



DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico III

Algoritmos y Estructuras de Datos III
Primer Cuatrimestre de 2015

Integrante	LU	Correo electrónico
Aldasoro Agustina	86/13	agusaldasoro@gmail.com
Noriega Francisco	660/12	frannoriega.92@gmail.com
Zimenspitz Ezequiel	155/13	ezeqzim@gmail.com
Zuker Brian	441/13	brianzuker@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>

Resumen

El objetivo de este trabajo es plantear soluciones a la problemática de *Conjunto Independiente Dominante Mínimo*, para lo cual se desarrolla un algoritmo exacto que calcula la solución óptima y también heurísticas con el fin de abordar la misma problemática.

El código de este trabajo práctico presenta una función `main` que permite correr cualquiera de los algoritmos desarrollados bajo el mismo input e incluso hacerlo veces consecutivas. Cada uno recibe parámetros necesarios para reutilizar el código. Búsqueda Local utiliza Greedy para generar instancias iniciales, GRASP aprovecha a la búsqueda local y una adaptación del greedy. Ver sección 2.

Para compilar se usa `g++ -o main correrCIDM.cpp -std=c++11`

Esta flag se añade con el fin de poder utilizar funciones de medición para los tiempos de ejecución dentro de la experimentación.

Índice

1. Heurística de Búsqueda Local	3
1.1. Explicación	3
1.1.1. Elección de Solución Inicial	3
1.1.2. Elección de Vecindad	3
1.2. Complejidad Temporal	5
1.2.1. dameParesVecinosComun	5
1.2.2. dameTernasVecinasComun	7
1.2.3. localCIDM	8
1.3. Experimentación	11
1.3.1. Análisis de tiempos de ejecución	11
1.3.2. Contrastación empírica de la complejidad	17
1.3.3. Comparación soluciones Local vs Exacto	18
1.3.4. Elección de versión óptima	21
2. Anexo	24
2.1. Ejecución de los métodos	24
2.2. Generación de casos de test	24

1. Heurística de Búsqueda Local

Un algoritmo de Búsqueda Local consiste en dos simples pasos: elegir una solución inicial y luego, **Aca explicarlo bien, esta horrible** iterar

modificarla (“mejorándola”), reemplazándola paso a paso con distintas soluciones que pertenezcan a la vecindad de la misma.

Para cada solución factible $s \in S$ se define $N(s)$ como el conjunto de “soluciones vecinas” de s . Un procedimiento de búsqueda local toma una solución inicial s e iterativamente la mejora reemplazándola por otra solución mejor del conjunto $N(s)$, hasta llegar a un óptimo local.

Sea $s \in S$ una solución inicial

Mientras exista $s \in N(s)$ con $f(s) > f(s')$

$s \leftarrow s'$

1.1. Explicación

Considerando el problema a tratar, establecimos nuestros criterios para encontrar las soluciones iniciales y las vecindades.

1.1.1. Elección de Solución Inicial

Al momento de seleccionar la solución Inicial, determinamos dos criterios.

Criterio I Solución Inicial: Golosa

Se realiza una ejecución del algoritmo Goloso de la Sección ??.

Esto quiere decir, se ordenan los nodos por grado de manera decreciente. Se eligen los nodos de a uno (de mayor a menor), de modo que al elegir un nodo se descartan sus vecinos para sus futuras elecciones.

Criterio II Solución Inicial: Secuencial

Los nodos al ser ingresados como parámetro del algoritmo tienen como identificador un número entre 0 y $n - 1$. El orden que vamos a utilizar para recorrerlos es el que haya sido dado cuando fueron ingresados como parámetro.

Lo primero que realizamos es tomar al nodo 0 y considerarlo parte de la solución. Se descartan todos los nodos vecinos a él y se continúa el proceso con el nodo que tenga menor número de *id*.

De este modo se forma un conjunto solución tal que en cada paso añade al nodo disponible que tenga su identificador número menor.

1.1.2. Elección de Vecindad

Dada una solución al problema, se establece un conjunto de soluciones “similares” denominadas *vecinas*. Los criterios para elegir esta vecindad pueden variar.

Criterio I Vecindad

El primer criterio elegido es, a partir de una solución, quitarle dos nodos y agregarle uno que no haya sido contenido.

Para ello, se prueban todas las combinaciones de pares de nodos dentro del conjunto posibles y se considera a los nodos que tienen ambas como vecinos. Si al sacar este par y agregar el nuevo nodo, se obtiene un conjunto Independiente Dominante Mínimo, se actualiza el conjunto solución.

Criterio II Vecindad

El segundo criterio es similar al anterior, sólo que ahora consideramos quitar tres nodos y agregar uno.

Se consideran todas las combinaciones posibles de grupos de tres nodos dentro del conjunto solución inicial y se prueba con los nodos que sean vecinos de todos ellos si forman un conjunto solución.

Las opciones que elegimos son: 2, 3, 4 y 5 y son BLABALLABA

1.2. Complejidad Temporal

Este algoritmo llama, según la vecindad a ejecutar, a una de las siguiente dos funciones, que dominan la complejidad del ciclo.

1.2.1. dameParesVecinosComun

Funciones usadas:

listaAd::dameVecinos
push_back
size

Dado un conjunto de nodos, se buscan todas las combinaciones de pares de nodos posibles. Luego, para cada par de nodos (i,j) se recorren: la lista de adyacencia de i y la de j . Por cada elemento que pertenezca a las dos listas, se añade al vector `vecinosEnComun` dentro de la estructura `pares`.

```
struct vecinosEnComun{  
    unsigned int nodoA;  
    unsigned int nodoB;  
    vector<unsigned int> vecinosComun;  
};
```

```
for int i = 0; i < optimo.size(); ++i do  
    for int j = i+1; j < optimo.size(); ++j do  
        list<unsigned int>* vecinosA = adyacencia.dameVecinos(optimo[i]);  
        list<unsigned int>* vecinosB = adyacencia.dameVecinos(optimo[j]);  
        vecinosEnComun par;  
        par.nodoA = optimo[i];  
        par.nodoB = optimo[j];  
        list<unsigned int>::iterator itVecinosA = vecinosA->begin(), itVecinosB =  
            vecinosB->begin();  
        while itVecinosA != vecinosA->end() and itVecinosB != vecinosB->end() do  
            if *itVecinosA == *itVecinosB then  
                par.vecinosComun.push_back(*itVecinosA);  
                itVecinosA++;  
                itVecinosB++;  
            else  
                if *itVecinosA > *itVecinosB then  
                    itVecinosB++;  
                else  
                    itVecinosA++;  
                end  
            end  
        end  
        if par.vecinosComun.size() > 0 then  
            pares.push_back(par);  
        end  
    end  
end
```

Para elegir todos los pares posibles de nodos en el conjunto óptimo, se recorre mediante dos fors. El primero itera i desde 0 hasta el último elemento y el segundo desde i hasta el último elemento.

De este modo, cada par de nodos se recorre una única vez. Ya que es lo mismo el par (i,j) que (j,i).

Dentro de los *fors* anidados, se crea una estructura `vecinosEnComun` **par** donde el nodoA es i y el nodoB es j.

Para poder armar la lista `vecinosComun` (miembro de la estructura `vecinosEnComun`), se iteran las listas de adyacencia con `itVecinosA` (i) e `itVecinosB` (j).

Como ambas listas fueron ordenadas antes de invocar a la función `dameParesVecinosComun`, es posible encontrar elementos en común recorriendolas secuencialmente de manera simultánea.

Se procede de manera simple, si `nodo(itVecinosA)` es igual a `nodo(itVecinosB)` entonces se añade el nodo actual a la lista `vecinosComun`.

En caso contrario, se avanza el iterador que sea menor.

Si concluída la iteración de las dos listas de adyacencia, la lista `vecinosEnComun` posee al menos un elemento; entonces se agrega **par** a la solución.

Dado un par (i,j), la complejidad de recorrer ambas listas de adyacencia es de: $O(\text{grado}(i) + \text{grado}(j))$. Cada par se recorre una única vez. Por lo tanto, los dos *fors* anidados van a iterar (considerando a (n-1) como el último nodo'):

Cuando sea el par de nodos (0,1) : $\text{grado}(0) + \text{grado}(1)$

Cuando sea el par de nodos (0,2) : $\text{grado}(0) + \text{grado}(2)$

...

Cuando sea el par de nodos (0,n-1) : $\text{grado}(0) + \text{grado}(n-1)$

Cuando sea el par de nodos (1,2) : $\text{grado}(1) + \text{grado}(2)$

Cuando sea el par de nodos (1,3) : $\text{grado}(1) + \text{grado}(3)$

...

Cuando sea el par de nodos (1,n-1) : $\text{grado}(1) + \text{grado}(n-1)$

...

Cuando sea el par de nodos (n-2,n-1) : $\text{grado}(n-2) + \text{grado}(n-1)$

Se puede apreciar que el grado de cada nodo se suma (n-1) veces. Por lo tanto, al sumar las complejidades da un total de:

$$\text{grado}(0)*(n-1) + \text{grado}(1)*(n-1) + \dots + \text{grado}(n-1)*(n-1)$$

lo que es equivalente a:

$$[\text{grado}(0) + \text{grado}(1) + \dots + \text{grado}(n-1)]*(n-1)$$

La complejidad en peor caso se obtiene cuando los grados de todos los nodos son máximos, por lo tanto se trata de un grafo completo. Donde vale que $2 * \# \text{ejes} = \text{grado}(0) + \text{grado}(1) + \dots + \text{grado}(n-1)$.

Por consecuencia, la complejidad de esta función es de:

$O(2 * \#ejes * (n-1))$ lo que equivale a $O(\#ejes * n)$ que pertenece a $O(n^3)$ ya que la mayor cantidad de ejes que puede tener un grafo es $((n-1)n)/2$

1.2.2. dameTernasVecinasComun

La función dameTernasVecinasComun funciona de manera análoga a la descripta en el inciso 1.2.1.

Se va a encargar de armar tomar de todas las maneras posibles tres nodos del conjunto pasado por parámetro.

Luego, obtener los nodos (si existe) que sean vecinos de los tres.

De esta manera, recorre el conjunto con tres *fors* tal que cada tupla la recorre una sola vez.

En este caso el cálculo de cada iteración, dada la tupla (i,j,k) , será de $\text{grado}(i) + \text{grado}(j) + \text{grado}(k)$.

Dada la tupla $(0,1,2)$ el costo será: $\text{grado}(0) + \text{grado}(1) + \text{grado}(2)$

Dada la tupla $(0,1,3)$ el costo será: $\text{grado}(0) + \text{grado}(1) + \text{grado}(3)$

...

Dada la tupla $(0,1,n-1)$ el costo será: $\text{grado}(0) + \text{grado}(1) + \text{grado}(n-1)$

Dada la tupla $(0,2,3)$ el costo será: $\text{grado}(0) + \text{grado}(2) + \text{grado}(3)$

...

Dada la tupla $(0,n-2,n-1)$ el costo será: $\text{grado}(0) + \text{grado}(n-1) + \text{grado}(n-2)$

Dada la tupla $(1,2,3)$ el costo será: $\text{grado}(1) + \text{grado}(2) + \text{grado}(3)$

...

Dada la tupla $(n-3,n-2,n-1)$ el costo será: $\text{grado}(n-3) + \text{grado}(n-2) + \text{grado}(n-1)$

En el peor caso, el grado de todos los nodos es de n (grafo completo). Por lo tanto, el costo de cada iteración es de $O(3n)$.

La cantidad de ternas que se pueden formar es de $(n(n - 1)(n - 2))/6$, equivale a decir que va a iterar $O(n^3)$ veces con costo $O(3n)$ cada vez.

Dando una complejidad de $O(n^4)$.

1.2.3. localCIDM

Como primera medida, ordena todas las listas de adyacencia: listaAd::ordenar, esto toma $O(n^2 \cdot \log(n))$, para cada nodo (n) ordenar su lista de adyacencia (en el caso del grafo completo, tendrán $(n - 1)$ vecinos, y ordenarlos toma $O(n \cdot \log(n))$)

Para las **ejecuciones 3 y 5**, el parametro greedy esta en true, por lo tanto comienza con una solucion inicial golosa. Por lo tanto invoca a la funcion greedyCIDM() la cual tiene complejidad $O(n^2)$ explicada en el inciso ??.

Para las **ejecuciones 2 y 4**, la solucion inicial es secuencial. Esto quiere decir que para obtener una primera solucion al problema se arma un vector nodos con la cantidad de nodos, donde en la posicion i se encuentra el nodo i.

Se recorre secuencialmente este arreglo (desde la posicion cero) de modo que se añade el nodo actual a la solucion y se elimina del vector nodos.

Luego, se borran tambien del vector a los vecinos del nodo actual.

En la siguiente iteracion se tienen $n - 1 - (\text{grado}(0))$ elementos en el vector nodos.

Lo cual, en el peor caso, seria n-1 donde grado(0)=0.

Considero la notacion vecinos(i) como la cantidad de nodos pertenecientes al vector nodos durante la iteracion i que sean adyacentes al nodo i.

En la iteracion i, el vector va a contener $n - 1 - \text{grado}(0) - \dots - 1 - \text{vecinos}(i)$

Donde en el peor caso, tambien, debera ser vecinos(i) con valor minimo. Por consecuente, el peor caso es un grafo donde cada nodo es una componente conexa trivial, es decir que no existen ejes.

En el peor caso, itera n veces ya que el tamaño del vector disminuye solo en una unidad por iteracion.

El costo de las operaciones por iteracion es $O(n)$.

Las funciones usadas son:

`push_back` (costo $O(n)$ amortizado)

`listaAdy::sonVecinos` (costo $O(\min(\text{grado}(nodoA), \text{grado}(nodoB)))$)

Por lo tanto esta seccion es del orden de $O(n^2)$

A continuacion, se introducen optimizaciones del algoritmo.

- En primer lugar, si la solucion optima actual posee tamano 1 no va a encontrarse una solucion mejor, por consiguiente se devuelve. (Costo $O(1)$)
- En segundo lugar, si la solucion optima actual posee tamano 2 la unica solucion que puede ser mejor es la que posee un solo nodo. Se chequea si existe una solucion con un solo nodo (costo $O(n)$). Si existe, la solucion optima es la de un solo nodo y se devuelve sino era la solucion con dos nodos. Funciones usadas: `assign`, `listaAdy::gradoDeNodo` y `listaAdy::cantNodos` (todas con costo $O(1)$)
- En tercer lugar, si la solucion optima actual posee tamano n debe ser la solucion que se retorne. Ya que la unica manera de que todos los nodos sean parte del conjunto solucion al problema es que

cada uno sea una componente conexa, es decir no existan ejes. A continuación, debe ser la solución devuelta. (Costo $O(1)$)

Si la ejecución del algoritmo no entró en ninguno de los casos citados, se prosigue ordenando al vector de solución óptima. (Costo $O(n \log n)$), en el peor caso tiene $n - 1$ elementos)

Ahora se prosigue con las iteraciones por vecindades.

En los casos de **ejecución 2 y 3**, se pasa por parámetro el valor de vecindad == true. Se ejecuta la función dameParesVecinosComun (vista en 1.2.1), es decir por cada iteración se querrá tomar un par de nodos tal que sea posible quitarlos y agregar un vecino de ambos al conjunto solución y se obtenga una solución con un nodo menos.

Se iterará en un while hasta que la variable hayCambiosHechas sea false (complejidad $O(n)$ explicada más adelante).

Lo primero que se hace es ejecutar `dameParesVecinosComun()` (complejidad $O(n^3)$).

Si esta función no devolvió ningún par, se debe salir del ciclo.

Se ejecuta otro while, mientras haya pares sin ser visitados (de los obtenidos) (complejidad $O(n \cdot (n - 1)/2) = O(n^2)$).

Como lo que se quiere hacer es borrar nodos del conjunto solución, se hace en una copia porque a priori no se sabe si conducirá a un conjunto que sea solución.

Se borra del conjunto solución al par actual (i,j) que por el modo en que armamos los pares siempre se cumple que $i < j$. Esto tiene un costo de $O(n)$ ya que itera la solución hasta que encuentre j y en el medio elimina i. Se borra con el operador `erase` de iterador (complejidad $O(3n)$ en el peor caso).

Ahora resta ver si al agregar algún nodo en común se forma un conjunto solución que sea Independiente Maximal. Recorreremos la lista de nodos que tienen en común i y j para ver si al agregar alguno de ellos al conjunto solución, el conjunto obtenido cumple ser Independiente Dominante Mínimo. Es decir, se itera la lista de vecinos hasta encontrar un nodo que al insertarlo cumpla ser un conjunto solución válido o hasta que termine sin haber podido formar un CIDM.

Lo que tendrá un costo de $O(n^2)$.

Primero se fija que esta combinación de quitar los nodos (i,j) y agregar el nodo actual no se haya probado todavía, si es así se agrega el nodo actual de la lista al conjunto solución de manera ordenada. Se itera el vector hasta la posición donde se debe insertar lo que tiene costo de $O(n)$ ya que en peor caso en la solución original había $n - 1$, se sacaron i y j por lo que el vector quedó de un tamaño $n - 2$. Si el nodo a insertar debe hacerse al final, recorre todos.

`iterator::insert` tiene complejidad $O(n)$

Ahora con el conjunto formado se invoca a la función lisAdy::esIndependienteMaximal que tiene una complejidad de $O(n^2)$

Si el conjunto armado hasta el momento no es Independiente Maximal, vamos a querer borrar el nodo añadido último por lo que se itera de nuevo el vector hasta encontrarlo y borrarlo con `iterator::erase` con un costo total de $O(n)$.

Este código si al momento de quitar dos nodos (i,j) puede añadir diversos nodos y con cualquiera resulta un conjunto Independiente Maximal, sólo considera al primero que logre cumplir ser un conjunto Independiente Maximal.

Si al quitar dos nodos y agregar uno se logró un conjunto solución válido se actualiza el vecotr solución optimo.

El ciclo mayor iterará hasta que no se hagan más cambios. Esto va a pasar cuando haya probado todos los pares de nodos posibles y por cada uno probado añadir otro.

Es decir, en el peor caso se comenzó con una solución inicial de $n-1$ nodos y, seizaron dos y añadió uno hasta que la solución final quede de tamaño 1. Es decir, el while más grande iterará en peor caso n veces.

Por cada par de nodos iterará n^2 veces y la cantidad de pares posibles en peor caso será de $O(n^2)$.

Por lo tanto el costo de cada iteración del ciclo mayor será de $O(n^4)$, iterándola n veces dará un costo total del algoritmo de $O(n^5)$.

En los casos de **ejecución 4 y 5**, el procedimiento será similar.

Primero buscará ternas con vecinos en común (dameTernasVecinasComun) con un costo de $O(n^4)$.

Pero el costo de cada iteración del ciclo mayor será de $O(n^4)$, por lo explicado anteriormente, con lo cual el costo total del algortimo es $O(n^5)$ también para esta vecindad.

1.3. Experimentación

El objetivo de esta sección es comparar los cuatro métodos implementados bajo el paradigma de *Búsqueda Local* entre sí, respecto de su tiempo de ejecución y del conjunto solución resultante.

Además, las soluciones obtenidas por los cuatro métodos se contrastan con la solución óptima resultante de ejecutar el algoritmo exacto para las mismas instancias.

Todas las instancias utilizadas fueron generadas con el generador de instancias detallado en la Sección 2.2.

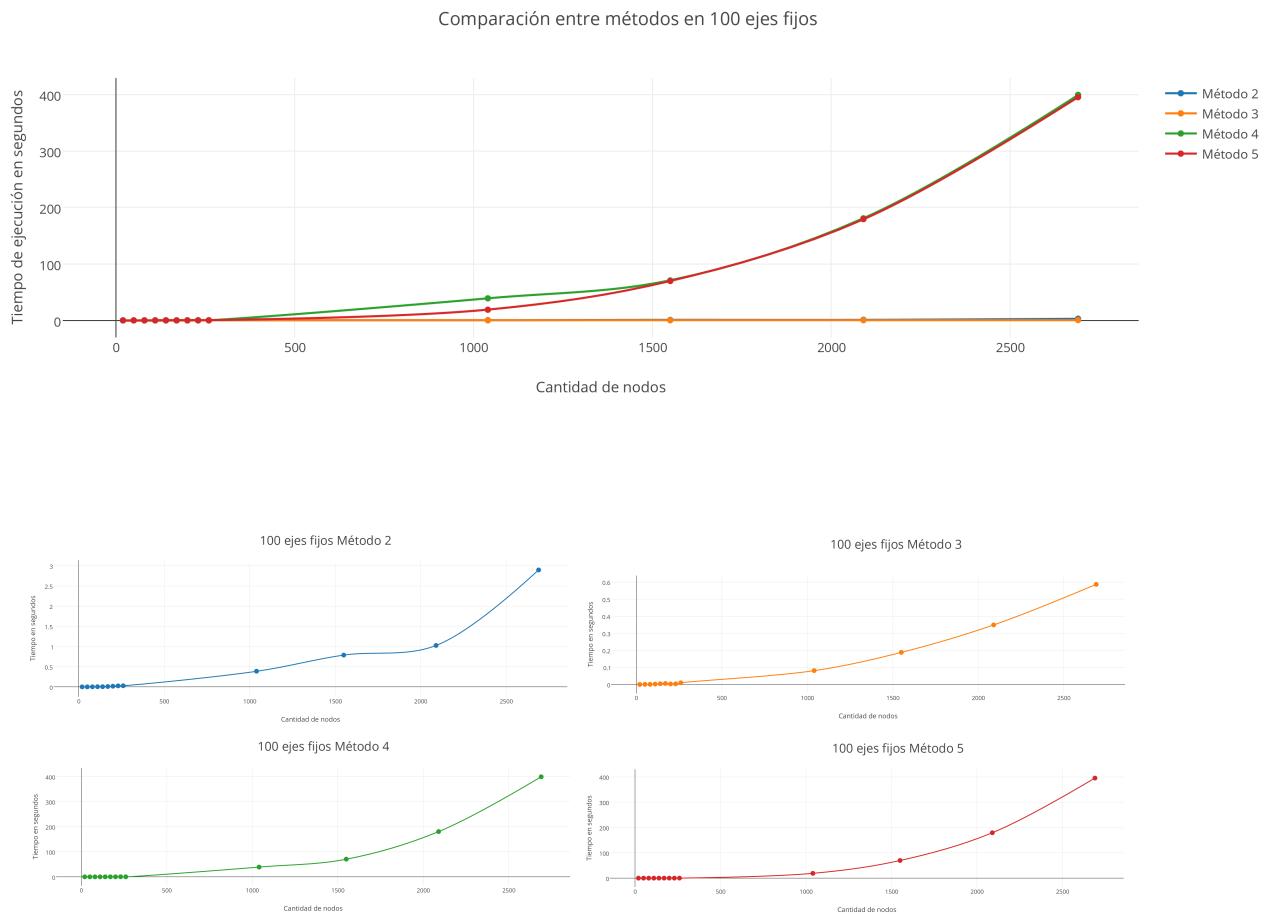
1.3.1. Análisis de tiempos de ejecución

En primera instancia, contrastamos tiempos de ejecución entre los cuatro métodos. Como así también el crecimiento de la curva de tiempos de ejecución acorde varían los ejes y los nodos dejando fijo nodos y ejes respectivamente.

Ejes Fijos

Con el generador de instancias se armaron grupos de grafos con 100, 500, 2000 y 4000 ejes variando la cantidad de nodos. Se ejecutaron los cuatro métodos para todos los casos generados.

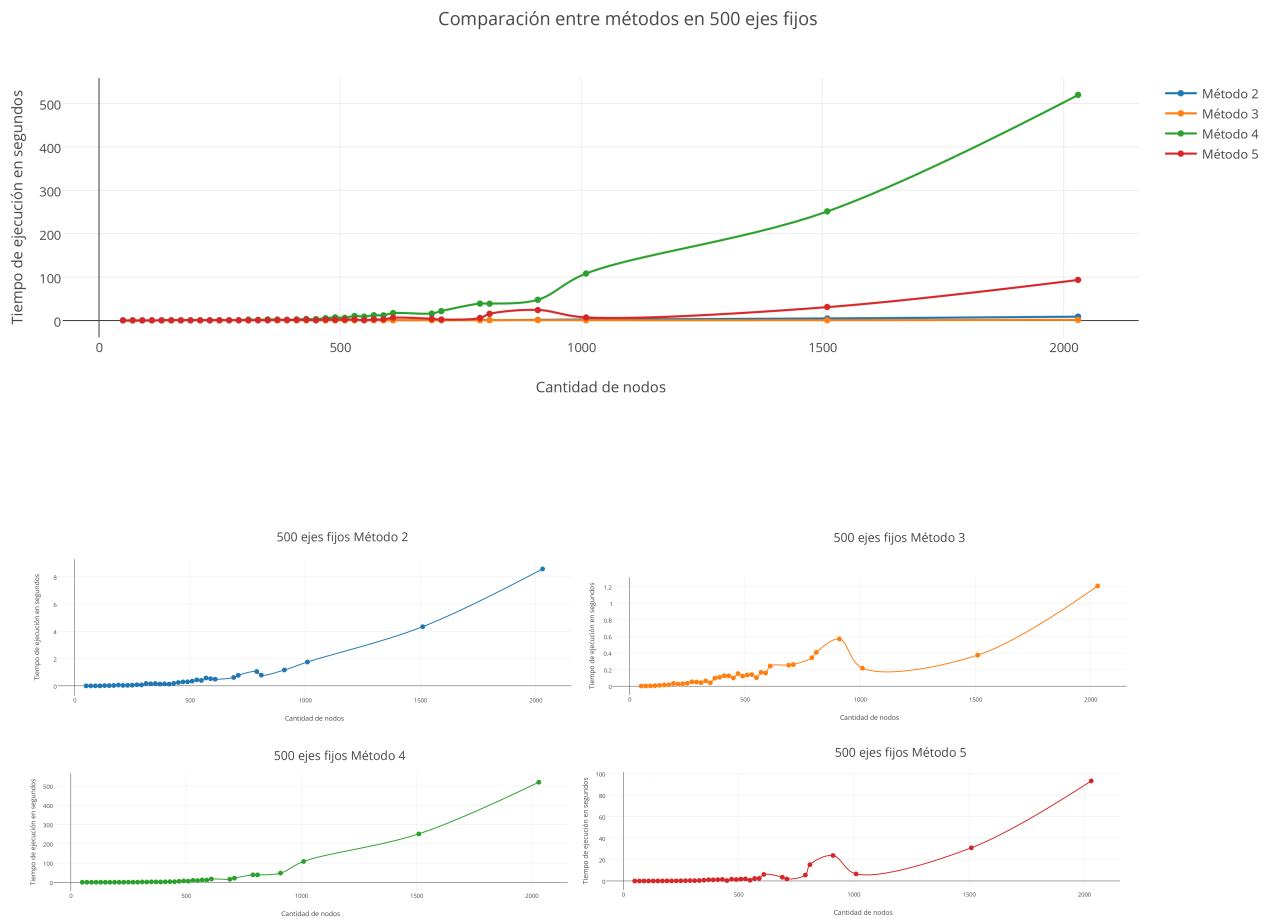
Se midieron los tiempos de ejecución y se exponen a continuación:



En el primer gráfico se puede apreciar que cuando la cantidad de nodos comienza a ser mayor, los métodos con la segunda vecindad (4 y 5) presentan una curva que posee un crecimiento mayor que las otras dos. Este comportamiento se condice con la complejidad teórica calculada ya que los métodos 3 y 4 tienen una complejidad de $O(n^6)$ (?) mientras que los primeros dos tienen $O(n^5)$ (?).

En el segundo gráfico lo que se quiso hacer es mostrar con detalle cada método ya que en el primero, al tener cantidades de tiempo variadas, no se puede apreciar con detalle el comportamiento de todas las curvas por sí solas.

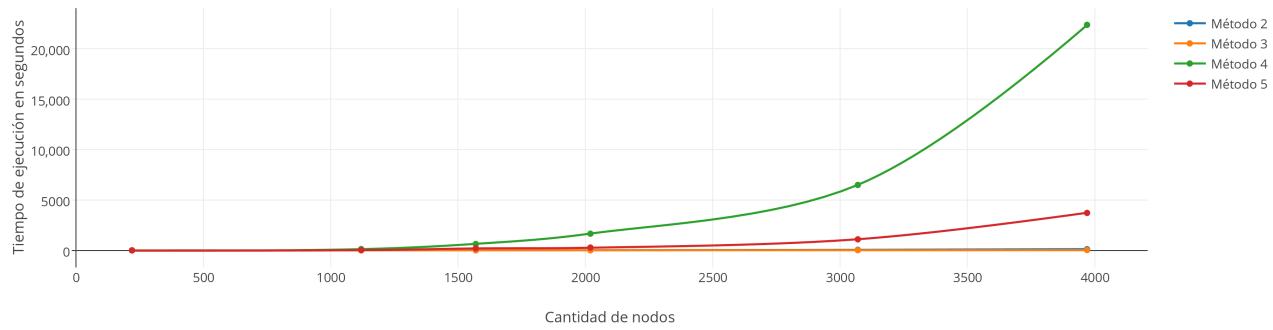
De este modo, se puede apreciar que los cuatro métodos tienen un comportamiento que asemeja a una función polinomial (**? es así?? nunca me acuerdo eso!**) aunque a simple vista no se podría acertar que sean las mismas funciones nombradas como complejidad.



Al realizar el mismo procedimiento pero ahora con 500 ejes fijos lo que se puede apreciar es que, si bien los métodos 3 y 4 se mantienen en las mediciones de tiempos mayores, la curva del **Método 4** se aleja notablemente de los demás.

En las ampliaciones de los métodos, desde una perspectiva amplia, se puede notar que poseen un comportamiento que asimila **polinomial** al igual que en el caso anterior. Se podría decir que la causa de que las gráficas no sean estrictamente creciente se debe a la complejidad $O(n^5)$ y $O(n^6)$.

Comparación entre métodos en 2000 ejes fijos



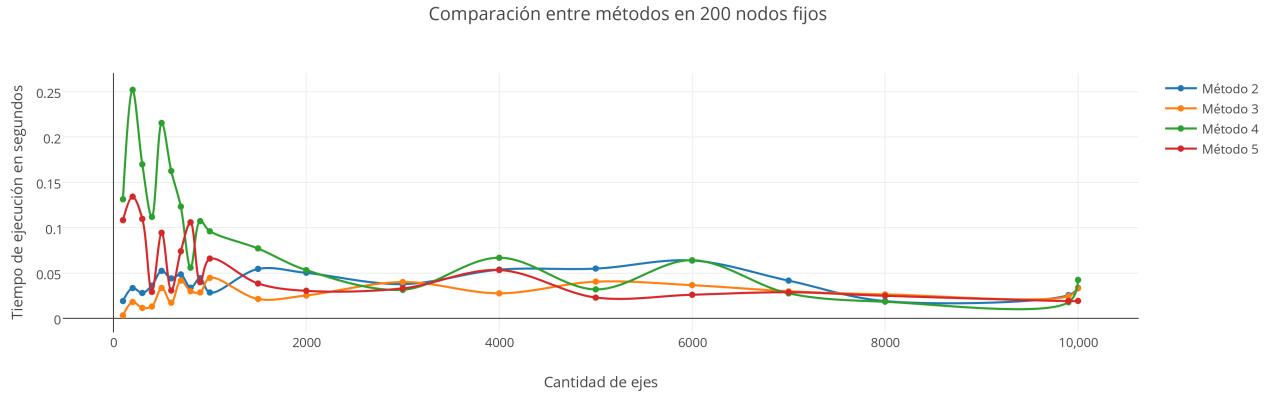
Comparación entre métodos en 4000 ejes fijos



Al plantear la misma experimentación con cantidad de ejes mayor, se puede apreciar un comportamiento análogo donde el **método 4** posee un tiempo de ejecución notablemente mayor al resto y el **método 3** permanece siendo el de menor tiempo de ejecución.

Nodos Fijos

En esta instancia, se contrastarán grupos de grafos que posean cantidad de nodos fijos: 200, 300, 500, 600 y 700 variando en cada caso la cantidad de ejes. Se comparan los tiempos de ejecución entre los cuatro métodos planteados en los casos generados.

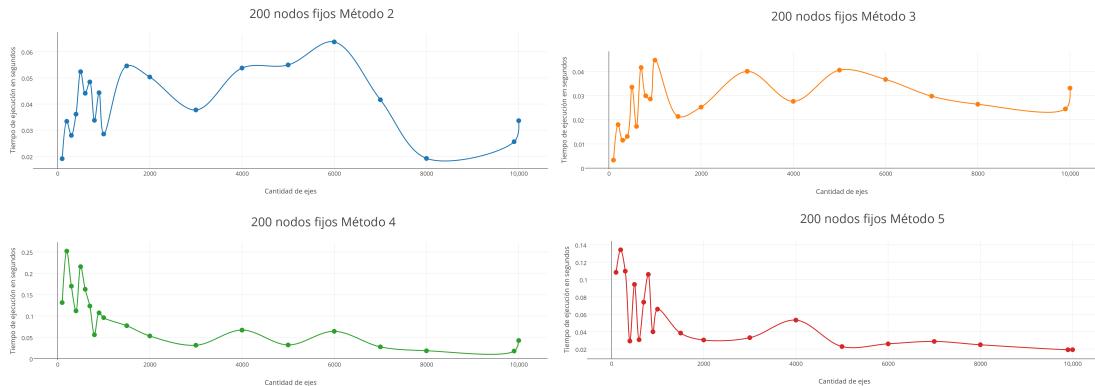


Dando una observación general sobre este gráfico, se puede apreciar que para todos los métodos el tiempo de ejecución disminuye al aumentar la cantidad de ejes.

A simple vista podría sonar un poco absurdo, sin embargo la causa de este comportamiento está ligada a que al existir mayor cantidad de ejes (manteniendo la cantidad de nodos) aumenta el grado de los nodos. Por lo tanto, [al igual que el algoritmo Goloso\(?\)](#), cuando se arma una solución inicial se inserta el nodo de mayor grado. Luego, se descartan todos los nodos vecinos ya que no serán candidatos al conjunto solución.

Si bien, para descartar nodos vecinos, se debe recorrer la lista de adyacencia completa; ese tiempo es compensado al tener menos nodos en la siguiente iteración para recorrer.

En cuanto a la parte del algoritmo que consiste en la búsqueda local, [Completar algo aca...](#)



Haciendo hincapie en cada método por separado, se observa un comportamiento distingible.

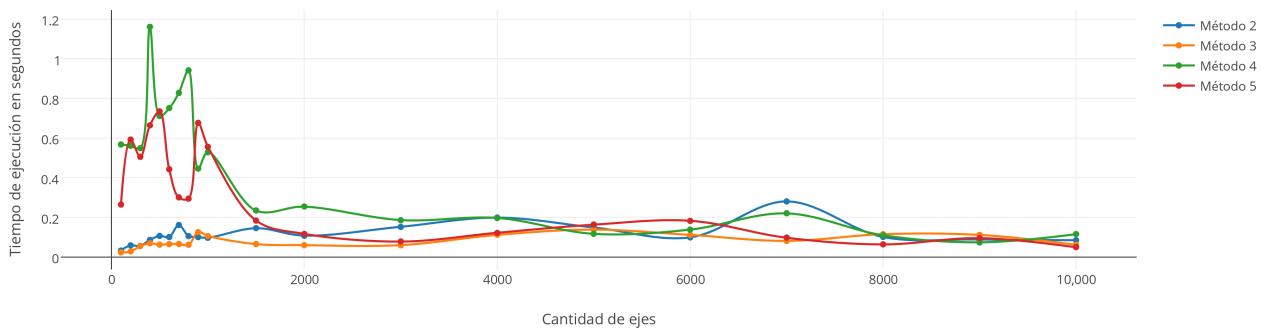
Si bien las curvas de los cuatro métodos oscilan notablemente, si uno observa bajo un punto de vista general se puede apreciar que los métodos 4 y 5 tienen un comportamiento notablemente decreciente similar al de una curva [polinomial \(?\)](#). Ambos métodos utilizan la segunda vecindad, lo que indicaría que el comportamiento es causado por [completar!!!](#).

Sección 1.3 Experimentación

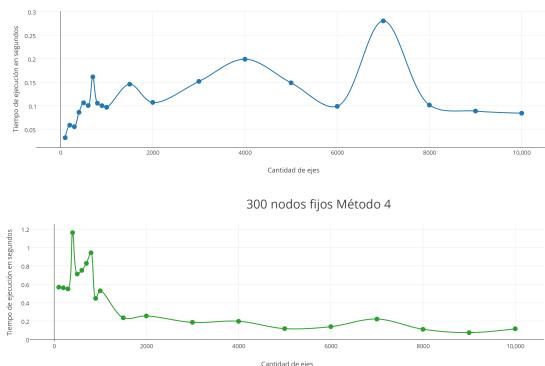
Por otro lado, los métodos 3 y 4 poseen un crecimiento abrupto con una cantidad de ejes menor y después sus curvas oscilan de manera constante. Como ambas manejan la misma vecindad, un potencial causante de dicho comportamiento corresponde a ... **COMPLETAR**

Ejecutamos la misma experimentación, pero en esta ocasión fijando la cantidad de nodos en 300, 500 y 600.

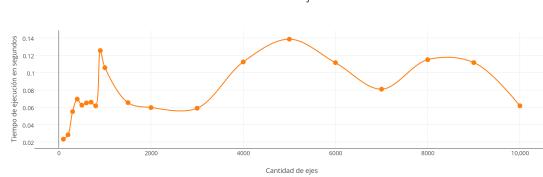
Comparación entre métodos en 300 nodos fijos



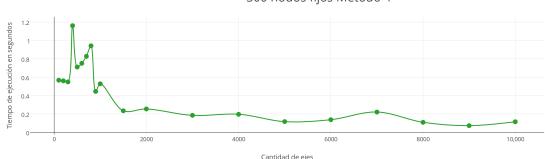
300 nodos fijos Método 2



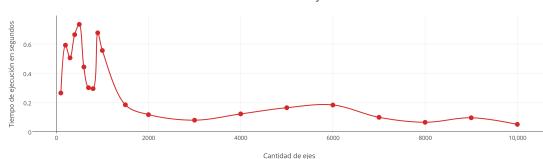
300 nodos fijos Método 3



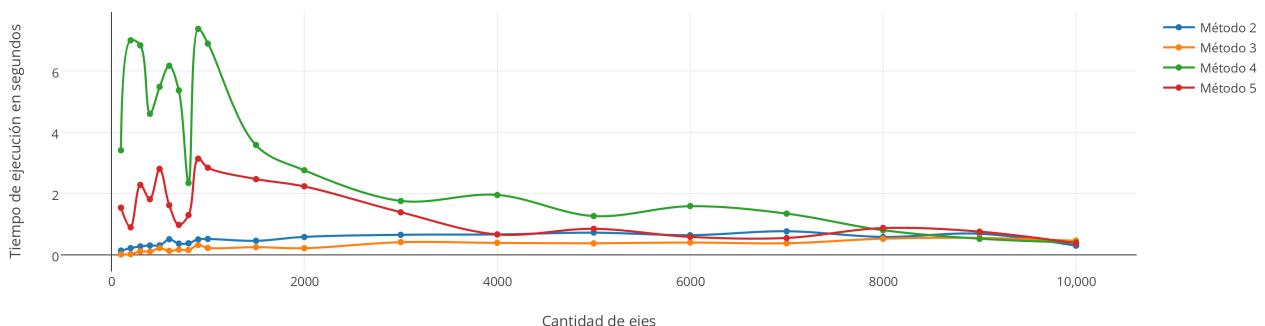
300 nodos fijos Método 4



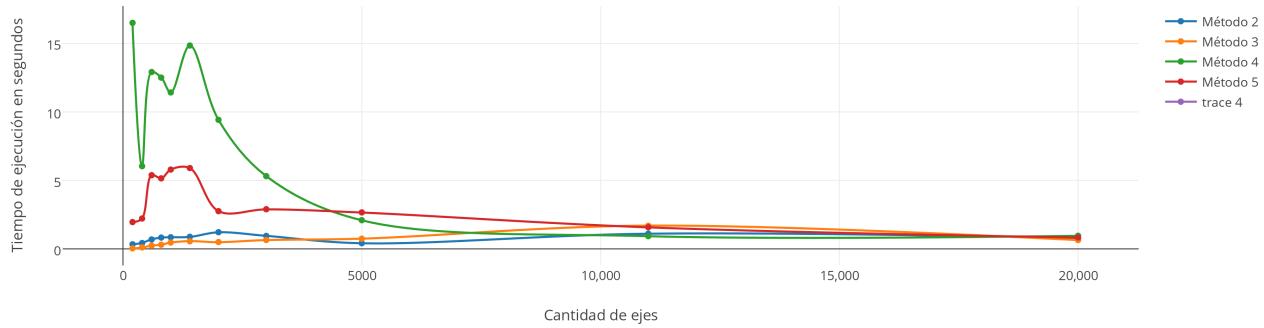
300 nodos fijos Método 5



Comparación entre métodos en 500 nodos fijos



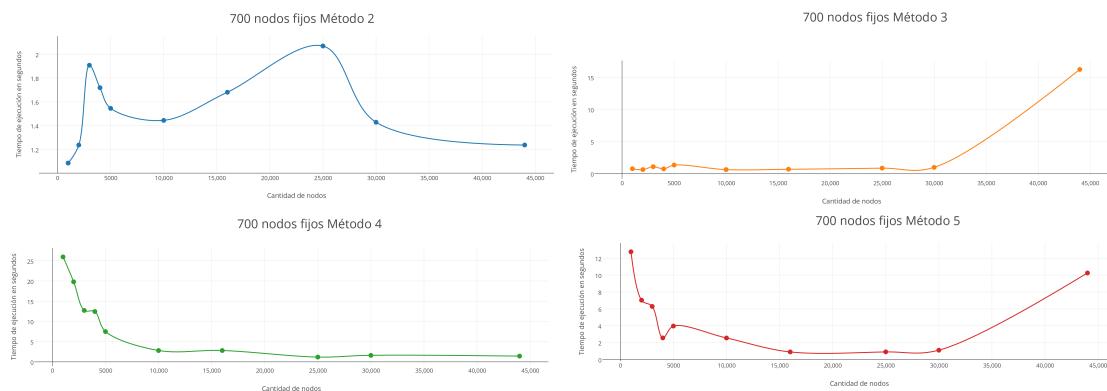
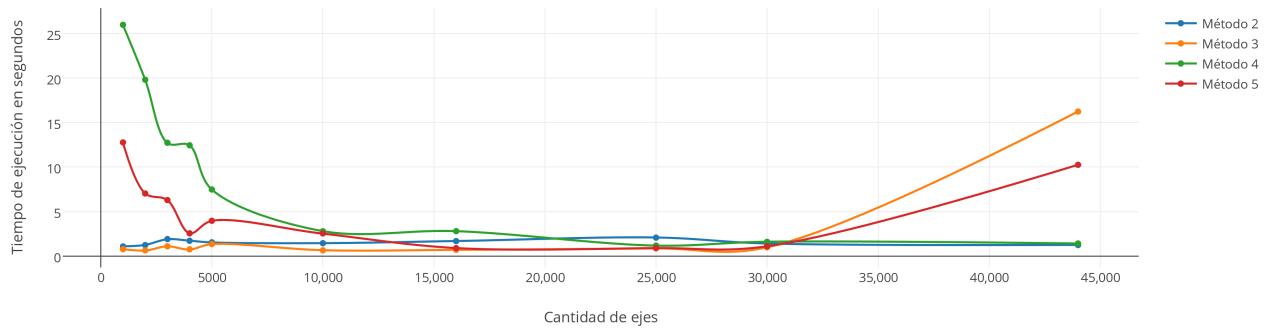
Comparación entre métodos en 600 nodos fijos



La conclusión que se puede sacar de este último set de gráficos es que, si se fija la cantidad de nodos, no importa en qué valor, los tiempos de ejecución de los métodos poseen un comportamiento similar al explicado en el inciso anterior.

Donde el **Método 4** permanece siendo quien tiene mayores tiempos de ejecución sin importar la cantidad de nodos y ejes. Así mismo, el **Método 3** se mantiene con los tiempos de ejecución menores.

Comparación entre métodos en 700 nodos fijos



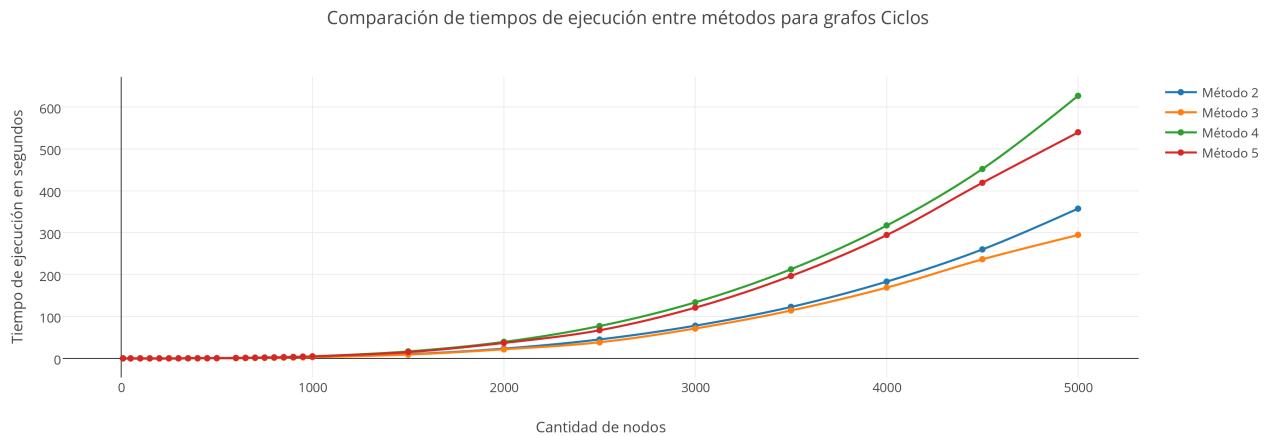
Nos pareció un caso notable de distinción el de 700 nodos fijos, cuando se analizan los métodos por separado.

Si bien a ciencia cierta no se puede asignar a qué función pertenece cada curva, pero se pueden notar las **raíces** (?) que poseen las curvas. De modo que se aprecia un comportamiento no estrictamente creciente.

1.3.2. Contrastación empírica de la complejidad

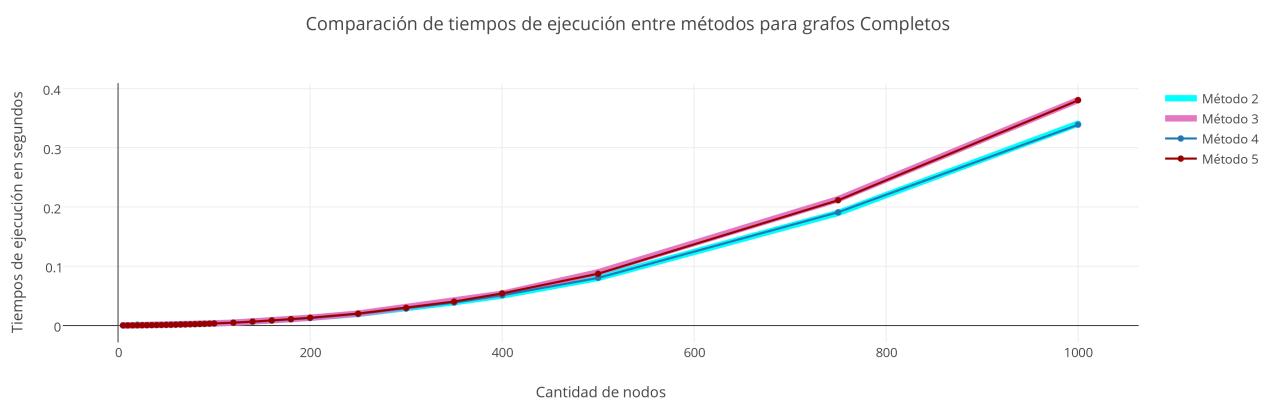
Se quiso linealizar los tiempos de ejecución con el fin de poder contrastar de manera óptima las complejidades empíricas con las teóricas. Sin embargo, esto no fue posible debido a que los tiempos de ejecución son muy pequeños y al dividirlos por la cantidad de nodos los resultados obtenidos no generan gráficos de real interés.

A continuación se exponen los tiempos de ejecución para los distintos métodos, considerando grafos Ciclo y grafos Completos:



Los tiempos de ejecución para grafos Ciclos se condicen con los gráficos de la sección ???. Esto significa que los **métodos 4 y 5** poseen mayor tiempo de ejecución y finalmente, el **método 2** preserva el menor.

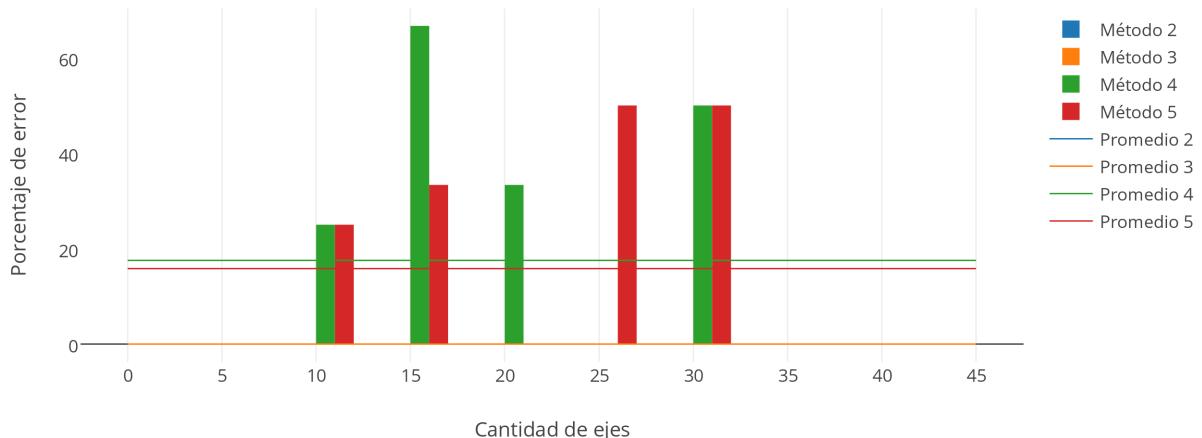
Todas las curvas presentan un comportamiento que se asemeja a un valor cuadrático o cúbico, ya que si bien la complejidad teórica es de $O(n^5)$ (?) al tener sólo dos vecinos cada nodo las listas de adyacencia se recorren en $O(2) \subseteq O(1)$.



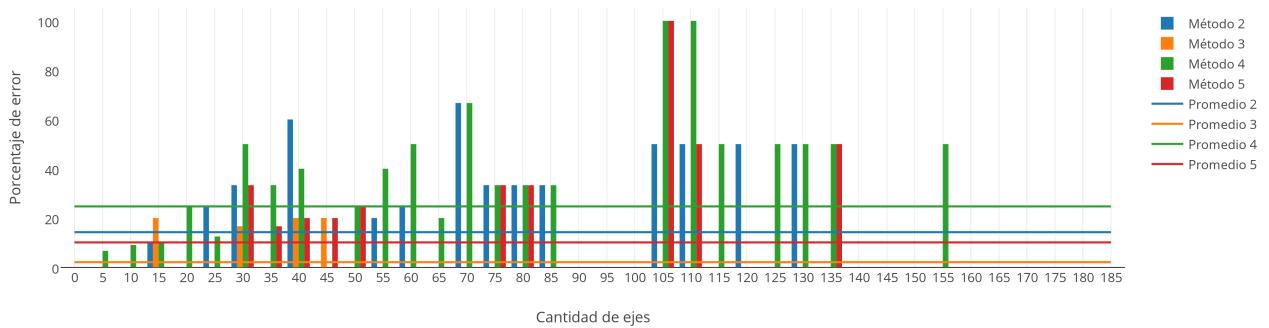
Los grafos completos son...

1.3.3. Comparación soluciones Local vs Exacto

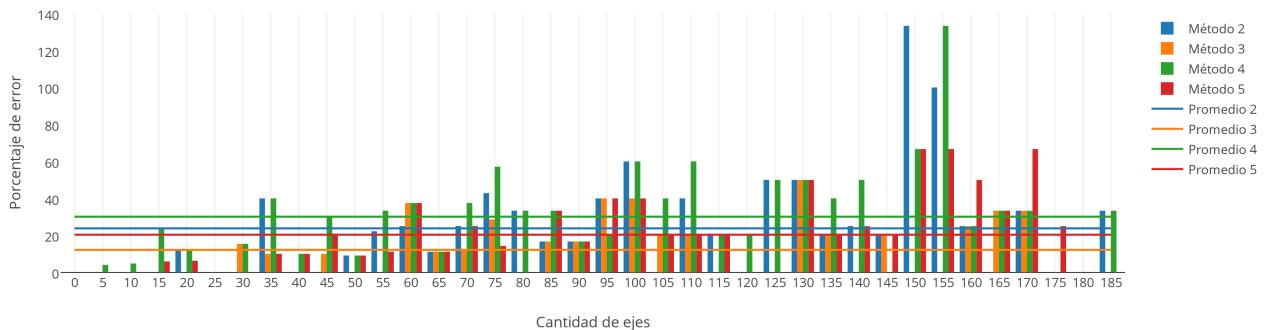
Comparación entre Local y Exacto con 10 nodos fijos



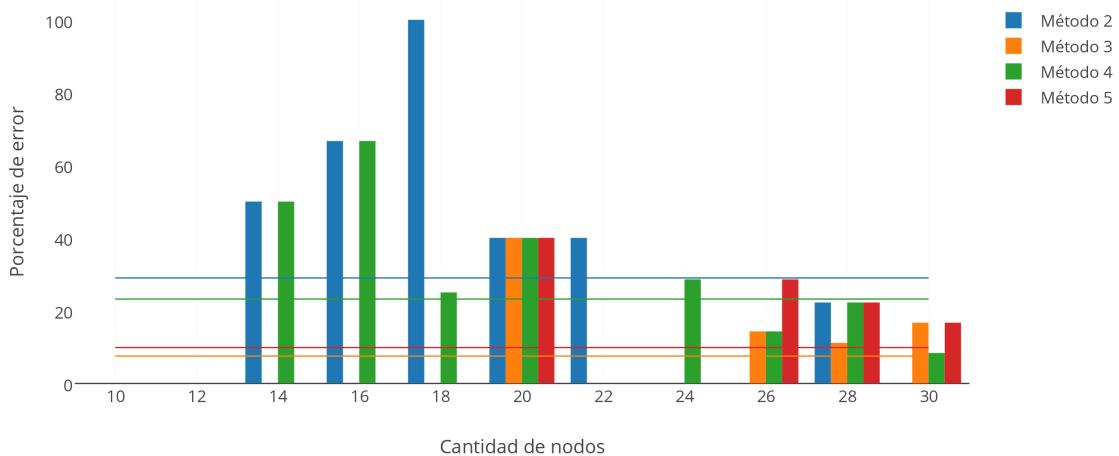
Comparación entre Local y Exacto con 20 nodos fijos



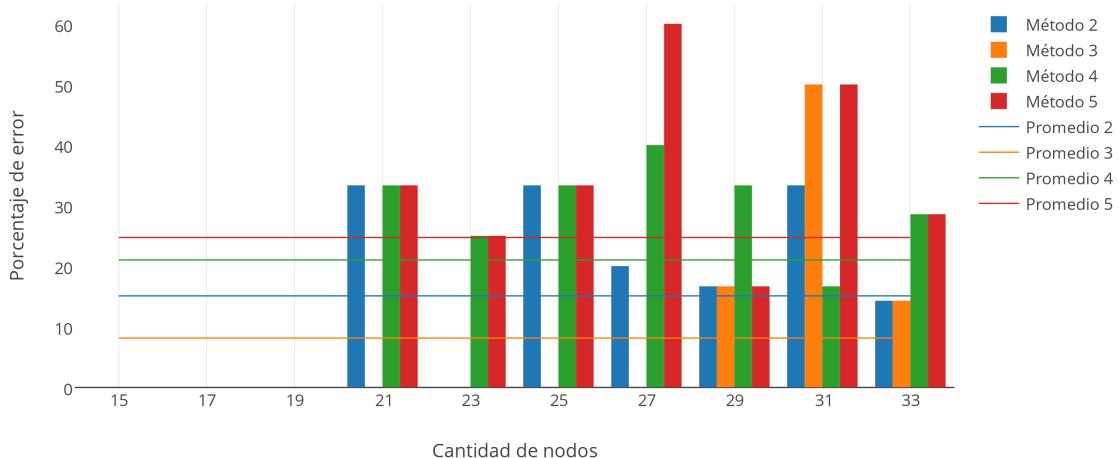
Comparación entre Local y Exacto con 30 nodos fijos

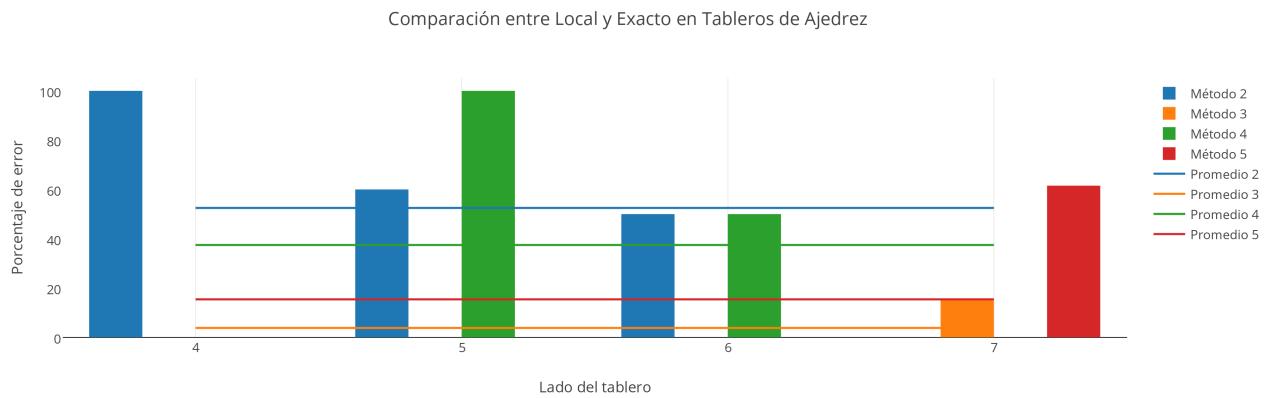


Comparación entre Local y Exacto con 45 ejes fijos



Comparación entre Local y Exacto con 90 ejes fijos

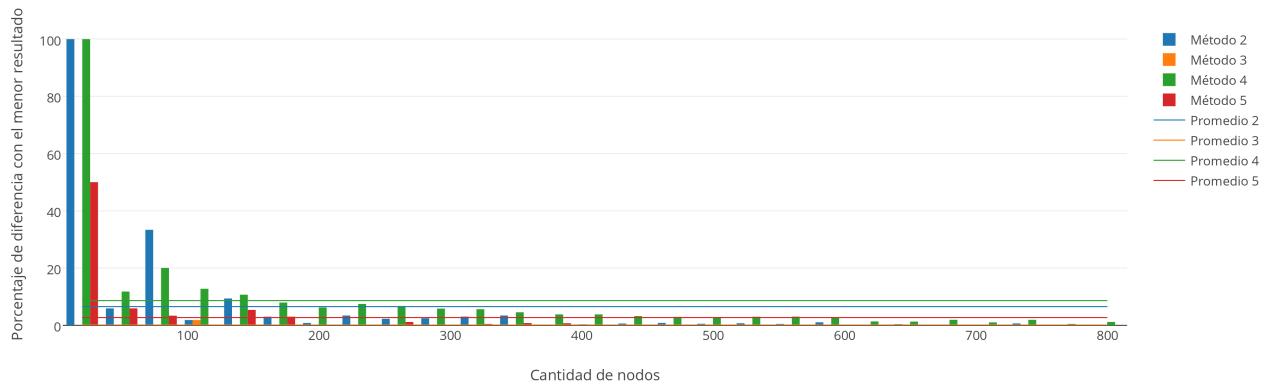




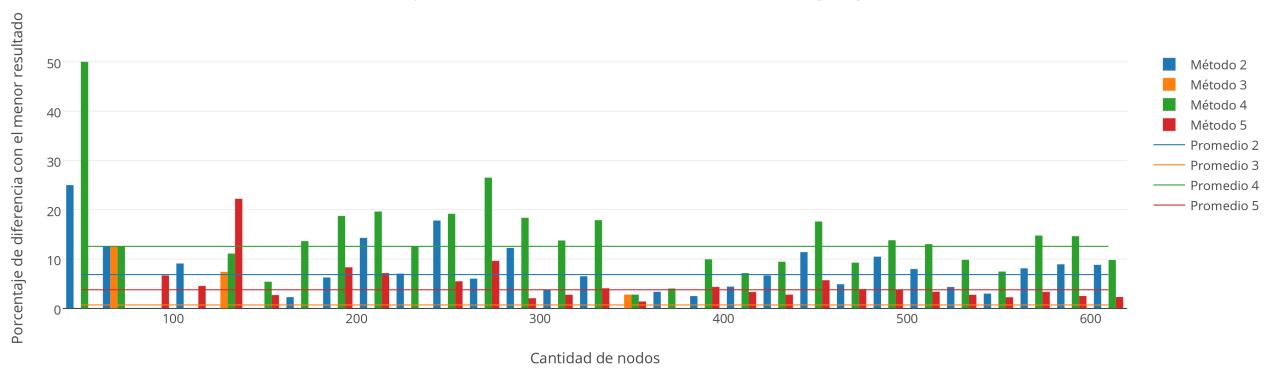
1.3.4. Elección de versión óptima

Ejes Fijos

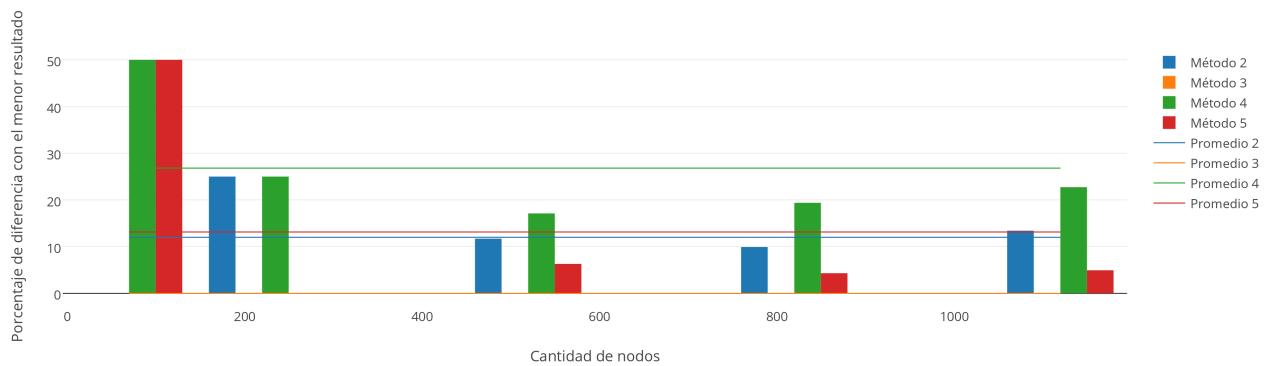
Comparación de resultados entre métodos con 100 ejes fijos



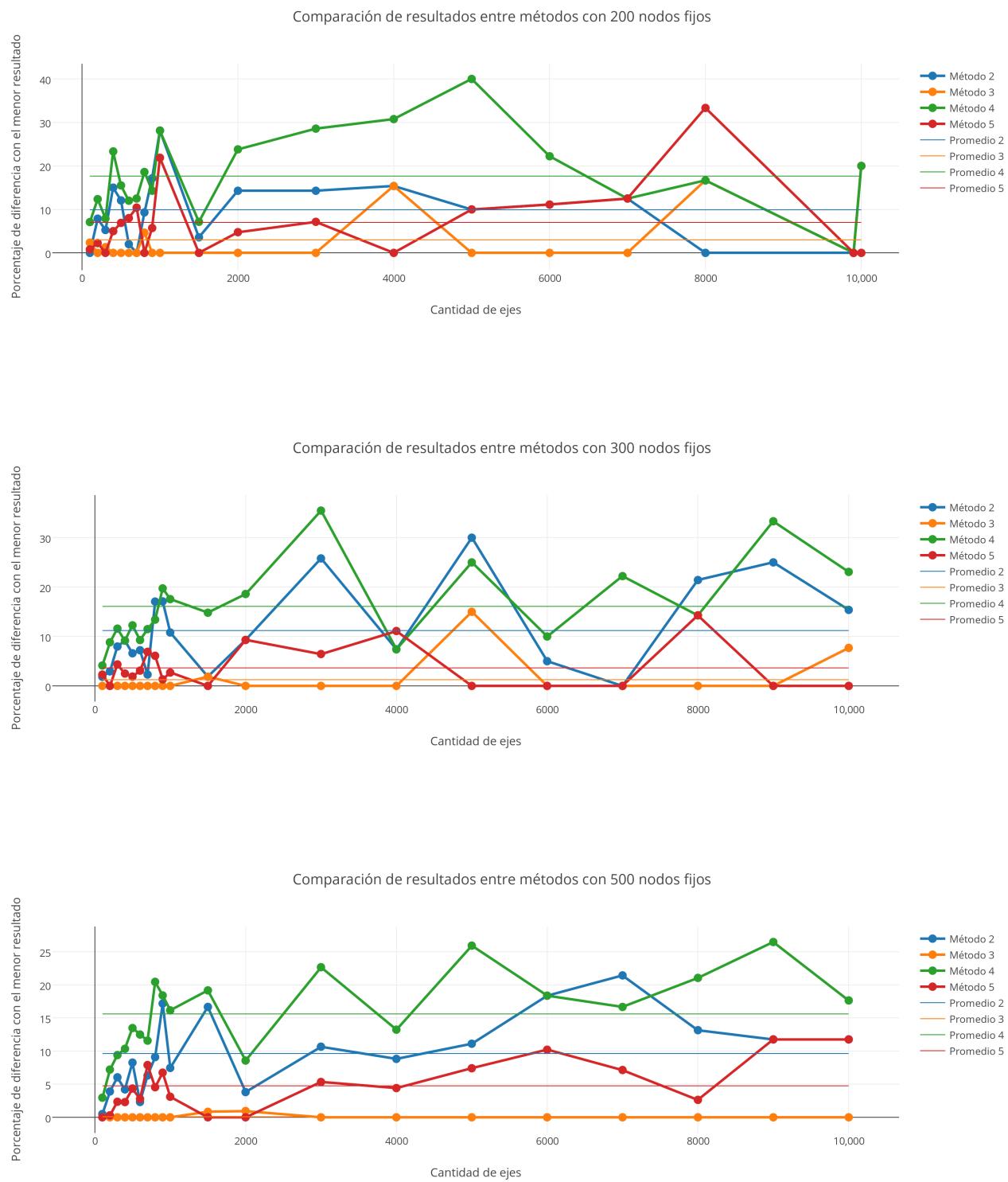
Comparación de resultados entre métodos con 500 ejes fijos

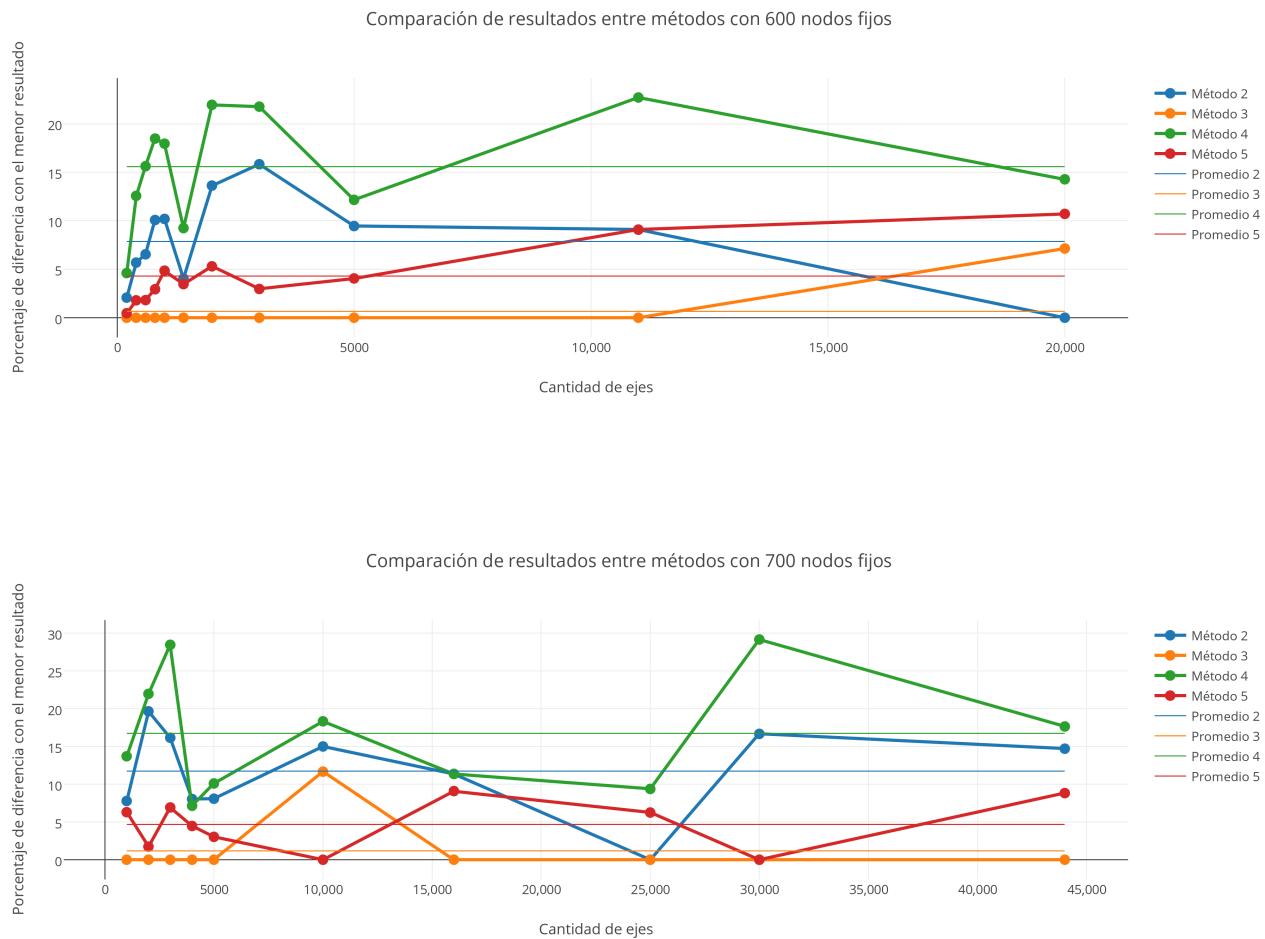


Comparación de resultados entre métodos con 2000 ejes fijos



Nodos Fijos





2. Anexo

2.1. Ejecución de los métodos

Al momento de ejecutar el `main` se le deben pasar los siguientes parámetros acorde a lo deseado:

- **0:** *Algortimo Exacto*;
- **1:** *Heurística Greedy* (con parámetros `alpha = 0`, `conAlpha = true`);
- **2:** *Heurística Búsqueda local* (solución inicial por orden de nomenclatura (`greedy = false`), vecindad `2x1` (`vecindad = true`));
- **3:** *Heurística Búsqueda local* (solución inicial `greedy` (`greedy = true`), vecindad `2x1` (`vecindad = true`), `alpha = 0`);
- **4:** *Heurística Búsqueda local* (solución inicial por orden de nomenclatura (`greedy = false`), vecindad `3x1` (`vecindad = false`));
- **5:** *Heurística Búsqueda local* (solución inicial `greedy` (`greedy = true`), vecindad `3x1` (`vecindad = false`), `alpha = 0`);
- **6:** *Heurística GRASP* (solución inicial por porcentaje de mejores (`conAlpha = true`), vecindad `2x1` (`vecindad = true`), `alpha = input`);
- **7:** *Heurística GRASP* (solución inicial por porcentaje de mejores (`conAlpha = true`), vecindad `3x1` (`vecindad = false`), `alpha = input`);
- **8:** *Heurística GRASP* (solución inicial por cantidad de mejores (`conAlpha = false`), vecindad `2x1` (`vecindad = true`), `alpha = input`);
- **9:** *Heurística GRASP* (solución inicial por cantidad de mejores (`conAlpha = false`), vecindad `3x1` (`vecindad = false`), `alpha = input`);
- **i:** *Imprime* la lista de adyacencia del grafo pasado por `input`;
- **q:** *Finaliza* la ejecución;

2.2. Generación de casos de test

Aca hablar un poco de como generamos los grafos para analizar y bla, no hace falta poner codigo ni nada... pero por lo menos contar que hace el generador de instancias. Asi en cada sección de experimentación se cita esta sección