrer la matriz de padres comenzando en ese nodo y hacia atras para encontrar todo el ciclo.
Puede suceder que cuando estás recorriendo hacia atras la matriz de padres, te encuentres con otro ciclo antes de volver al nodo inicial del ciclo que estás buscando, tomamos la decision de que si esto sucede vamos a tirar a la basura el ciclo que encontramos hasta ahora y seguir recorriendo nodos de la diagonal hasta encontrar el nodo inicial del ciclo más chico y así devolver ese.
Finalmente damos vuelta el ciclo obtenido ya que lo reconstruimos de atrás para adelante y lo devolvemos ya que es el resultado del ejercicio.

7.2.2. Análisis de Complejidad

Al igual que en Bellman-Ford, podremos reutilizar la complejidad teórica del algoritmo subyacente, el cual es sabido es $\mathcal{O}(n^3)$, y diremos que esta será la complejidad final ya que después del triple ciclo no hay nada que tenga mayor complejidad. Es decir:

Complejidad del triple ciclo = $\mathcal{O}(n^3)$.

Complejidad de llenar la matriz distancias y rearmar el ciclo en caso de existir uno = $\mathcal{O}(n^2)$.

Por lo tanto podemos concluir:

Complejidad Total del algoritmo = $\mathcal{O}(n^3)$.

8. Experimentación, Análisis y Conclusiones

8.1. Experimento de Complejidad

Para este experimento se compararán los tiempos de ejecución de cada algoritmo con su complejidad teórica en el peor caso.

A continuación se detallará el peor caso de cada uno de los algoritmos: Para nuestra primera aproximación (Bellman-Ford) diremos que el peor caso será aquel en el cual el ciclo que genere arbitraje sea el que involucre al último nodo pero no a los primeros n-2 nodos. Es decir, que el ciclo de divisas sea $d_n - d_{n-1} - d_n$. Esto se deberá a que nuestro algoritmo hace una búsqueda secuencial para ver si la divisa i forma parte de un ciclo, con i = 0....$n-1$. Y en particular para nuestra segunda aproximación, diremos que su peor caso = mejor caso = caso promedio por lo tanto probaremos varios casos para ver que cumpla con la complejidad propuesta.

En particular consideraremos las siguientes instancias en base a las cuales se realizará la experimentación: Todos unos: en esta instancia todos los valores de la "matriz de cambios"serán 1, es decir: $d_{i,j} = 1 \ \forall$ i,j. Por lo tanto *no* habrá oportunidad de arbitraje. Ciclo Cn: para esta instancia tendremos en cuenta el ciclo que contenga las n divisas como lo indica el nombre. Ciclo $d_n - d_{n-1} - d_n$: como se mencionó antes, esta será el peor caso de nuestro primer algoritmo.

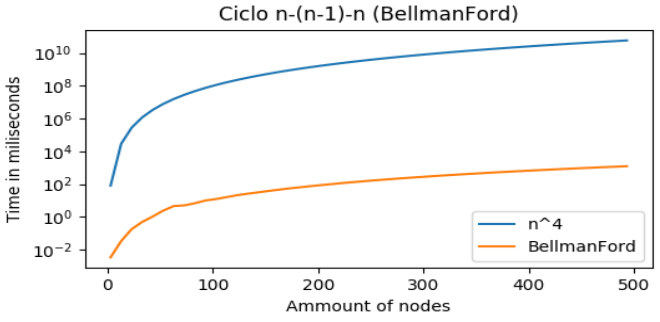
Notar que para toda la experimentación de complejidad se tomarán los siguiente valores para las variables involucradas en el problema:

- El valor de n = cantidad de nodos/divisas será creciente empezando en 3 hasta llegar a 500.
- En los experimentos en los que se compare un algoritmo con su complejidad teórica se usará escala logarítmica a fines de que las diferencias de tiempos sean más apreciables.
- El peso de las aristas que no formen parte del ciclo será demasiado chico(Ej: 0.2, 0.0).

Para la experimentación comenzaremos con el primer algoritmo: problemaArbitrajeBF por lo que querremos formular lo siguiente:

- **Hipótesis BF:** El algoritmo que resuelve el problema de arbitraje que usa la idea del algoritmo de Bellman-Ford como intermedia tendrá una complejidad práctica a lo sumo igual que la de su complejidad teórica: $\mathcal{O}(n^4)$

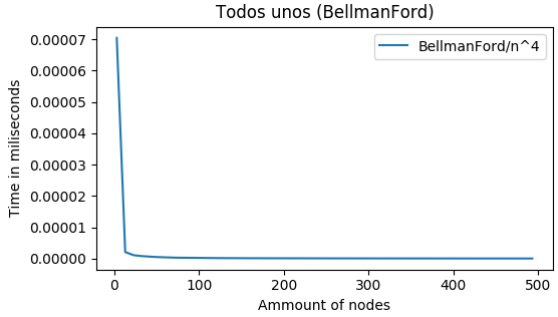
Empezaremos con la instancia Ciclo $d_n - d_{n-1} - d_n$:

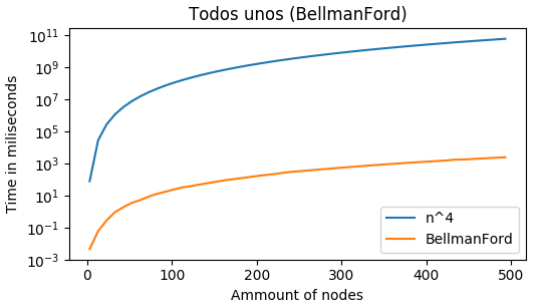


Y lo que podemos notar hasta ahora es que la cota propuesta es muy grosera. Entonces, si observamos los siguientes graficos correspondientes a la instancia Todos unos:

Figura 1: El primer gráfico (izquierda) corresponde a el tiempo de ejecución dividido por la complejidad teórica.

Figura 2: El segundo corresponde a las comparaciones en escala logarítmica.

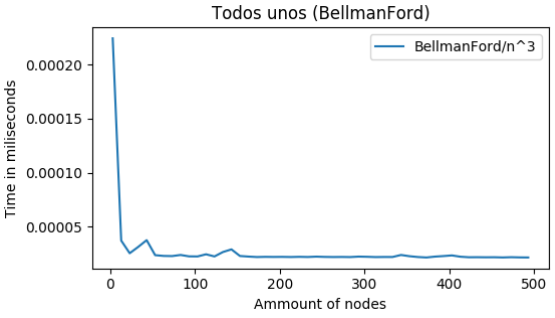


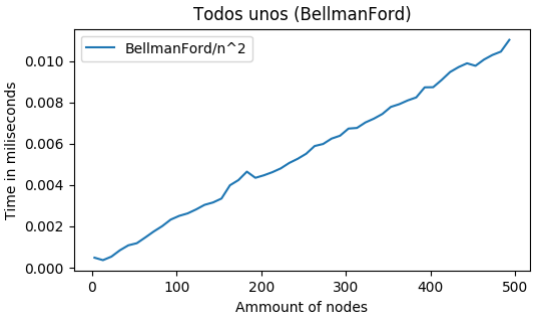


Podremos llegar a dos análisis: como mencionamos antes, la cota será muy grosera y esto se puede notar en los dos gráficos pero más que nada en el primero, ya que de dividir nuestro tiempo de ejecución actual por la complejidad propuesta resulta una función que tiende a cero asintóticamente, lo cual nos indica que efectivamente $\mathcal{O}(n^4)$ será una cota superior.

Y lo segundo que podemos mencionar es que a diferencia de lo afirmado previamente, el caso en el que todas las divisas tienen tasa de cambio 1 entre sí, es peor que el caso Ciclo $d_n - d_{n-1} - d_n$.

En cuanto a lo primero, si vemos los siguientes gráficos en los cuales dividimos el tiempo de ejecución del caso Todos unos por n^3 y n^2 respectivamente:



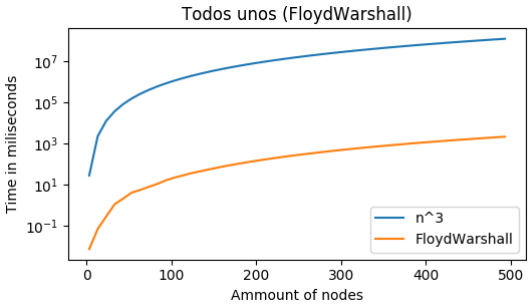


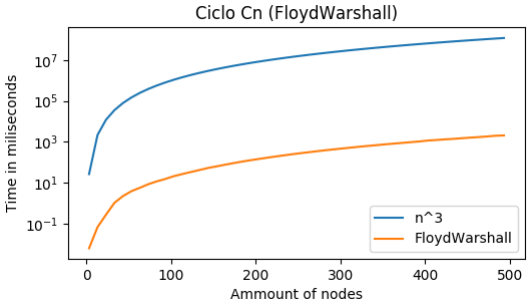
Podremos ver que recién cuando dividimos por n^2 tendrémos una función lineal que inclusive tendrá una pendiente muy chica lo cual confirmaría que $\mathcal{O}(n^4)$ es una cota muy grosera pero que sigue sirviendo como cota superior. Por lo tanto podemos concluir: nuestro algoritmo cumple que en la práctica, su complejidad teórica es una cota superior.

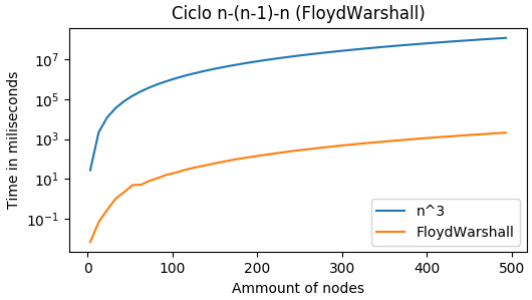
Ahora, formularemos la hipótesis correspondiente a nuestra segunda aproximación:

- **Hipótesis FW:** El algoritmo que resuelve el problema de arbitraje que usa la idea del algoritmo de Floyd-Warshall como intermedia tendrá una complejidad práctica a lo sumo igual que la de su complejidad teórica: $\mathcal{O}(n^3)$, y la complejidad práctica de todos los casos (peor, mejor, promedio) será muy parecida.

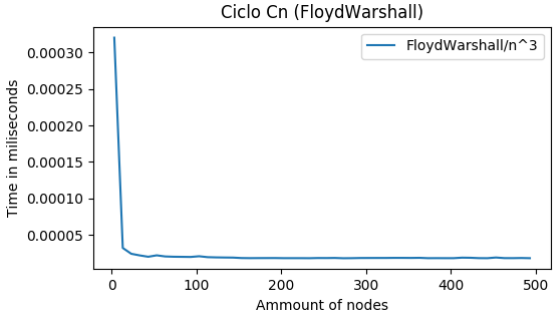
Observación: Para este algoritmo usaremos todas las instancias previamente mencionadas.
Entonces, empezando con los gráficos con escala logarítmica:

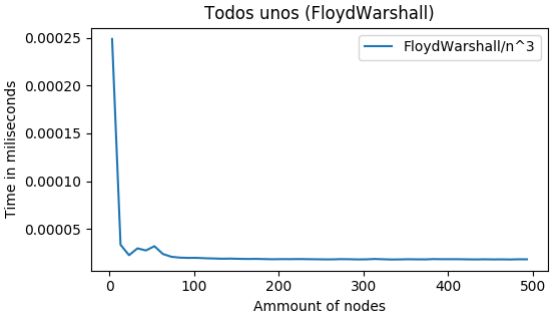


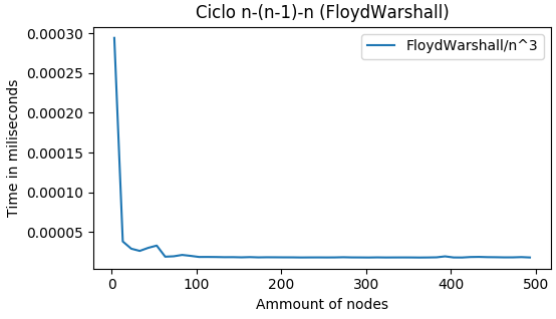




Si observamos todos estos gráficos podemos observar que distintos casos de ejecución del mismo algoritmo las cuales, algunas tienen solución (Ciclo $d_n - d_{n-1} - d_n$ y Ciclo C_n) y otras no (Todos unos) tendrán curvas muy similares, lo cual validaría la segunda parte de nuestra hipótesis.
Además podemos notar que aquí también la complejidad teórica será una cota. Para poder ver esto mejor veremos los siguientes gráficos:







De lo que podremos concluir: al igual que con nuestro anterior experimento, la complejidad teórica será una cota superior grosera, y esto hará que al dividir nuestro tiempo de ejecución por la misma resulte una función que tiende a cero. Es decir, n^3 tenderá a infinito más rápido en tiempo de ejecución que nuestra función propuesta, lo cual nos permite asegurar que la complejidad teórica será una buena cota superior.