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 codígo. Búsqueda Local utiliza Greedy para generar instancias iniciales, GRASP aprovecha a la búsqueda local y una adaptación del greedy. Ver sección 7.

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. Introducción al problema

1.1. Conjunto Independiente Dominante Mínimo (CIDM)

Sea $G = (V, E)$ un grafo simple. Un conjunto $D \subseteq V$ es un *conjunto dominante* de G si todo vértice de G está en D o bien tiene al menos un vecino que está en D.

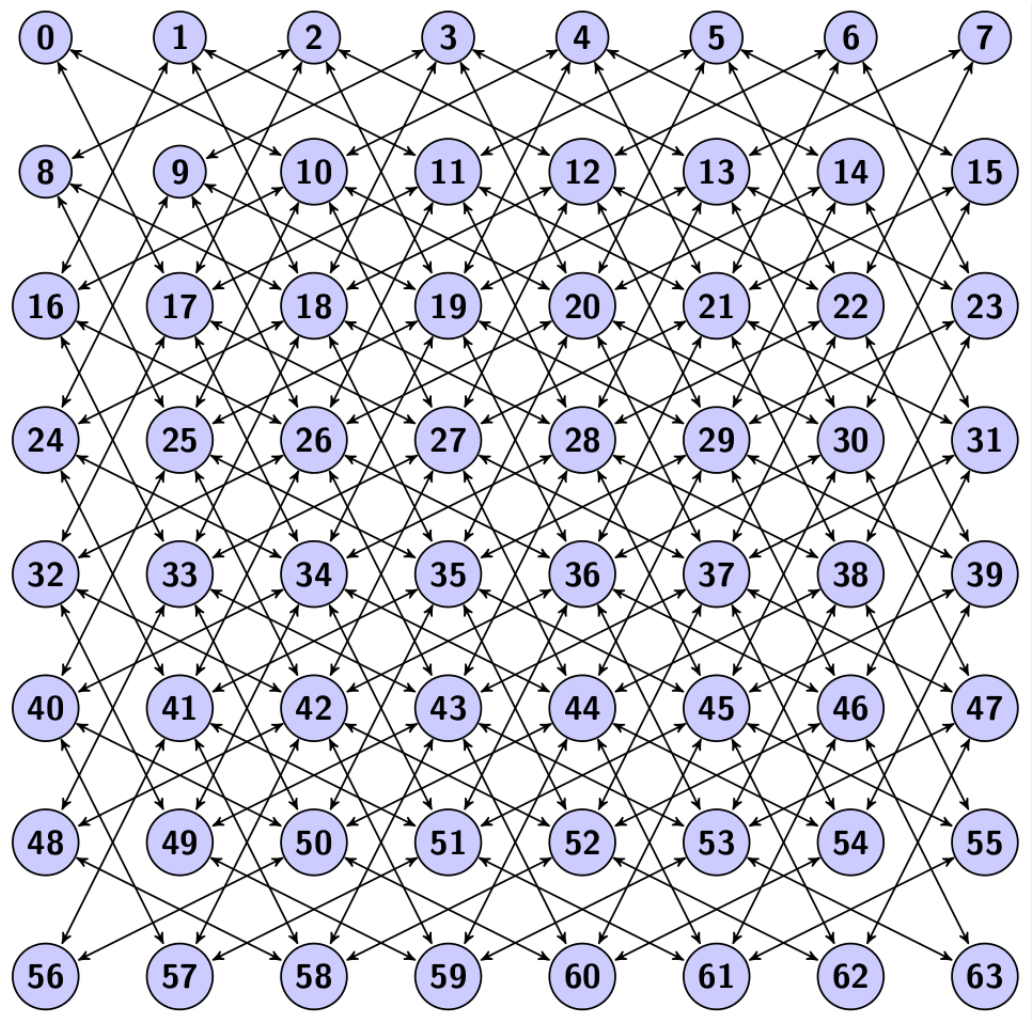
Por otro lado, un conjunto $I \subseteq V$ es un *conjunto independiente* de G si no existe ningún eje de E entre dos vértices de I.

Definimos entonces un *conjunto independiente dominante* de G como un conjunto independiente que a su vez es un conjunto dominante del grafo G.

El problema de Conjunto Independiente Dominante Mínimo (CIDM) consiste en hallar un conjunto independiente dominante de G con mínima cardinalidad.

1.2. Paralelismo con "El señor de los caballos"

El problema "*El señor de los caballos*" es similar al problema de encontrar un *Conjunto Independiente Dominante Mínimo* considerando solamente una familia específica de grafos. Esta es la familia de grafos donde cada nodo modela un casillero de un tablero de ajedrez y sólo existe un eje entre dos nodos v_1 y v_2 si el movimiento desde v_1 a v_2 o *viceversa* es un movimiento permitido para una pieza de caballo.



La relación con nuestro problema es la siguiente:

- "El señor de los caballos" busca un conjunto dominante, dado que intenta ocupar el tablero, y para ello requiere que o bien cada casillero tenga un caballo, o bien que cada casillero sea amenazado por un caballo.
- "El señor de los caballos" busca un conjunto mínimo, es decir, que utilice la menor cantidad de caballos posibles.
- "El señor de los caballos" **NO** busca un conjunto independiente, dado que si fuese necesario, un caballo puede ubicarse en un casillero que estuviese siendo amenazado por otro.

1.3. Todo conjunto independiente maximal es un conjunto dominante

Para asegurarnos de que un conjunto es independiente y dominante al mismo tiempo, vamos a demostrar que vale la siguiente propiedad:

Sea $G = (V, E)$ un grafo simple, un *conjunto independiente* de $I \subseteq V$ se dice *maximal* si no existe otro conjunto independiente $J \subseteq V$ tal que $I \subset J$, es decir que I está incluido estrictamente en J.

Todo conjunto independiente maximal es un *conjunto dominante*.

Demostración

Sean $G = (V, E)$ grafo simple, $I \subseteq V$ *conjunto independiente maximal*.

Quiero ver que I es un *conjunto dominante*:
Lo que es equivalente a probar que $(\forall\ nodo\ v \in V)\ ((v \in I) \vee (\exists\ nodo\ w \in adyacentes(v), w \in I))$

Supongo por el absurdo que: $(\exists\ nodo\ v \in V)\ tq\ ((v \notin I) \wedge (\forall\ nodo\ w \in adyacentes(v), w \notin I))$

Considero a $adyacentes(I)$ como el conjunto que se obtiene de concatenar todos los vecinos de cada elemento de I. Por lo tanto, es equivalente decir $(\forall\ nodo\ w\ \in adyacentes(v), w\ \notin\ I)$ y decir $(v \notin adyacentes(I))$.

$\Rightarrow v \notin I \wedge v \notin adyacentes(I)$
$\Rightarrow \exists$ conjunto independiente J: $J \subseteq V$ tq $J = I + \{v\}$
J es independiente porque I lo era y al agregarle el nodo v se mantiene esta propiedad ya que v no pertenecía a I y además no estaba conectado a ningún nodo del conjunto I.
$\Rightarrow \exists J$ conjunto independiente tq $I \subset J$. **Absurdo!**(I era un conjunto Independiente Maximal)

El absurdo provino de suponer que I era un conjunto independiente maximal, pero no dominante. Por lo tanto, I debe ser un conjunto dominante.

1.4. Situaciones de la vida real

Situaciones de la vida real que puedan modelarse utilizando CIDM:

- **Ubicación de estudiantes al momento de rendir un examen:** Encontrar la mejor manera de ubicar a todos los estudiantes en el aula, tal que ninguno esté suficientemente cerca de otro como para copiarse, pero tal que entre la mayor cantidad de estudiantes posibles. Esta situación es lo mismo que modelar el aula como un grafo donde cada asiento es un nodo y cada asiento es adyacente a los asientos que estan a sus costados (en todas las direcciones); luego buscar el *CIDM* de dicho grafo.
- **Ubicación de servicios en ciudades:** Para minimizar costos, es probable que si se quiere situar centros de servicios (cualesquiera sean estos: hospitales, estaciones de servicio, distribuidoras, etc), se los situe de manera tal que tengan amplia cobertura, pero sin situar demasiados centros, es decir, situando la mínima cantidad. Tampoco se querría que un centro cubra la misma zona que otro centro. Si se modela a la ciudad, tomando cada zona (arbitraria, como barrios, o manzanas, o conjunto de manzana) como un nodo, en los que cada nodo es adyacente al nodo que representa la zona vecina, entonces el problema de situar estos centros minimizando costos, es igual a encontrar un *CIDM* en el grafo mencionado.

2. Algoritmo Exacto

De acuerdo a lo ya explicado en el inciso 1.2, podemos establecer una analogía con este problema y "El señor de los caballos". Por lo tanto, la metodología empleada para la implementación del algoritmo exacto también fue la de *Backtracking*.

De este modo, nos vemos obligados a recorrer inteligentemente todos los conjuntos dentro del conjunto de partes del total de nodos V. Mediante el backtracking podemos realizar podas y estrategias para saltear algunas ramas de decisión donde se predice que no se va a poder encontrar la solución óptima allí.

2.1. Explicación y mejoras

Nuestro algoritmo recorre ordenadamente el conjunto de partes de V y por cada uno de ellos verifica que cumpla la función `esIndependienteMaximal()`. La misma devuelve 0 si es falso, la cantidad de nodos en el conjunto en caso contrario.

Es decir, se itera sobre los nodos y, considerando el nodo actual como presente o como ausente, termina encontrando el mínimo conjunto independiente maximal.

En una variable se acumula la solución óptima hasta el momento, la cual se actualiza cuando se encuentra un nuevo conjunto independiente maximal que tiene un cardinal menor al óptimo actual. En ella va a quedar la solución buscada luego de correr el algoritmo.

La poda que implementamos consistió en verificar que una futura solución posible no cuente con una cantidad igual o superior de nodos que la solución óptima obtenida hasta el momento. Esto quiere decir que si la óptima actual consiste de k nodos y nos encontramos ante la pregunta de agregar un nuevo nodo a una posible solución de $k - 1$ nodos, ésta será a lo sumo tan buena como la que ya teníamos, por lo que no nos resulta útil contemplarla. De esta forma, se descartan rápidamente todas las soluciones peores o iguales que la actual.

Otra poda posible consistía en separar cada una de las componentes conexas del grafo y, para cada una de ellas, buscar el mínimo conjunto independiente maximal. Luego, uniendo todos estos resultados, se obtiene el conjunto deseado del grafo.

Una estrategia podría ser verificar si agregar el nodo actual me va a resultar útil. Es decir, por ejemplo, si estoy agregando un vecino de un nodo que ya se encuentra en el conjunto, este no será útil ya que la solución no sería maximal. De esta forma, podríamos evitar revisar aquellos conjuntos.

Estas dos optimizaciones no se implementaron porque requerían un manejo distinto de estructuras y nos pareció que hubieran empeorado los tiempos de ejecución.
está bien así? o es super choto?. no sabría que más poner sino :/

```
unsigned int calcularCIDM(Matriz& adyacencia, unsigned int i, vector<unsigned int>& conjNodos,
vector<unsigned int>& optimo){

if (conjNodos.esIndependienteMaximal()) then
    optimo ⟵ conjNodos;
    return optimo.size();
end
if (i.estaEnRango()) then
    if (conjNodos es más grande que el optimo actual) then
        return 0;
    end
    siNoAgrego ⟵ calcularCIDM(adyacencia, i+1, conjNodos, optimo);
    agrego el nodo i a conjNodos;
    siAgrego ⟵ calcularCIDM(adyacencia, i+1, conjNodos, optimo);
    elimino el nodo i de conjNodos;
    return el mejor entre siNoAgrego y siAgrego;
end
return 0;
```

Algorithm 1: algoritmo exacto

2.2. Complejidad Temporal

Al ser un algoritmo de $Backtracking$ o fuerza bruta, tiene una complejidad exponencial. En este caso, la misma es de $O(2^n)$, siendo n la cantidad de nodos del grafo.

Se llega a dicha complejidad dado que para cada nodo, tenemos que preguntarnos qué ocurre tanto si lo agregamos al conjunto, como si no.

Dicho de otra forma, vamos a recorrer todos los conjuntos dentro del conjunto de partes del total de nodos de V. El cardinal de dicho conjunto de partes es 2^n.

Para cada uno de los subconjuntos, el algoritmo verifica si cumple la función `esIndependienteMaximal()`, la cual tiene una complejidad en peor caso de $O(n^2)$. De esta forma, nuestro algoritmo tiene complejidad $O(2^n * n^2)$, lo que es equivalente a $O(2^n)$.

Al tener en cuenta la poda utilizada, se puede ver que la misma no disminuye la complejidad teórica planteada dado que en el peor caso, podría haber que recorrer completamente el conjunto de partes. De todas formas, dicha poda mejora notablemente los tiempos de ejecución del algoritmo, como veremos más adelante, ya que descarta las soluciones peores que la óptima hasta el momento.

2.3. Experimentación

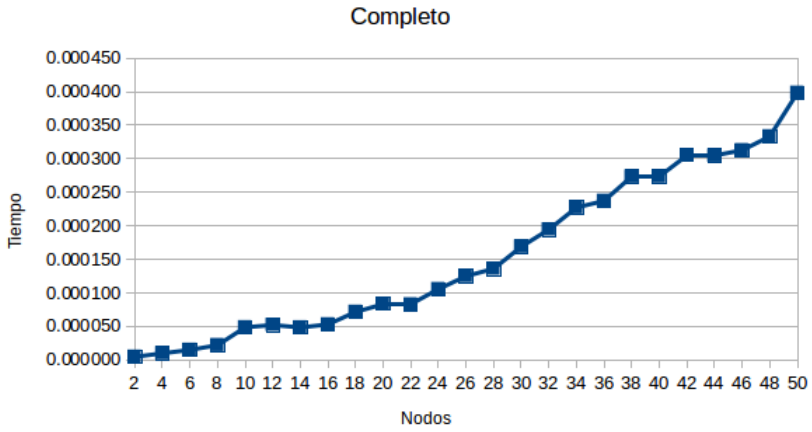
Para realizar la experimentación, se genereron grafos específicos, que nos permitieron ver el funcionamiento de nuestro algoritmo. Las conexiones entre los nodos de cada grafo son aleatorias, pero respetando la cantidad de ejes que nosotros definamos previamente.

Las instancias con tiempos de ejecución bajos fueron corridas 100 veces, obteniéndo luego un promedio de todas, con el fin de eliminar outliers. Aquellas instancias que tardaban mucho más tiempo, determinamos que los outliers no modificaban en forma considerable, por lo que no nos pareció necesario realizarlas reiteradas veces.

2.3.1. Mejor caso

De acuerdo a cómo fue planteado nuestro algoritmo, se puede ver que el mejor caso posible será cuando los grafos pasados por parámetro sean completos. Esto es así debido a la poda implementada: cuando considera al primer nodo, encuentra una posible solución y luego descarta rápidamente todas las demás, ya que una solución con un solo nodo será necesariamente una solución óptima.

Tabla

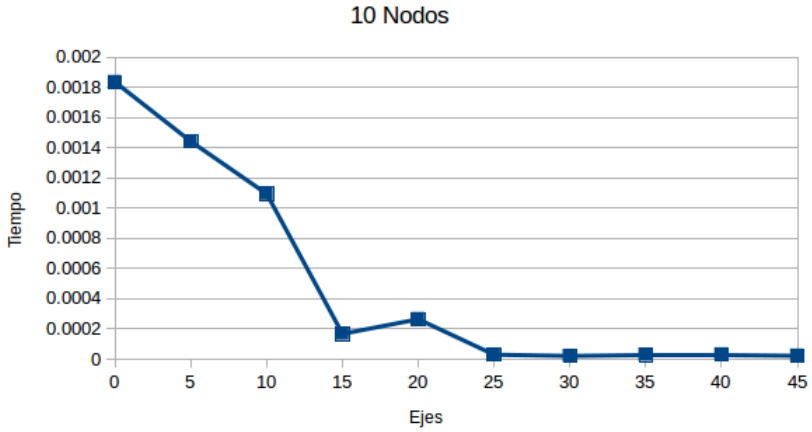


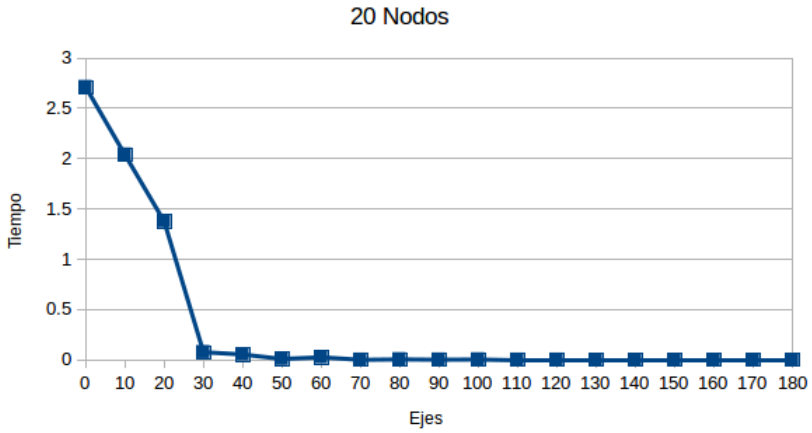
Se puede ver que el gráfico no tiene una tendencia exponencial, lo que era esperable ya que encontrará la solución en el primer nodo, pero luego deberá descartar los siguientes n nodos, teniendo una complejidad de $O(n^2)$.

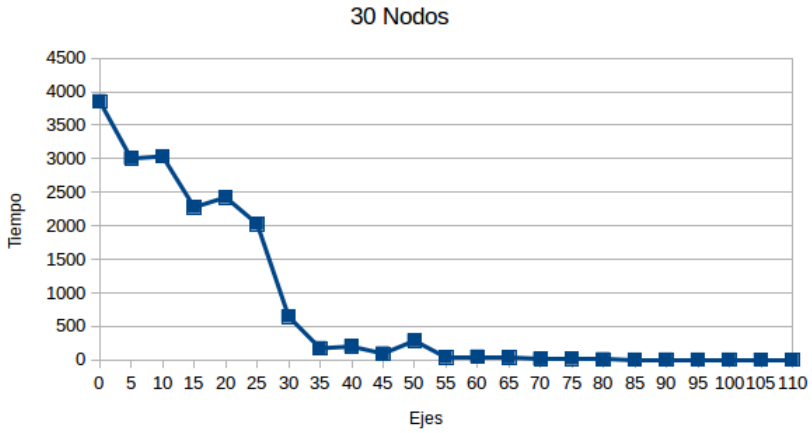
2.3.2. Nodos fijos

Para continuar viendo el comportamiento del algoritmo, decidimos realizar experimentos fijando la cantidad de nodos y variando la cantidad de ejes.

Los tiempos de ejecución para nodos fijos en 10, 20 y 30 fueron los siguientes:
TABLA



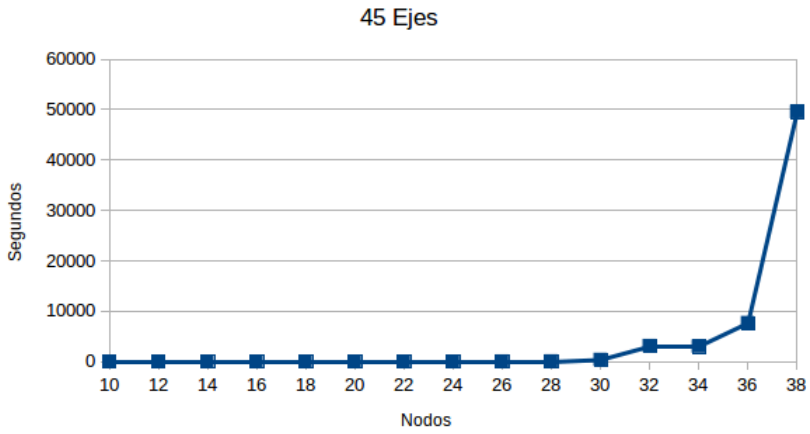


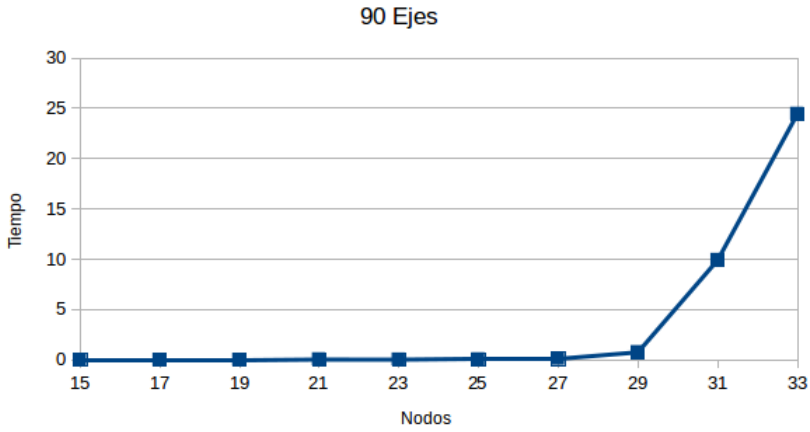


Como podemos ver, en todos los casos ocurre algo similar: cuando no tienen ejes la ejecución es más lenta, ya que debe recorrer todas las posibles opciones del conjunto de partes, sin lograr descartar ninguna. A medida que se van agregando ejes, los tiempos van decreciendo ya que gracias a la poda implementada, no es necesario recorrer todas las posibles soluciones.

2.3.3. Ejes fijos

En esta sección podremos ver el comportamiento exponencial del ejercicio, ya que se mantuvo fija la cantidad de ejes, pero fue aumentando la cantidad de nodos. Los resultados de los tiempos de ejecucion, con 45 y 90 ejes fijos fueron:
TABLA





Como preveíamos, se puede ver que los valores van aumentando exponencialmente en ambos casos, lo cual nos indica que la complejidad temporal depende directamente de la cantidad de nodos que tenga el grafo. De todas formas, en algunos casos es posible ver el efecto que tiene la poda, ya que el tiempo de ejecución no aumentó exponencialmente, por ejemplo, entre los casos con 32 y 34 nodos, con 45 ejes fijos. Veremos que esto no ocurre en el peor caso.

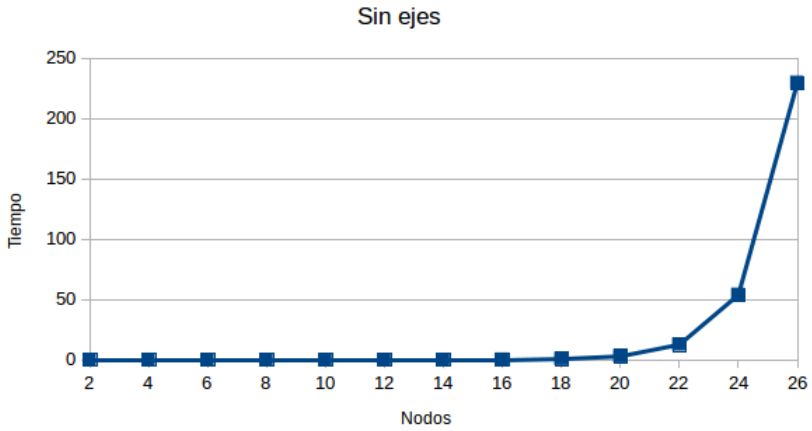
2.3.4. Sin ejes

El peor caso para nuestro algoritmo se da cuando el grafo pasado por parámetro no cuenta con ningún eje. Esto es así porque deberá revisar todos los posibles conjuntos dentro del conjunto de partes, inten-

tando encontrar uno mejor que el que utiliza todos los nodos. Claramente esto no es posible, ya que al no contar con ejes, aquella es la única solución posible.

Los tiempos de ejecución para los distintos grafos sin ejes fueron:

TABLA



Aquí se ve claramente que los tiempos aumentan exponencialmente en cada paso, lo que confirma nuestra complejidad temporal teórica de $O(2^n)$.

3. Heurística Constructiva Golosa

3.1. Explicación

La heurística constructiva golosa busca, dado un grafo, determinar un conjunto independiente dominante mínimo, eligiendo en cada iteración, la mejor opción según el criterio definido

Nuestro criterio se basa en el grado de los nodos. Dado que el grafo nunca se modifica, podemos ordenar un conjunto con todos los nodos del grafo en función de su grado al comienzo del algoritmo.

Luego, haremos un ciclo iterando este conjunto, el primer nodo siempre será agregado al conjunto solución, y luego se procede a eliminarlo junto a sus vecinos.

Esto se traduce en elegir en cada paso el nodo de mayor grado que aún no haya sido dominado por otro con grado superior.

De esta manera, el algoritmo siempre obtiene un conjunto independiente (dado que por cada nodo que toma, elimina a sus vecinos) y dominante (dado que por cada nodo que toma, lo agrega al conjunto, y elimina a sus vecinos, es decir, que estos son adyacentes a un nodo del conjunto), pero no puede asegurar que siempre sea mínimo. Eso dependerá del grafo.

```
vector<nodoGrado>grados(adyacencia.cantNodos());
for int i = 0; i<grados.size(); ++i
do
    grados[i].nodo = i;
    grados[i].grado = adyacencia.gradoDeNodo(i);
end
sort(grados.begin(), grados.end());
while grados.size()>0 do
    unsigned int nodo = grados[0].nodo;
    optimo.push_back(nodo);
    vector<nodoGrado>::iterator iter = grados.begin();
    while iter != grados.end() do
        if adyacencia.sonVecinos(nodo, iter->nodo) or nodo == iter->nodo then
            iter = grados.erase(iter);
        else
            iter++;
        end
    end
end
```

Algorithm 2: heurística greedy

3.2. Complejidad Temporal

En lo que respecta a la complejidad temporal, demostraremos a continuación que la misma es $O(n^2)$, con n la cantidad de nodos del grafo.

Recordemos que el algoritmo ordena los nodos de mayor grado a menor grado, luego toma el primero de dicha lista, lo agrega al conjunto solución, y lo elimina de la lista junto a sus adyacentes. Luego repite el proceso, tomando el siguiente nodo de la lista de nodos restantes.

Dicho esto, definiremos a $f(i, n) : \mathbb{N} \to \mathbb{N}$ como:

$f(i, n)$ = Cantidad de operaciones en el peor caso, teniendo n nodos totales y faltando eliminar i.

Es decir,

- $f(0, n) = c$, dado que no falta eliminar ningun nodo, el algoritmo termina, con c alguna cantidad constante de operaciones.
- $(\forall i \in 1...n-1) f(i, n) = h * (i-1) + f(i-1, n) + c$, con h la cantidad de vecinos del nodo que estoy viendo, c alguna cantidad constante de operaciones, y $f(i-1, n)$ es el llamado recursivo.

Expliquemos que significa cada monomio:

- $h * (i-1)$: En el peor de los casos, recorre por cada nodo de la lista de adyacencia del nodo tomado, a los demas nodos sin marcar (sin incluir el nodo tomado).
- $f(i-1, n)$: En el peor de los casos, no hay nodos de la lista de adyacencia, que no hayan sido eliminados aún.
- c: Alguna cantidad constante de operaciones.

Entonces, querremos demostrar que $(\forall i \in 1...n-1) f(i, n) \leq k * i * n + c$ (con k y c alguna constante), pues al momento de ejecutar el algoritmo, se tienen n nodos, y faltan eliminar n nodos, siendo la complejidad del mismo, $O(f(n, n)) = O(n^2)$; y dado que la cantidad de nodos por borrar no puede ser mayor que la cantidad total de nodos, es correcto pedir que $0 \leq i < n$.

Teorema

Dado $f(i,n) : \mathbb{N} \to \mathbb{N}$ definida como

$f(i,n)$ = Cantidad de operaciones en el peor caso, teniendo n nodos totales y faltando eliminar i.

$f(0,n) = c$

$f(i,n) = h * (i-1) + f(i-1,n) + c$

Luego, $(\forall i \in 1...n-1) f(i,n) \leq k * i * n + c$.

Demostración

Sea $P(i) = f(i,n) \leq k * i * n + c$, demostraremos por inducción global, que $(\forall i \in 1...n-1)P(i)$

Entonces,

Casos base:

- $f(0,n) = c$
- $f(1,n) = h * (1-1) + c = c \leq k * 1 * n + c$
  Dado que tengo n nodos, y solo falta eliminar uno. k es alguna cantidad constante de operaciones para eliminar el nodo y finalizar el algoritmo.

Paso inductivo:

$$\underbrace{P(1) \wedge P(2) \wedge ... \wedge P(m-1)}_{\text{Hipótesis inductiva}} \Rightarrow \underbrace{P(m)}_{\text{Tesis inductiva}}$$

Es decir, que por hipótesis inductiva, podemos suponer que vale $f(r,n) \leq k * r * n + c$, con $r \leq m-1$, y queremos ver que $f(m,n) \leq k * m * n + c$.

Luego,

$$\begin{aligned} & f(m,n) = h * (m-1) + f(m-1,n) + c \\ \overset{HI}{\Longrightarrow} & f(m,n) \leq h * (m-1) + n * (m-1) + c \end{aligned}$$

Dado que la cantidad de vecinos adyacentes esta acotada por la cantidad de nodos totales, $0 \leq h < n$, luego

$$\begin{aligned} & \leq n * (m-1) + n * (m-1) + c \\ & \leq n * m + n * m + c \\ & = 2 * n * m + c \\ & \leq \underbrace{k * n * m + c}_{\text{Tesis inductiva}} \text{ En particular, con } k = 2 \text{ en este caso} \end{aligned}$$

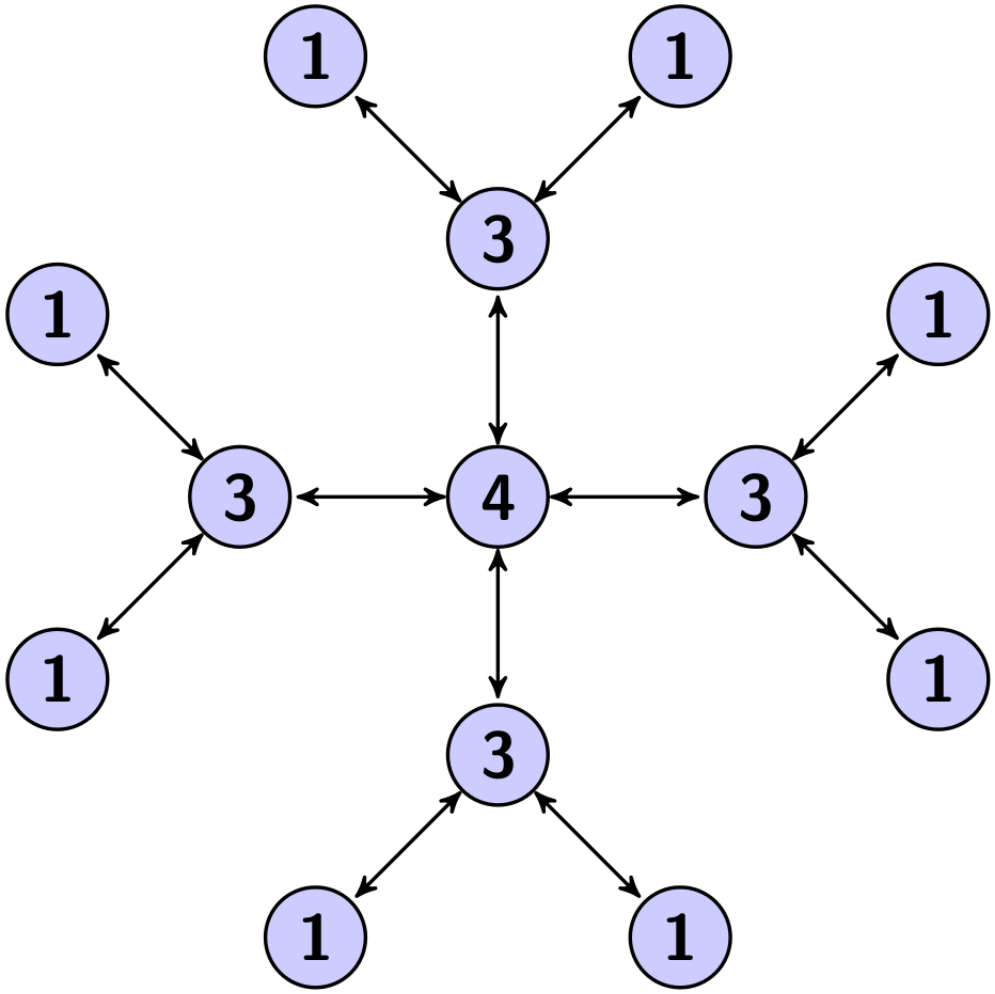
■

Entonces, $f(i,n) \leq c * i * n + k$, para algún c y k constantes. Dado que el algoritmo ejecuta, en peor caso, $f(n,n)$ operaciones, su complejidad esta acotada por $f(n,n)$, por lo tanto, tiene complejidad $O(n^2)$.

También es adecuado decir, que el algoritmo tiene complejidad $\Omega(n * log(n))$, dado que debe ordenar los nodos de acuerdo al grado de cada uno, y cualquier algoritmo de ordenamiento basado en el árbol de decisiones no posee mejor complejidad que la mencionada.

3.3. Comparación de resultados con solución óptima

La heurística constructiva golosa fallará en encontrar la solución óptima para todos los casos. Particularmente, existen grafos para los cuales la heurística fallará siempre.

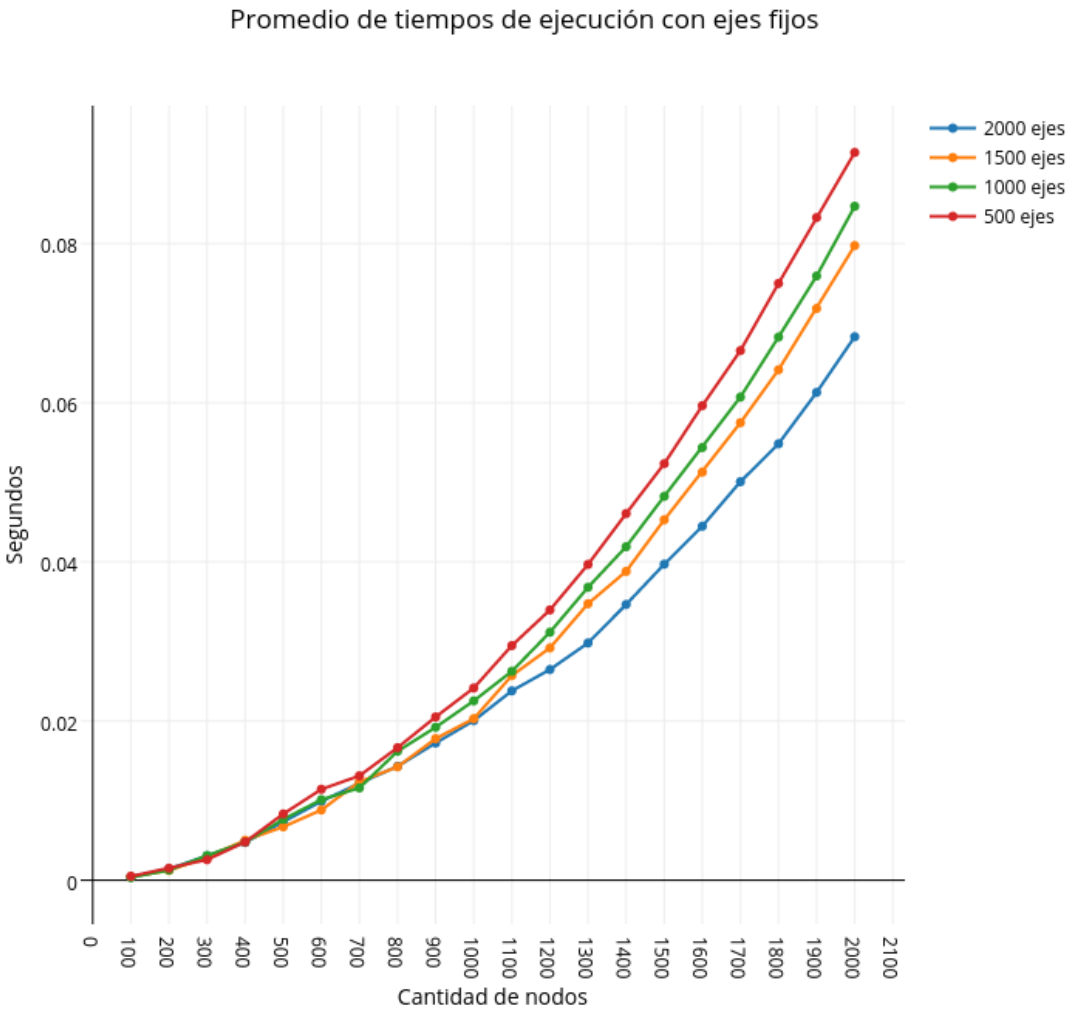


En este ejemplo, el greedy tomará como primer nodo, al nodo del centro, y eliminará a todos sus adyacentes. Luego, tomará todas las componentes conexas restantes. El resultado final será una solución con $(m-2)*m$ versus m de la solución óptima, con m igual al grado del nodo central.

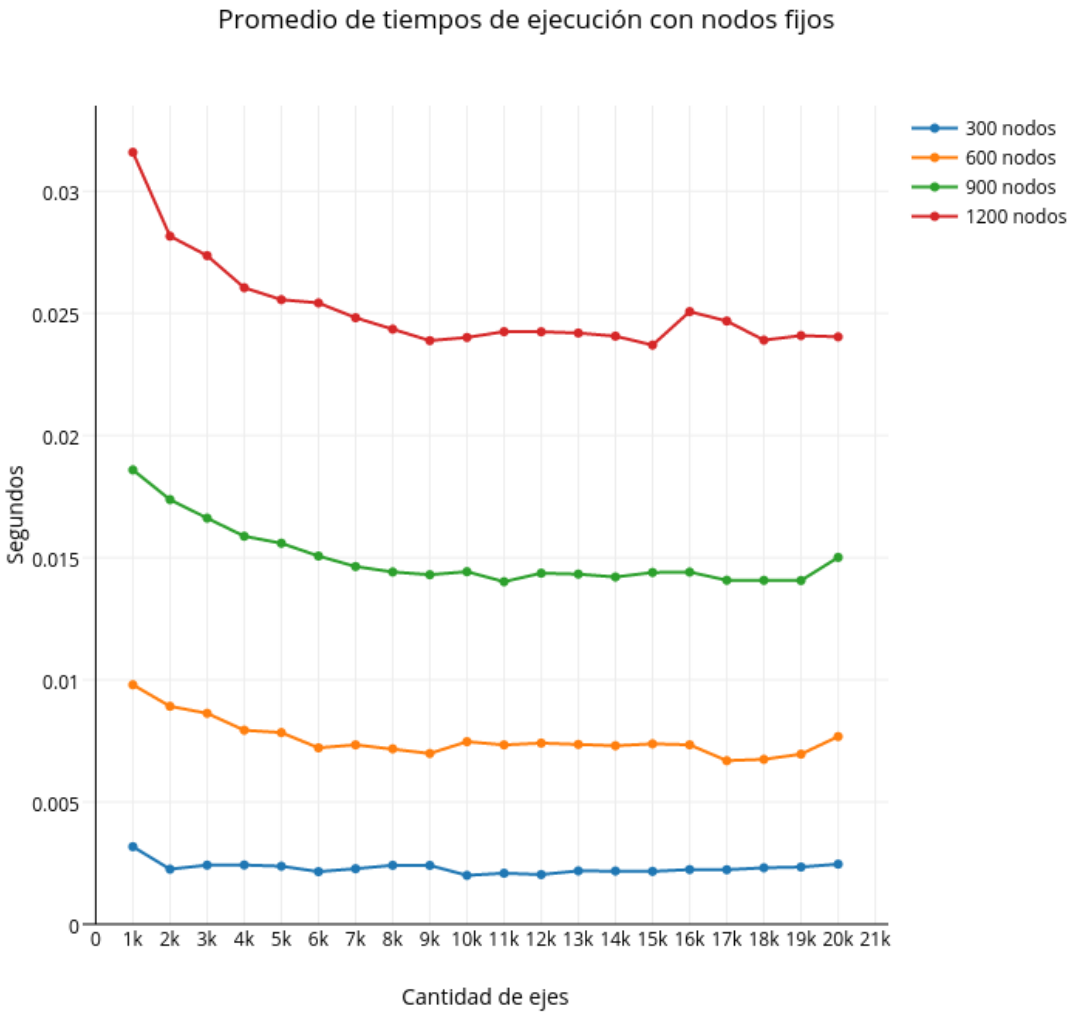
3.4. Experimentación

Para la experimentación, se realizaron experimentos sobre grafos generados de manera aleatoria, fijando nodos y variando ejes, y viceversa.

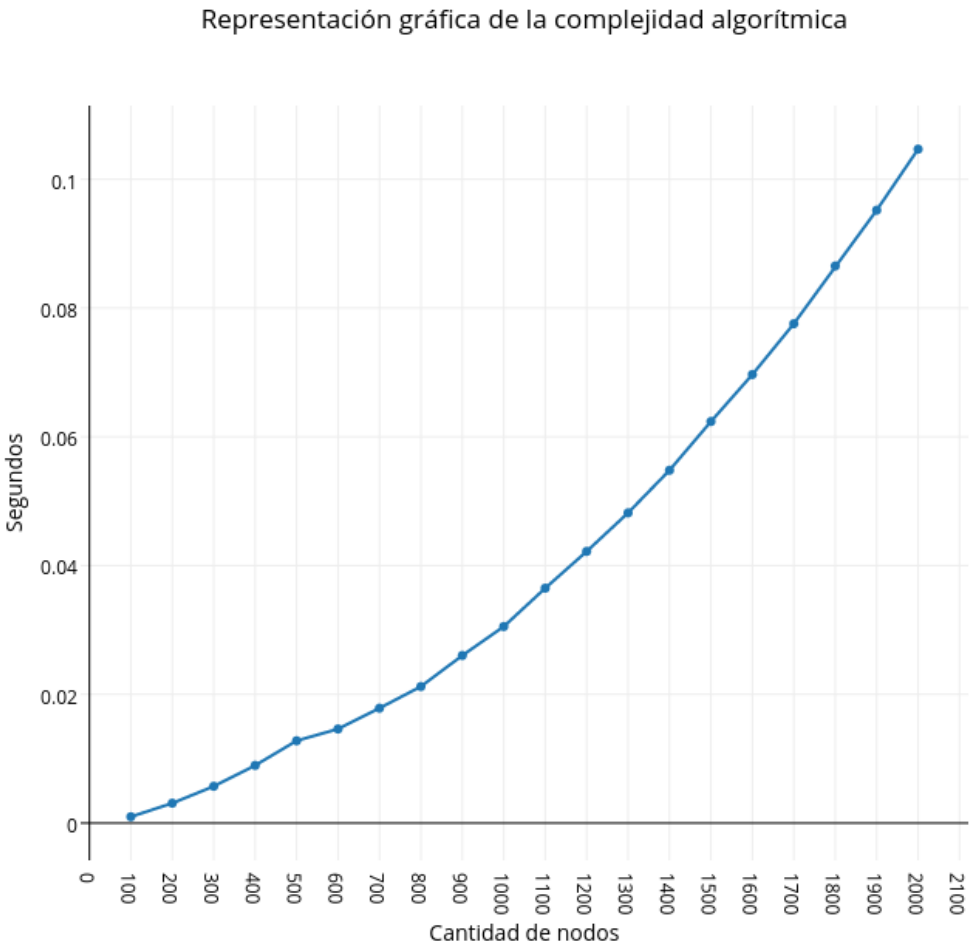
A continuación se adjuntan los gráficos con los resultados:



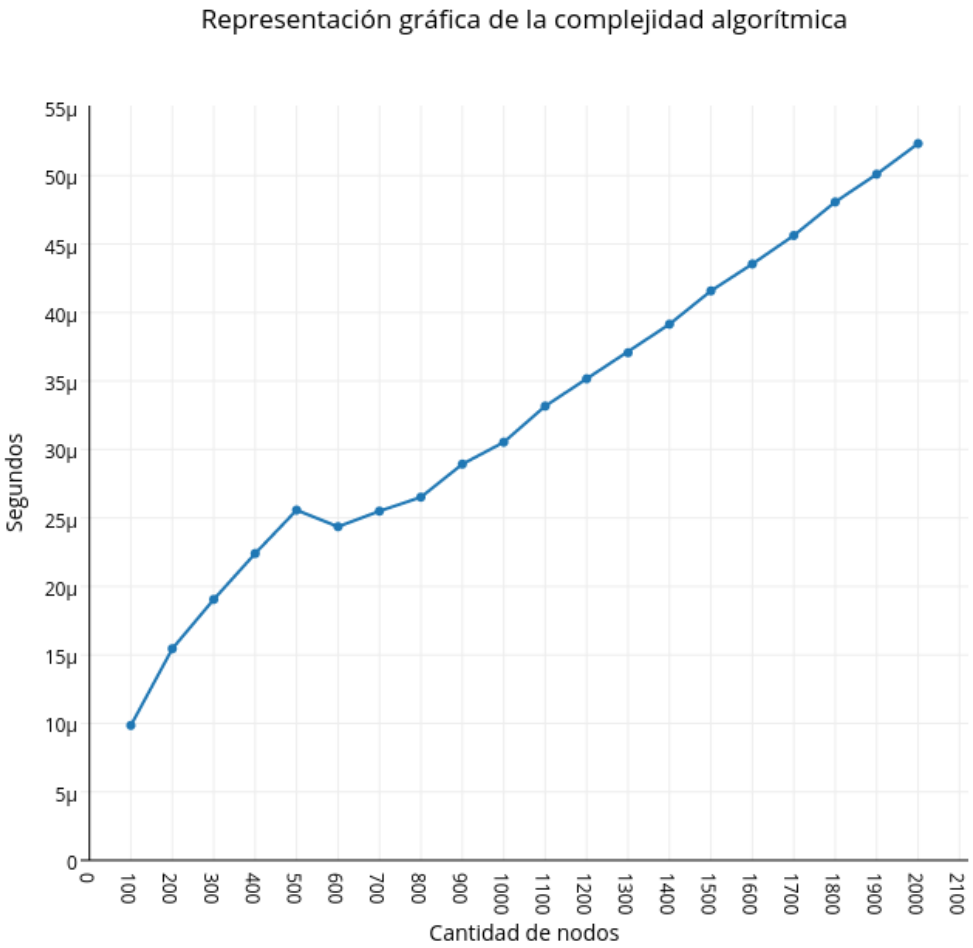
Como se puede observar, los tiempos aumentan a medida que disminuyen la cantidad de ejes. Esto se debe a que a menor cantidad de ejes, menor cantidad de nodos se eliminan de la lista de nodos que itera el algoritmo.



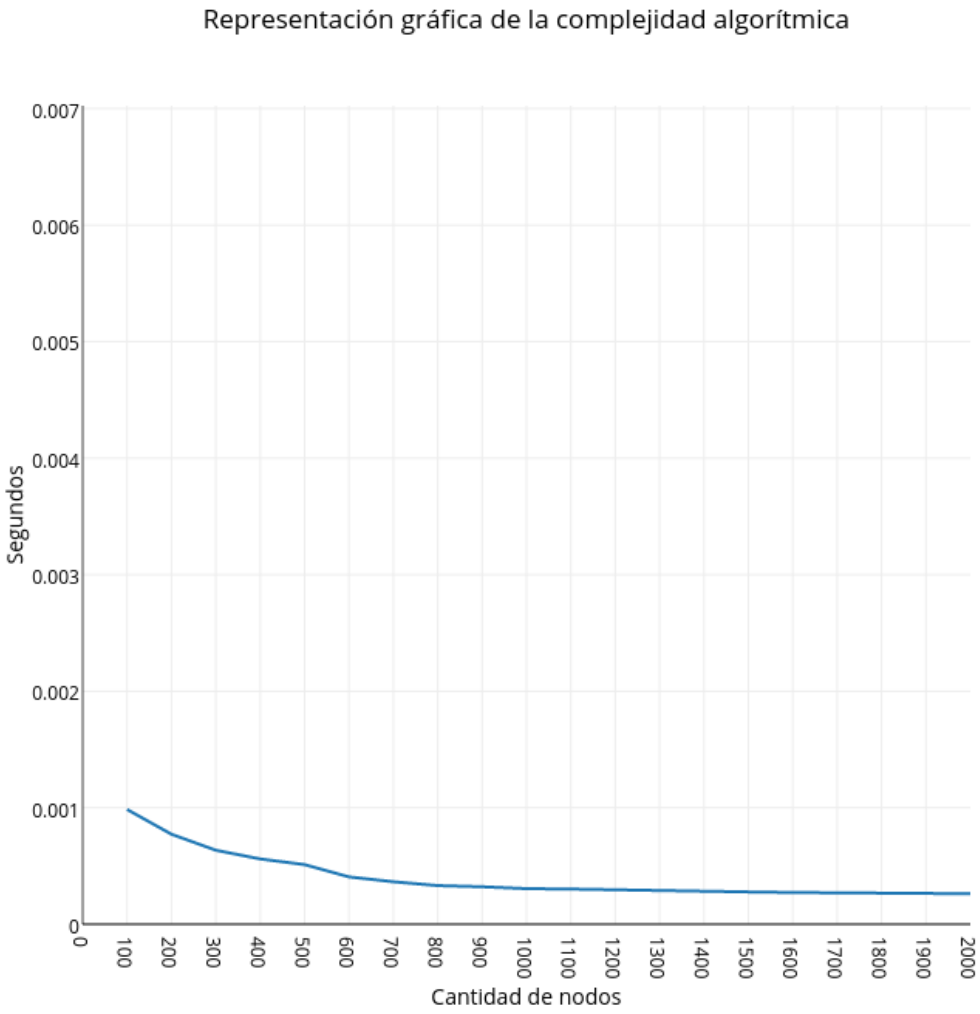
En este gráfico, se puede ver que si bien la cantidad de ejes (como fue mencionado en el gráfico anterior) afecta los tiempos, a mayor cantidad de ejes se puede observar que el verdadero limitante es la cantidad de nodos. Aquí se aprecia que la complejidad teórica se corresponde, en cuanto a que depende de la cantidad de nodos, y no de la cantidad de ejes.



Resultó interesante graficar, a su vez, algún caso en el que se refleje la complejidad algorítmica. Para ello, se tomo el grafo sin ejes, es decir, todas componentes conexas, puesto que el algoritmo deberá recorrer, por cada nodo, el resto de todos los nodos preguntando si son vecinos. Aquí se puede apreciar que efectivamente, los tiempos incrementan de manera cuadrática.



Para asegurarnos de que efectivamente la función corresponde a la familia de funciones cuadráticas y no a otra, se procedió a dividir los resultados por el tamaño de la entrada. Se puede apreciar que el gráfico se aproxima a una función lineal.



Finalmente, se procedió a dividir los resultados (ya divididos) por el tamaño de la entrada nuevamente. Efectivamente, el resultado tiende a una constante, lo que termina demostrando de manera empírica, la complejidad teórica explicada.

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

Para cada solución factible s ϵ S se define N(s) como el conjunto de “soluciones vecinas” de s. Un procedimiento de busqueda 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 ϵ S una solución inicial

Mientras exista s ϵ N(s) con f (s) $>$ f (s)

s $\leftarrow$ s

4.1. Explicación

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

4.1.1. 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 3.

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.

4.1.2. 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 BLABALLABLA

4.2. Complejidad Temporal

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

4.2.1. 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 *for*s 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: $O(grado(i)+grado(j))$. Cada par se recorre una única vez. Por lo tanto, los dos *for*s anidados van a iterar (considerando a (n-1) como el último nodo'):

Cuando sea el par de nodos (0,1) : grado(0)+grado(1)

Cuando sea el par de nodos (0,2) : grado(0)+grado(2)

...

Cuando sea el par de nodos (0,n-1) : grado(0)+grado(n-1)

Cuando sea el par de nodos (1,2) : grado(1)+grado(2)

Cuando sea el par de nodos (1,3) : grado(1)+grado(3)

...

Cuando sea el par de nodos (1,n-1) : grado(1)+grado(n-1)

...

Cuando sea el par de nodos (n-2,n-1) : grado(n-2)+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:

grado(0)*(n-1) + grado(1)*(n-1) + ... + grado(n-1)*(n-1)

lo que es equivalente a:

[grado(0)+grado(1)+ ... + grado(n-1)]*(n-1)

La complejidad en peor caso se obtiene cuando los grados de todos los nodos son maximos, por lo tanto se trata de un grafo completo. Donde vale que 2*#ejes = grado(0)+grado(1)+ ... 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

4.2.2. dameTernasVecinasComun

La función dameTernasVecinasComun funciona de manera análoga a la descripta en el inciso 4.2.1.

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 *for*s 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 grado(i)+grado(j)+grado(k).

Dada la tupla (0,1,2) el costo será: grado(0)+grado(1)+grado(2)

Dada la tupla (0,1,3) el costo será: grado(0)+grado(1)+grado(3)

...

Dada la tupla (0,1,n-1) el costo será: grado(0)+grado(1)+grado(n-1)

Dada la tupla (0,2,3) el costo será: grado(0)+grado(2)+grado(3)

...

Dada la tupla (0,n-2,n-1) el costo será: grado(0)+grado(n-1)+grado(n-2)

Dada la tupla (1,2,3) el costo será: grado(1)+grado(2)+grado(3)

...

Dada la tupla (n-3,n-2,n-1) el costo será: grado(n-3)+grado(n-2)+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 $O(n^4)$.

4.2.3. localCIDM

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

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 3.

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

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

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

En la siguiente iteración se tienen $n - 1 - (grado(0))$ elementos en el vector `nodos`.

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

Considero la notación vecinos(i) como la cantidad de nodos pertenecientes al vector `nodos` durante la iteración i que sean adyacentes al nodo i.

En la iteración i, el vector va a contener $n - 1 - grado(0) - ... - 1 - vecinos(i)$

Donde en el peor caso, también, deberá ser vecinos(i) con valor mínimo. 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 sólo en una unidad por iteración.

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

Las funciones usadas son:

push_back (costo $O(n)$ amortizado)

listaAdy::sonVecinos (costo $O(min(grado(nodoA), grado(nodoB)))$)

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

A continuación, se introducen optimizaciones del algoritmo.

- En primer lugar, si la solución óptima actual posee tamaño 1 no va a encontrarse una solución mejor, por consiguiente se devuelve. (Costo $O(1)$)
- En segundo lugar, si la solución óptima actual posee tamaño 2 la única solución que puede ser mejor es la que posee un sólo nodo. Se chequea si existe una solución con un sólo nodo (costo $O(n)$). Si existe, la solución óptima es la de un sólo nodo y se devuelve sino era la solución con dos nodos. Funciones usadas: assign, listaAdy::gradoDeNodo y listaAdy::cantNodos (todas con costo $O(1)$)
- En tercer lugar, si la solución óptima actual posee tamaño n debe ser la solución que se retorne. Ya que la única manera de que todos los nodos sean parte del conjunto solución 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 4.2.1), 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 $O(n.(n-1)/2) = O(n^2)$).

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, se quitaron 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 $O(n^5)$.

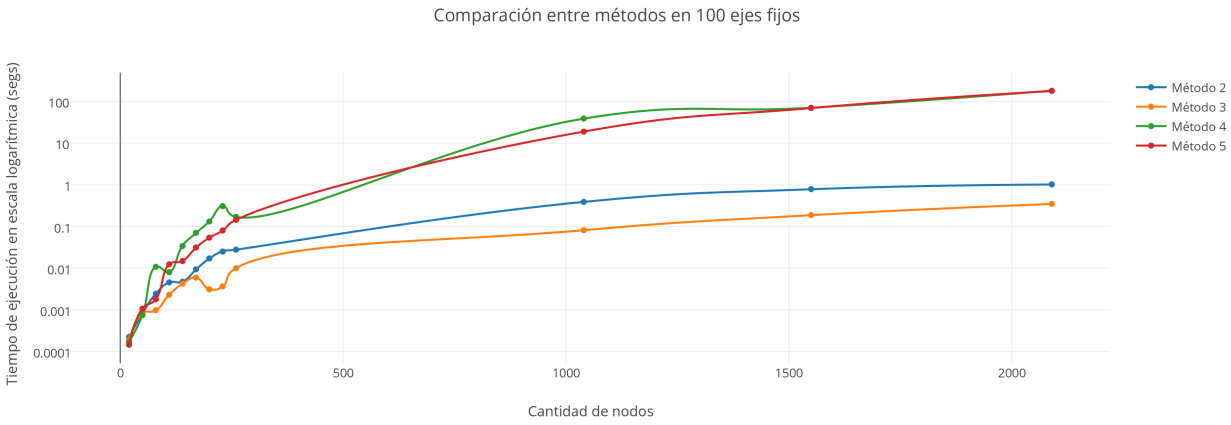
En los casos de **ejecución 4 y 5**, el procedimiento será similar.

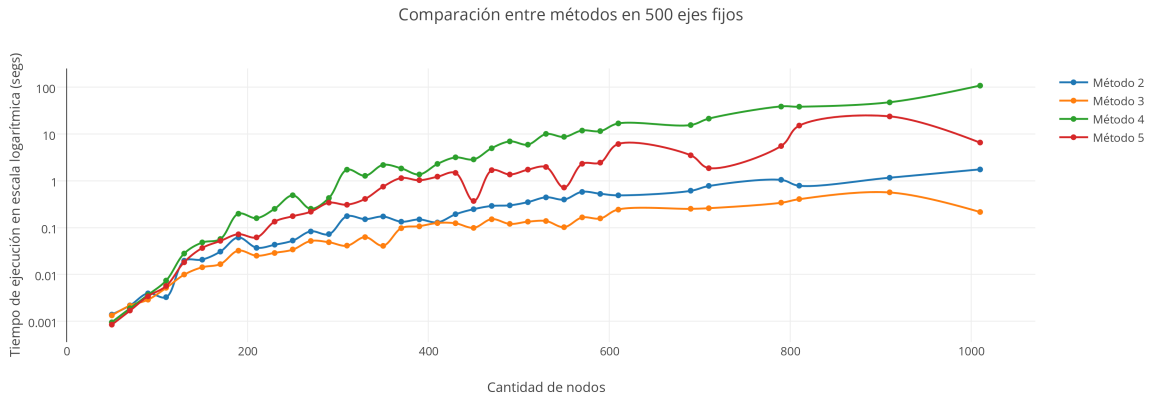
Primero buscará ternas con vecinos en común (`dameTernasVecinasComun`) con un costo de $O(n^4)$.

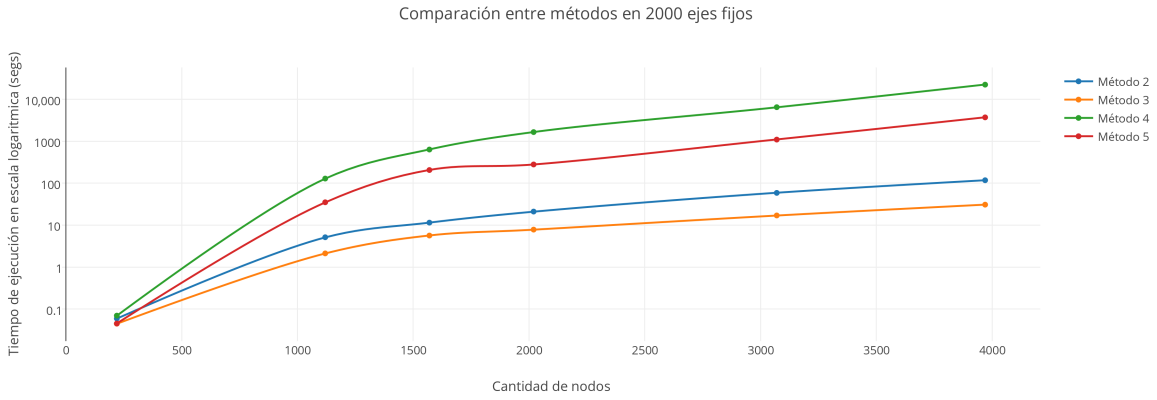
Pero el costo de cada iteración del ciclo mayor será de $O(n^4)$, por lo explicado anteriormente, con lo cual el costo total del algortimo es $O(n^5)$ también para esta vecindad.

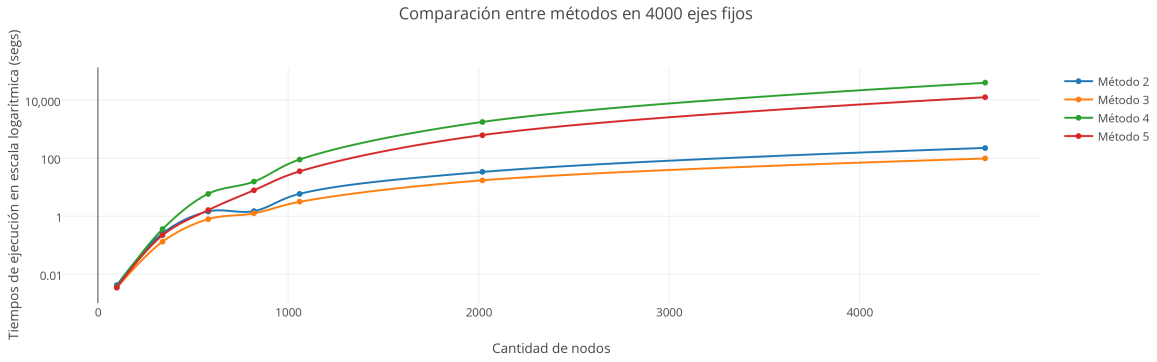
4.3. Experimentación

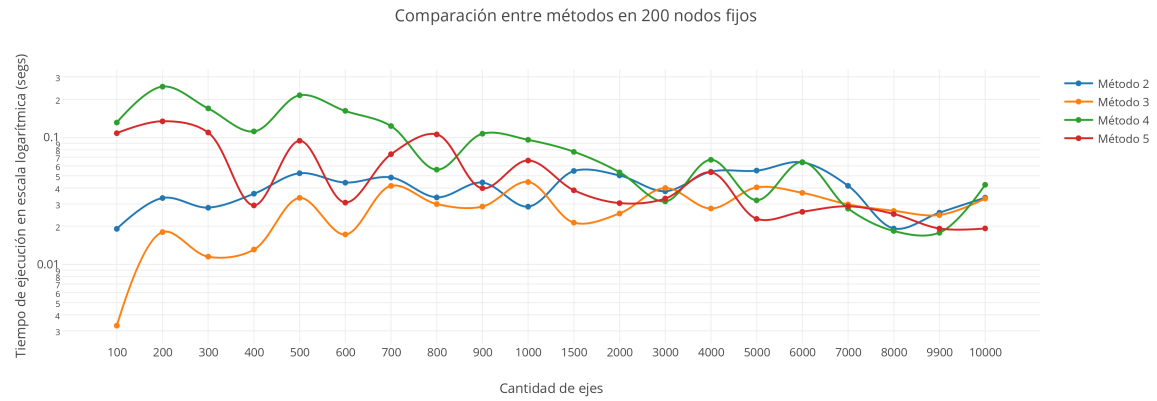
4.3.1. Análisis de tiempos de ejecución

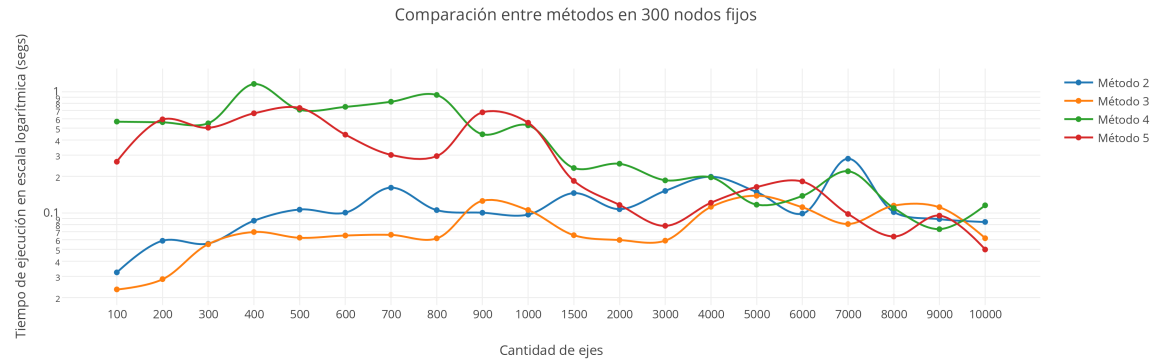


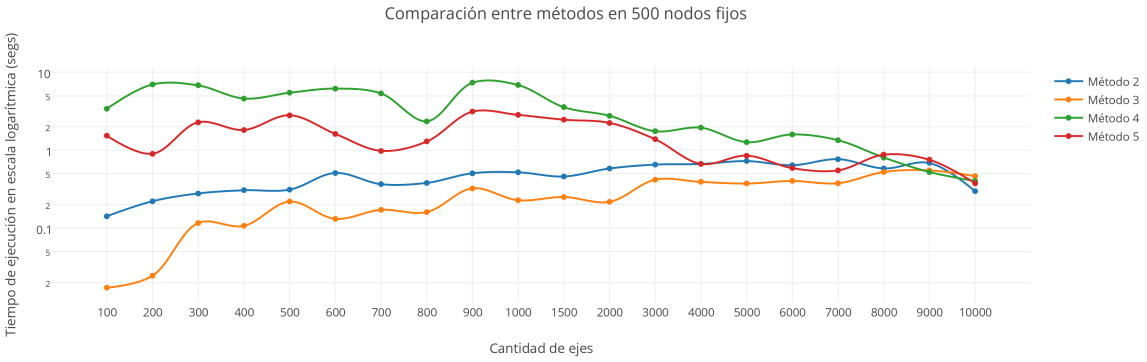


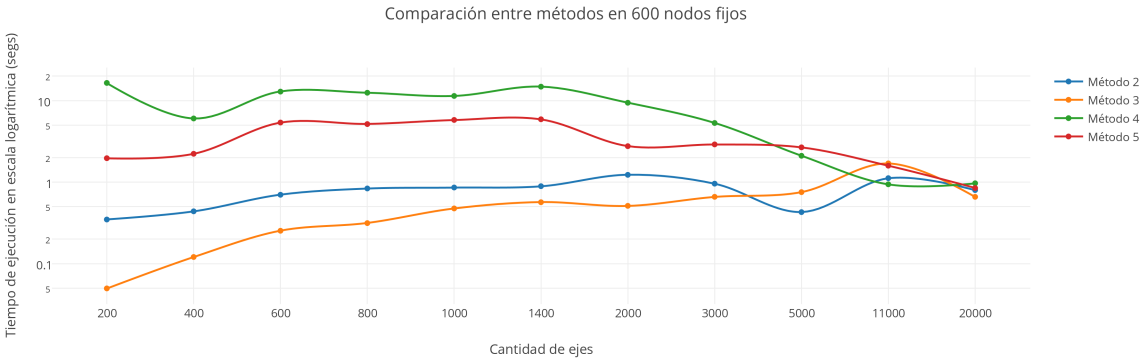


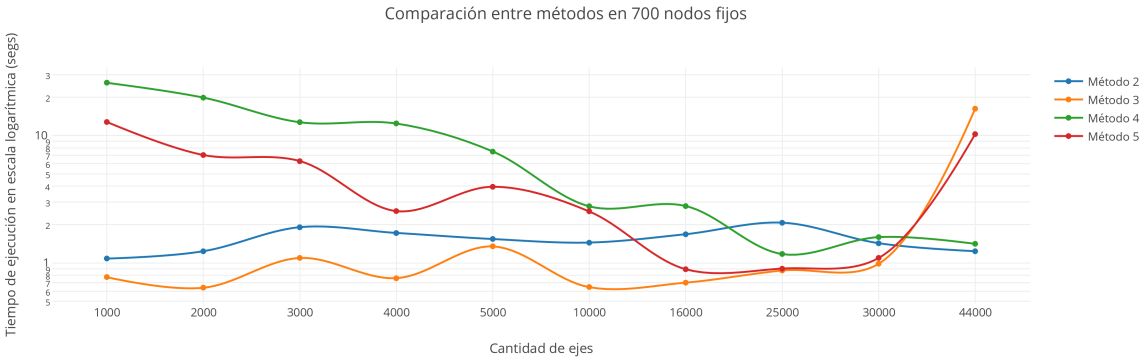

Comparación entre métodos en 4000 ejes fijos
Tiempos de ejecución en escala logarítmica (segs)
10,000
100
1
0.01
0
1000
2000
3000
4000
Cantidad de nodos
Método 2
Método 3
Método 4
Método 5


Comparación entre métodos en 200 nodos fijos
Tiempo de ejecución en escala logarítmica (segs)
0.1
0.01
100
200
300
400
500
600
700
800
900
1000
1500
2000
3000
4000
5000
6000
7000
8000
9900
10000
Cantidad de ejes
Método 2
Método 3
Método 4
Método 5


Comparación entre métodos en 300 nodos fijos
Tiempo de ejecución en escala logarítmica (segs)
0.1
100
200
300
400
500
600
700
800
900
1000
1500
2000
3000
4000
5000
6000
7000
8000
9000
10000
Cantidad de ejes
Método 2
Método 3
Método 4
Método 5

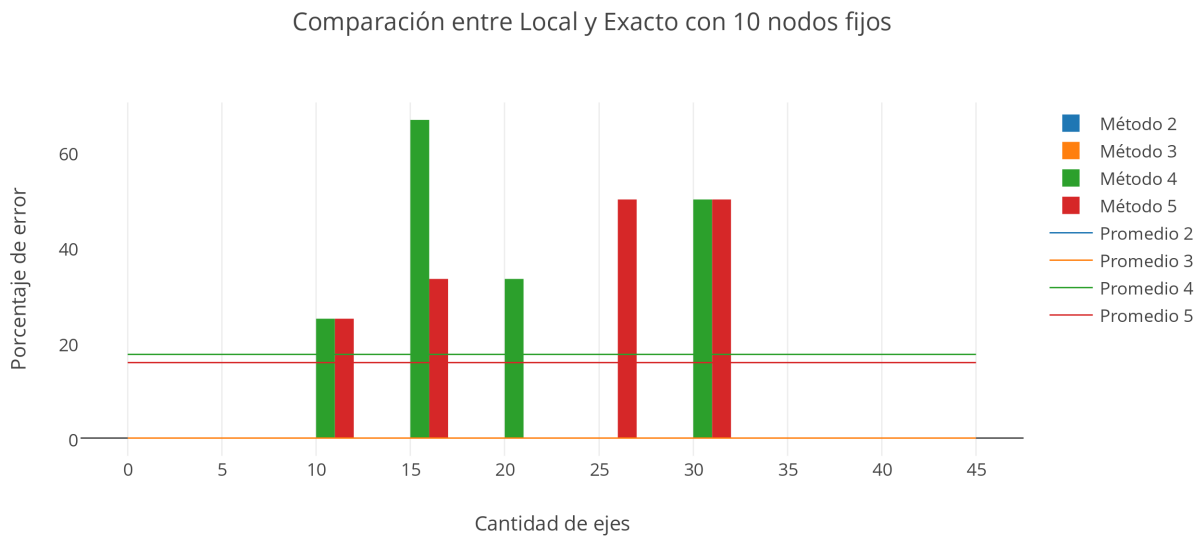


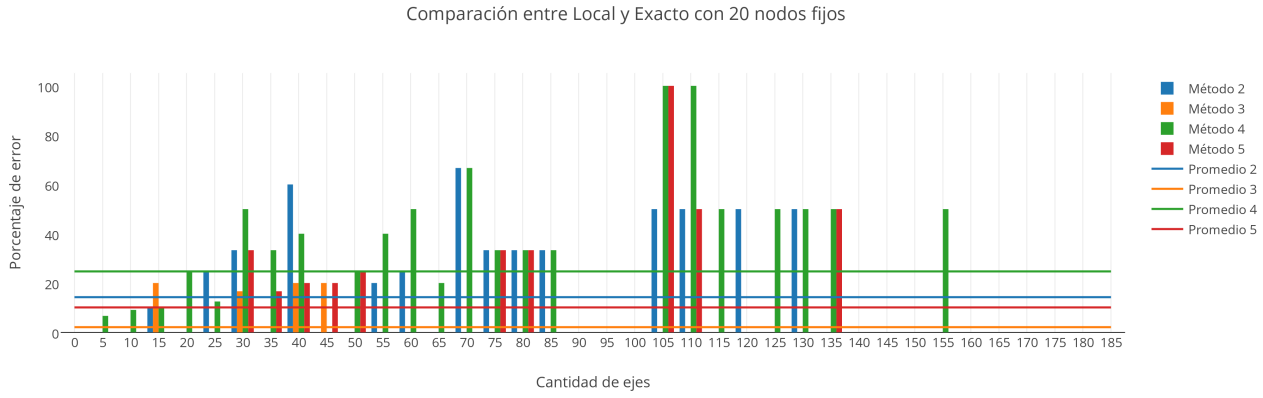


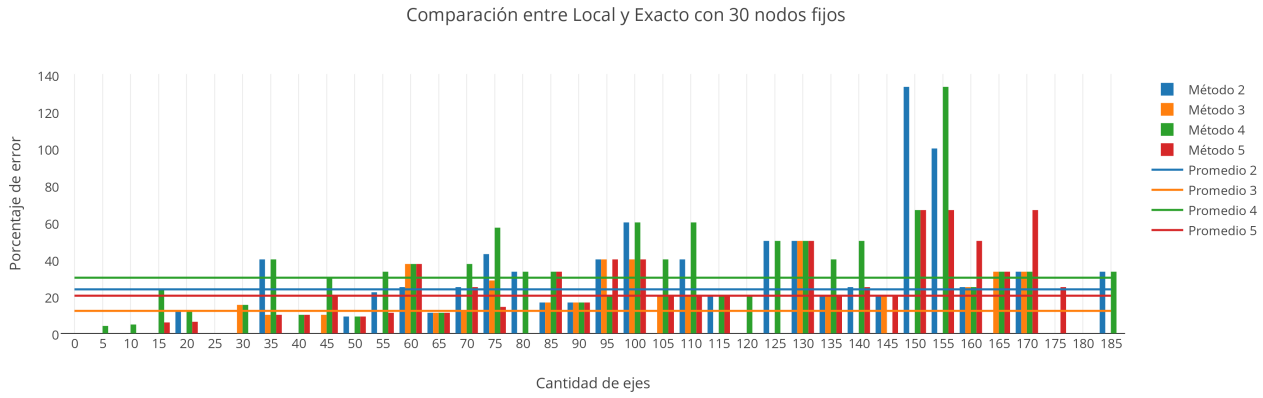


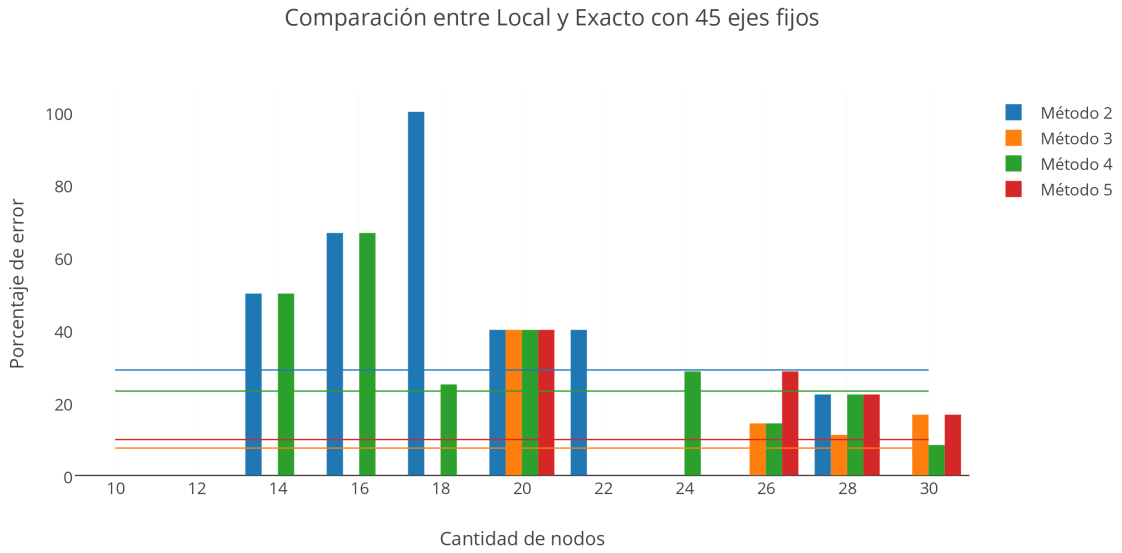
4.3.2. Contrastación empírica de la complejidad

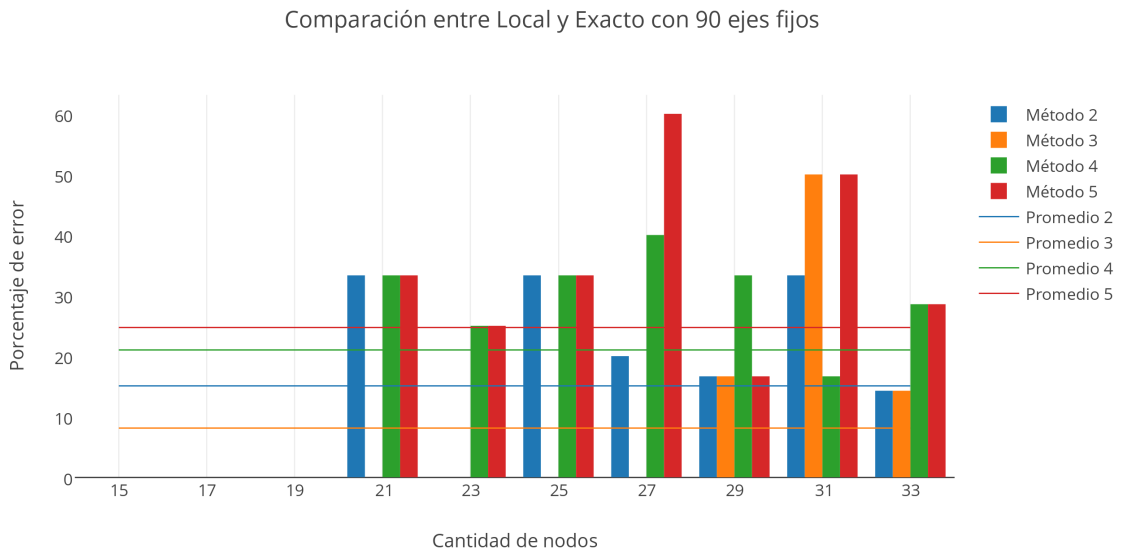
4.3.3. Comparación soluciones Local vs Exacto

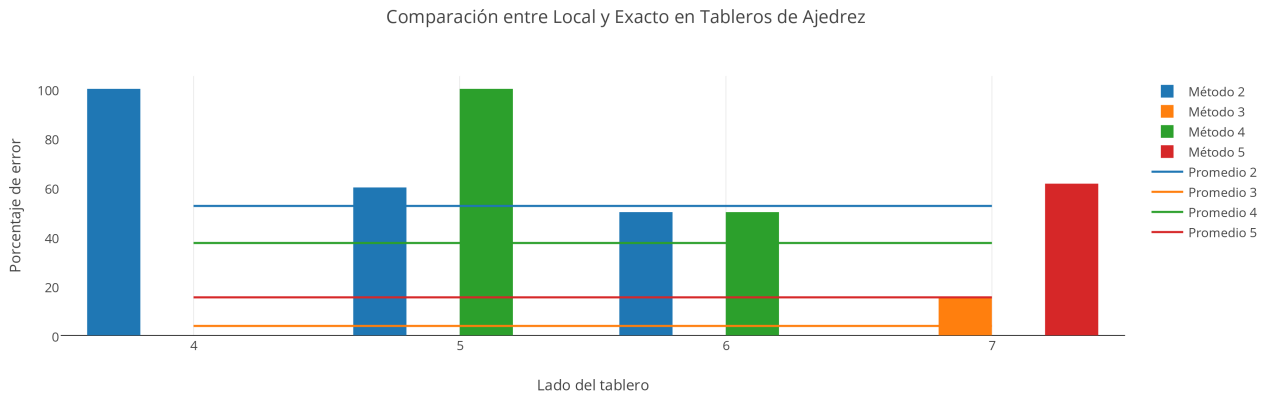







Comparación entre Local y Exacto con 45 ejes fijos
Porcentaje de error
Cantidad de nodos
Método 2
Método 3
Método 4
Método 5
0
20
40
60
80
100
10
12
14
16
18
20
22
24
26
28
30


Comparación entre Local y Exacto con 90 ejes fijos
Porcentaje de error
Cantidad de nodos
Método 2
Método 3
Método 4
Método 5
Promedio 2
Promedio 3
Promedio 4
Promedio 5
0
10
20
30
40
50
60
15
17
19
21
23
25
27
29
31
33



4.3.4. Elección de versión óptima

5. Metaheurística GRASP

5.1. Explicación

La metaheurística *Greedy Randomized Adaptive Search Procedure* (**GRASP**), es una mezcla de las dos heurísticas previas (vistas en 3 y 4). Dicho de manera simple: genera un punto de partida de forma golosa para el algoritmo de búsqueda local.

La distinción de este algoritmo radica en cómo se construye "*golosamente*" la solución inicial.

Como la sigla lo indica, consiste en un algoritmo *Goloso Randomnizado*. Es decir que se escogen candidatos a solución inicial de una manera golosa ligeramente distinta a la ultilizada en la sección anterior. El método *Greedy* de la sección 3 escoge a los nodos que van a pertenecer al conjunto solución, de a uno siempre eligiendo al que tiene mayor grado. En cambio, en este caso por cada paso no se elige al nodo de mayor grado sino que se elige uno al azar entre los que "mejor grado" tienen.

Hablar de "mejor grado" nos obliga a dar un criterio para ello, lo que da pie a la definición de la *Restricted Candidate List* (**RCL**), que es el conjunto de candidatos elegibles para la solución base.

La ***solución inicial*** se puede formar de diversas maneras. En todos los casos, se añade de a un nodo al conjunto solución hasta que se forme una solución válida. Tres formas para determinar la elección por nodo son:

- Elegir un nodo entre los $alpha\,\%$ nodos que tengan mayor grado hasta completar una solución válida.
- Elegir un nodo entre los $alpha$ nodos que tengan mayor grado hasta completar una solución válida.
- Elegir un nodo entre los nodos que cumplan determinada propiedad en un $alpha\,\%$ hasta completar una solución válida (como por ej: "Los nodos que tengan grado, a lo sumo, $alpha\,\%$ menor que el nodo de mayor grado").

Optamos por implementar las primeras dos opciones para generar una solución inicial. Una vez obtenida bajo el método deseado, se aplica el algoritmo de *búsqueda local* explicado en el inciso 4 sin modificaciones.

Un aspecto también diferencial de esta heurística, es que no generamos una única instancia inicial, sino que se toma una determinada cantidad de ellas (acorde al criterio de parada). Se ejecuta el algoritmo para la primer instancia y se guarda la solución como `óptima`, luego si en alguna ejecución futura se mejora (se obtiene otra solución con menor cantidad de nodos) se actualiza `óptima`.

Las ***vecindades*** utilizadas son las mismas que se utilizaron en la heurísitca de búsqueda local (4).

El ***criterio de parada*** que adoptamos fue contabilizar las ejecuciones que no produjeron mejora, de modo que sólo se ejecute una determinada cantidad de repeticiones "malas". Es decir, siempre que la ejecución otorgue una solución óptima con menos cantidad de nodos que la existente, se seguirá ejecutando. Pero si las ejecuciones no otorgan mejoras se suman al contador, de modo que al llegar a la cantidad indicada se terminará la ejecución.

Otra opción podría haber sido correr un número fijo de veces y quedarnos con la mejor solución encontrada; o también si conocieramos alguna cota, acercarnos a esta en un determinado porcentaje; o bien una combinación de todas.

5.2. Experimentación

Para analizar qué combinación de vecindades de búsqueda local y elección de solución inicial se acercan a encontrar el óptimo en distintos escenarios, se experimentó con una serie de grafos según criterios:

- Mantener los ejes fijos, variando la cantidad de nodos
- Mantener los nodos fijos, variando la cantidad de ejes
- Grafos Tablero (análogos a los del "Señor de los Caballos")

Se optó por ejecutar el algoritmo 30 veces y luego, con los tamaños de cada conjunto solución, sacar un promedio.

Es decir, obtener un promedio de la cantidad de nodos que requería la solución óptima en cada ejecución del algoritmo.

Luego, sabiendo cuántos son los nodos que pertenecen a la solución óptima (ejecutando el algoritmo exacto), divimos y obtuvimos en qué porcentaje la heurística falla en encontrar el *óptimo verdadero*.

Las siguientes tabla muestra los valores obtenidos, las columnas 2, 3 y 4 indican el criterio aplicado al grafo; vecindad 2x1 indica que la vecindad usada fue la de quitar dos nodos y agregar uno, vecindad 3x1 quitar tres y agregar uno; n representa el alfa elegido; n mejores indica que como criterio de búsqueda de solución inicial, se tomó uno entre los n mejores según el criterio greedy establecido, n % mejores indica seleccionar uno entre los que pertenezcan al n % de los mejores.

La diferencia en las dos tablas radica en el criterio de parada, para la primera, se tomó la decisión de no seguir buscando si no se modificó el óptimo luego de 5 iteraciones, para la segunda, si no se lo modificó luego de 10.

	Ejes Fijos	Nodos Fijos	Tableros
Vecindad 2x1 3 mejores	0.1444444444	0.0443696313	0
Vecindad 2x1 5 mejores	0.0873015873	0.0878199237	0.7916666667
Vecindad 2x1 7 mejores	0.1272510823	0.0897730705	0.8083333333
Vecindad 2x1 10 mejores	0.0977272727	0.1190627034	0.75
Vecindad 2x1 12 mejores	0.1186147186	0.1041559051	0.4583333333
Vecindad 3x1 3 mejores	0.278466811	0.151036013	0.0833333333
Vecindad 3x1 5 mejores	0.1813888889	0.2439944838	0.5416666667
Vecindad 3x1 7 mejores	0.2068867244	0.2352629853	0.8083333333
Vecindad 3x1 10 mejores	0.2764466089	0.2326181999	0.8333333333
Vecindad 3x1 12 mejores	0.258968254	0.2521559379	0.1666666667
Vecindad 2x1 10 %	0.086468254	0.0821545316	0
Vecindad 2x1 25 %	0.1322474747	0.0895125969	0.4583333333
Vecindad 2x1 50 %	0.1164862915	0.0996809898	0.7416666667
Vecindad 2x1 75 %	0.1488816739	0.1212623738	1.075
Vecindad 2x1 100 %	0.0852272727	0.0990432947	1.0333333333
Vecindad 3x1 10 %	0.2001839827	0.1837163913	0
Vecindad 3x1 25 %	0.1706132756	0.1460248135	0.1666666667
Vecindad 3x1 50 %	0.2624531025	0.203697747	0.4833333333
Vecindad 3x1 75 %	0.2133477633	0.2416739045	0.9416666667
Vecindad 3x1 100 %	0.2849206349	0.2945727019	0.4833333333

Cuadro 1: Promedio de porcentajes de error de GRASP con respecto al algoritmo exacto para cada vecindad y cada selección de solución inicial. Con criterio de parada fijado en 5 repeticiones sin mejorar la mejor solución hallada

	Ejes Fijos	Nodos Fijos	Tableros
Vecindad 2x1 3 mejores	0.1823232323	0.0716093318	0.3333333333
Vecindad 2x1 5 mejores	0.1201370851	0.0936978155	0.4166666667
Vecindad 2x1 7 mejores	0.1566738817	0.0903101931	0.3333333333
Vecindad 2x1 10 mejores	0.1187950938	0.124087402	0.6166666667
Vecindad 2x1 12 mejores	0.0911976912	0.1113699446	0.6166666667
Vecindad 3x1 3 mejores	0.2596572872	0.1778930085	0.55
Vecindad 3x1 5 mejores	0.2637698413	0.2698233628	0.6333333333
Vecindad 3x1 7 mejores	0.3213239538	0.2286797787	0.55
Vecindad 3x1 10 mejores	0.2844047619	0.2735979587	0.4833333333
Vecindad 3x1 12 mejores	0.2242460317	0.269319628	0.2833333333
Vecindad 2x1 10 %	0.0830808081	0.0538074353	0
Vecindad 2x1 25 %	0.1161616162	0.0841469551	0
Vecindad 2x1 50 %	0.218989899	0.1268513858	0.325
Vecindad 2x1 75 %	0.2454076479	0.126309841	0.3333333333
Vecindad 2x1 100 %	0.2024242424	0.1329405378	0.8333333333
Vecindad 3x1 10 %	0.1853138528	0.1630435123	0
Vecindad 3x1 25 %	0.2839177489	0.2205421872	0.1666666667
Vecindad 3x1 50 %	0.2687914863	0.212152652	0.45
Vecindad 3x1 75 %	0.3088888889	0.1862762984	0.5916666667
Vecindad 3x1 100 %	0.3076803752	0.2703141941	0.9666666667

Cuadro 2: Promedio de porcentajes de error de GRASP con respecto al algoritmo exacto para cada vecindad y cada selección de solución inicial. Con criterio de parada fijado en 10 repeticiones sin mejorar la mejor solución hallada

Para seleccionar la mejor combinación de parámetros de la heurística, respecto a la solución encontrada contra la óptima del algoritmo exacto, tomamos como criterio que se minimice la suma de cada fila, es decir, que para casos donde los grafos incrementan su cantidad de ejes, o bien solo su cantidad de nodos, y tableros de caballos, la diferencia contra la óptima sea mínima.

Esto nos lleva a que la combinación óptima es: Vecindad 2x1 10 % mejores con 10 iteraciones consecutivas sin ver modificaciones en el óptimo hallado hasta ese momento. Con un error promedio del 4,5 %.

Además nos interesa ver que sucede con los tiempos de ejecución, para ver si existe alguna configuración que convenga aplicar por esta métrica.

Para esto, se tomaron tiempos de los mismas instancias para las que analizamos el porcentaje de acierto del óptimo.

En primer lugar analizamos grafos completos y vacíos, esperamos que sus gráficos sean parábolas cuadráticas, ya que tienen una única solución posible en cuánto a la cantidad de nodos de la misma. Recordando que GRASP ejecuta una adaptación del algortimo Greedy de costo $O(n^2)$ y sabiendo que el algoritmo de búsqueda local retorna en tiempo constante si la solución hallada es o bien un nodo (grafo completo), o bien todos los nodos del grafo (grafo vacío, todas componentes conexas triviales), podemos argumentar por qué esperamos encontrar esa curva particular.

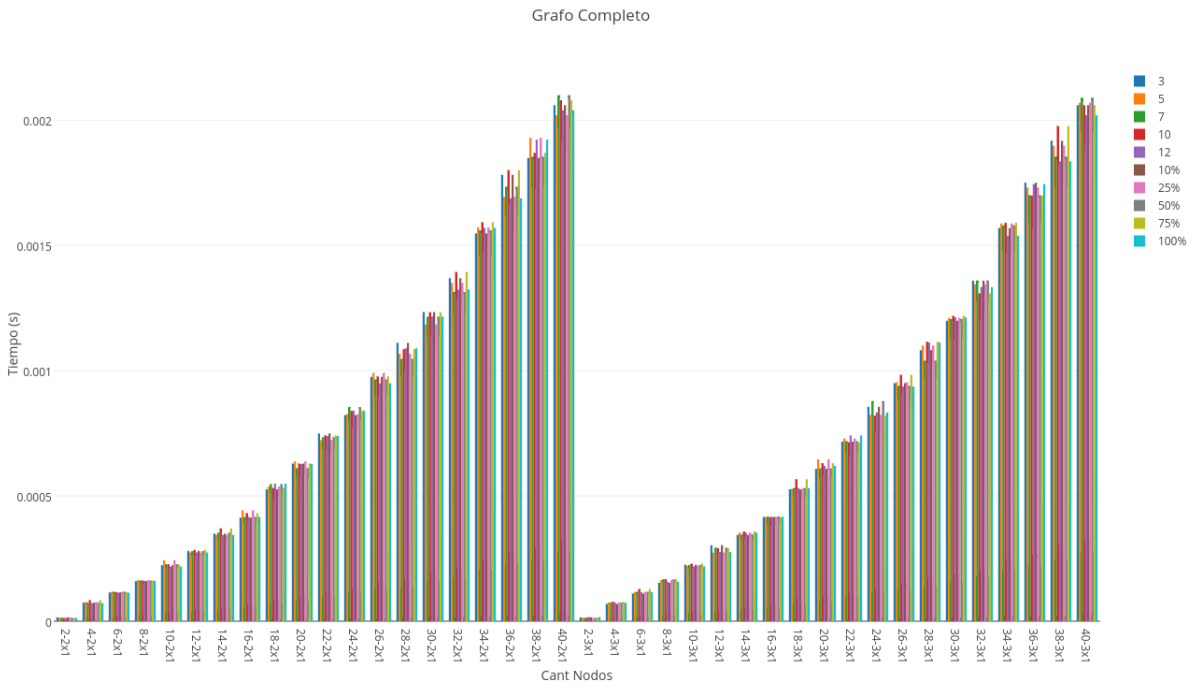


Figura 1: Comparación tiempos de ejecución de la heurística GRASP para grafos completos, aumentando su cantidad de nodos y contrastando las dos vecindades planteadas. Criterio de parada = 5 iteraciones

Figura 2: Comparación tiempos de ejecución de la heurística GRASP para grafos completos, aumentando su cantidad de nodos y contrastando las dos vecindades planteadas. Criterio de parada = 10 iteraciones

Estos dos gráficos verifican la curvatura que esperábamos para el caso del grafo completo, como también que no importa que vecindad se ejecute, pues nunca se intenta mejorar la solución inicial dado que sabemos que es óptima por sus características. Además se aprecia que el alpha elegido no genera oscilaciones significativas al ejecutarse bajo la misma instancia.

El anterior permite vislumbrar que efectivamente los criterios de parada afectan al tiempo de ejecución, se puede ver que las columnas pares duplican a la de su izquierda. Esto se da porque las columnas impares son mediciones sobre no mejorar el óptimo durante 5 iteraciones y las demás durante 10.

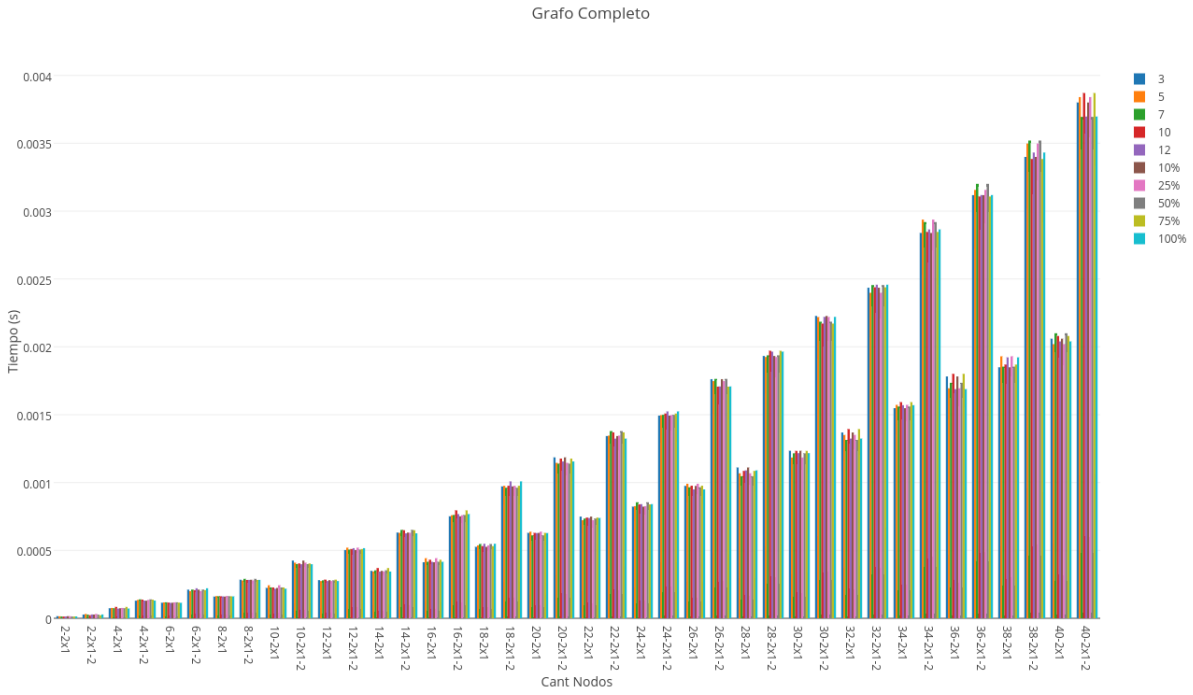


Figura 3: Comparación tiempos de ejecución contrastando los criterios de parada utilizados

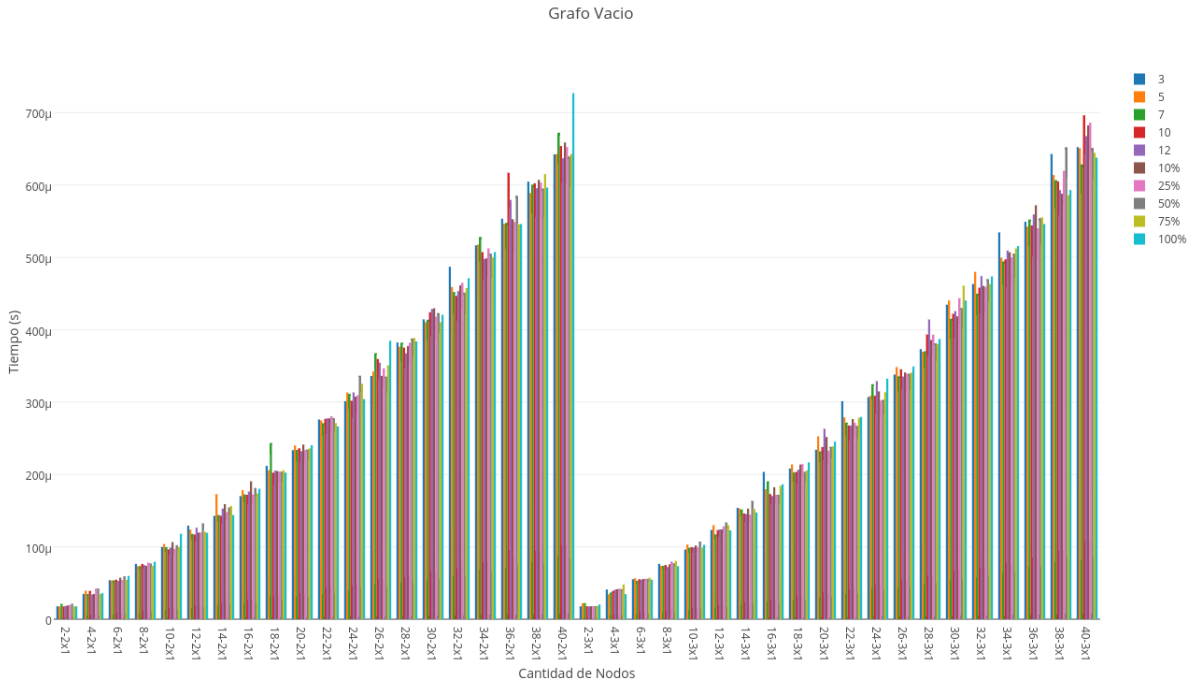


Figura 4: Comparación tiempos de ejecución de la heurística GRASP para grafos vacios, aumentando su cantidad de nodos y contrastando las dos vecindades planteadas. Criterio de parada = 5 iteraciones

Estos 3 gráficos de grafos vacíos se pueden analizar del mismo modo que a los de grafos completos y verificar que cumplen con las mismas caracteríticas, pues son mínimas en su conjunto de vecindades asociados.

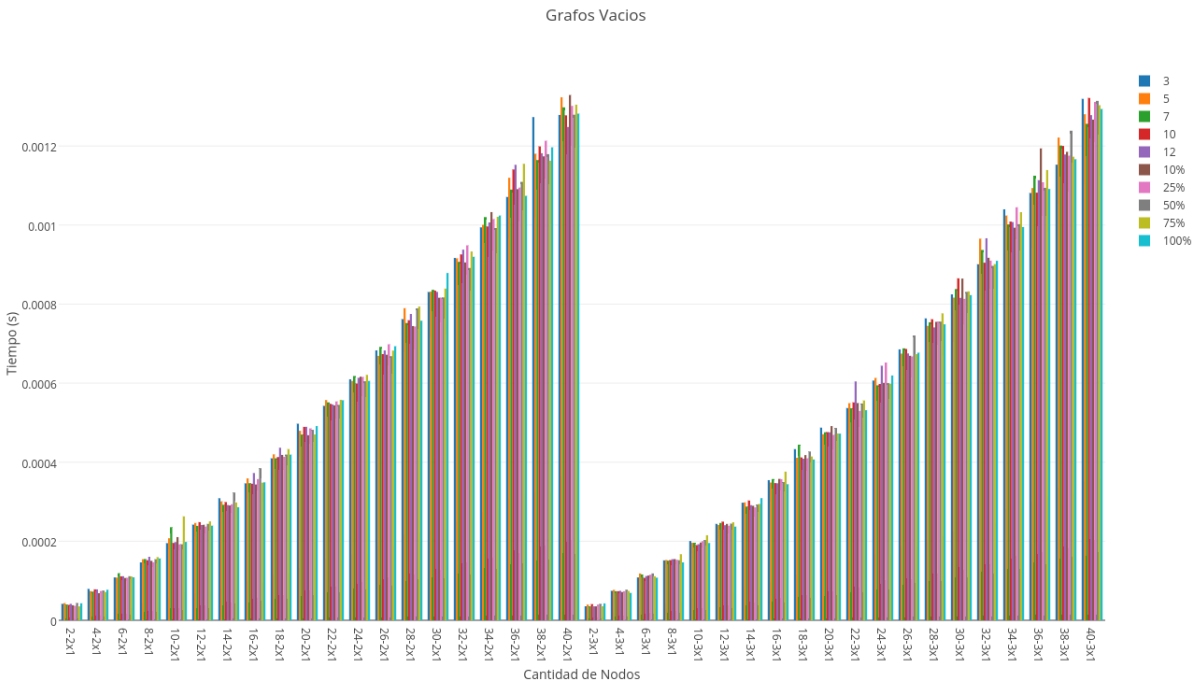


Figura 5: Comparación tiempos de ejecución de la heurística GRASP para grafos vacios, aumentando su cantidad de nodos y contrastando las dos vecindades planteadas. Criterio de parada = 10 iteraciones

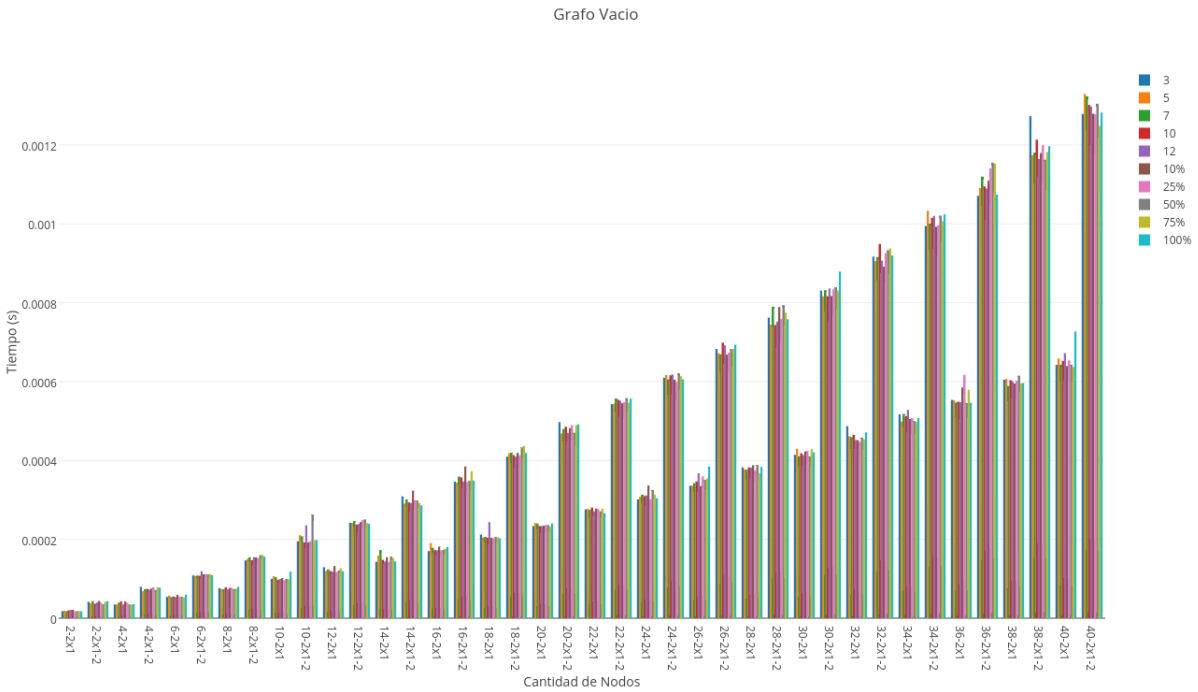


Figura 6: Comparación tiempos de ejecución contrastando los criterios de parada utilizados

Este gráfico y el siguiente muestran lo analizado en las secciones de las heurísticas anteriores, los tiempos de ejecución dependen de la cantidad de nodos que posea el grafo.

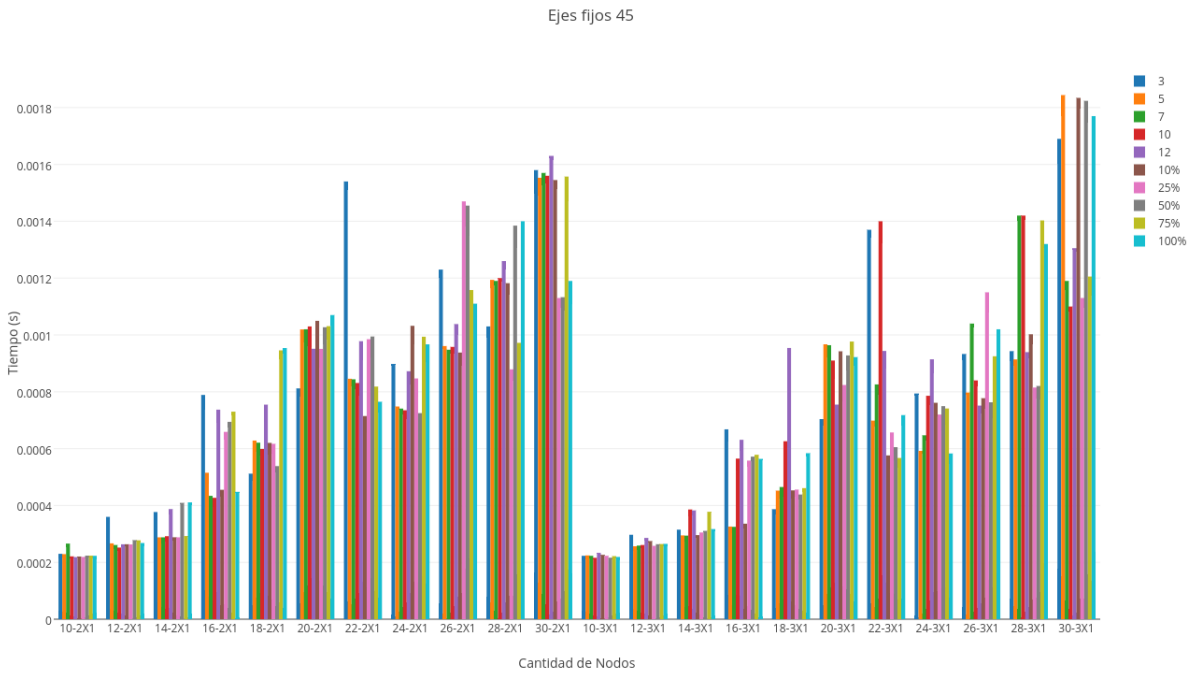


Figura 7: Comparación tiempos de ejecución de la heurística GRASP para grafos aleatorios con 45 ejes, aumentando su cantidad de nodos y contrastando las dos vecindades planteadas. Criterio de parada = 5 iteraciones

Mantener la cantidad de ejes fija y modificar la cantidad de nodos implica que no tendremos una solución trivial como en los casos anteriores, entonces el algoritmo comienza a revisar la vecindad de la solución inicial. Cómo el método para generarla es randomnizado, a priori no sabemos cuántas veces actualizará su óptimo, y esto explica los altos y bajos para algunas ejecuciones de la misma instancia para distintos alphas y vecindades; no obstante se observa que se mantiene una marcada relación instancia-tiempo de ejecución más allá de la vecindad seleccionada.

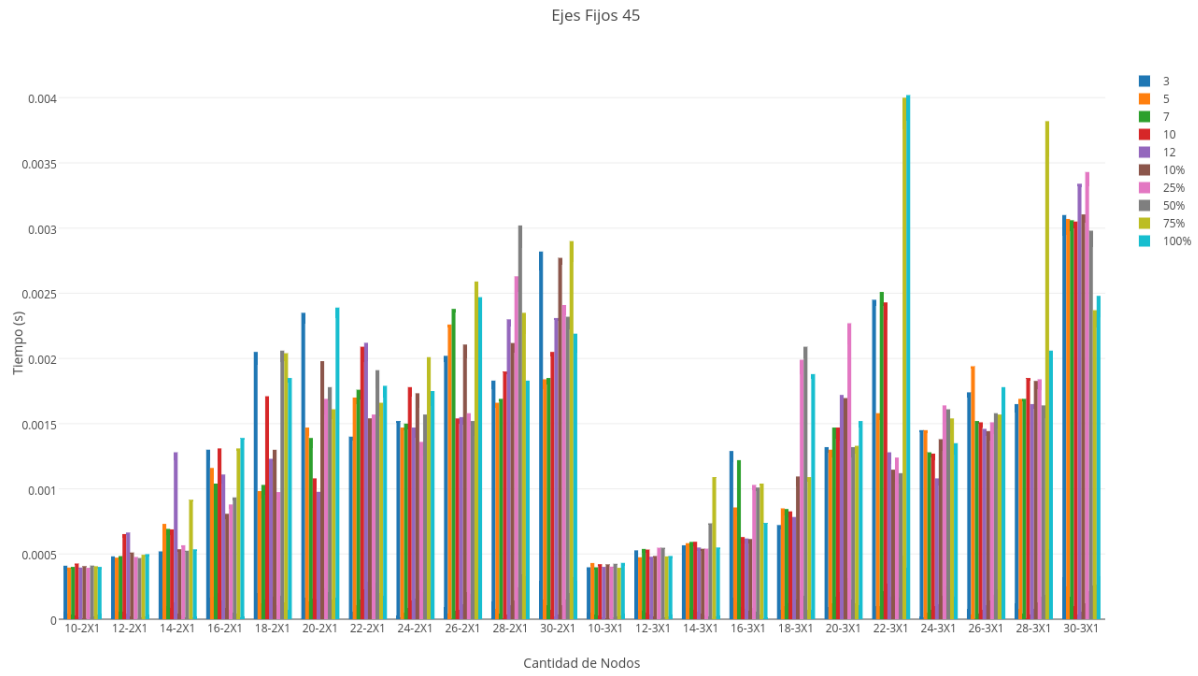


Figura 8: Comparación tiempos de ejecución de la heurística GRASP para grafos aleatorios con 45 ejes, aumentando su cantidad de nodos y contrastando las dos vecindades planteadas. Criterio de parada = 10 iteraciones

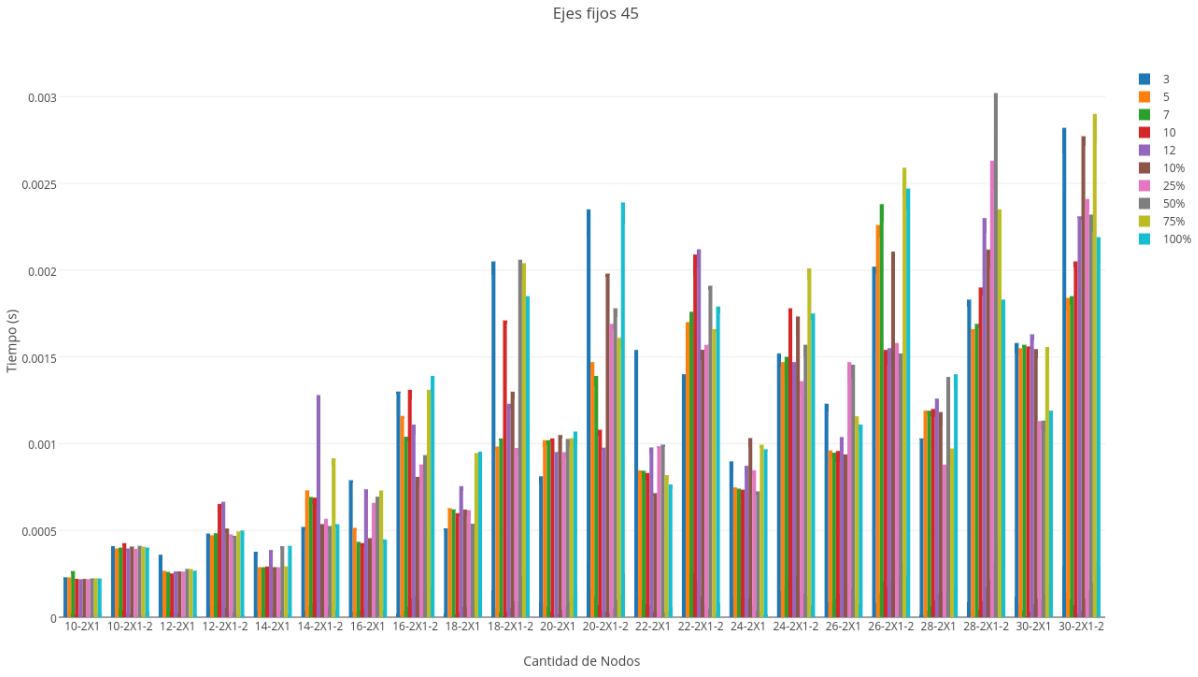


Figura 9: Comparación tiempos de ejecución contrastando los criterios de parada utilizados

Nuevamente, como en los casos Completo y Vacío, la tendencia de que parar luego de 10 iteraciones duplique a parar a las 5, se mantiene vigente.

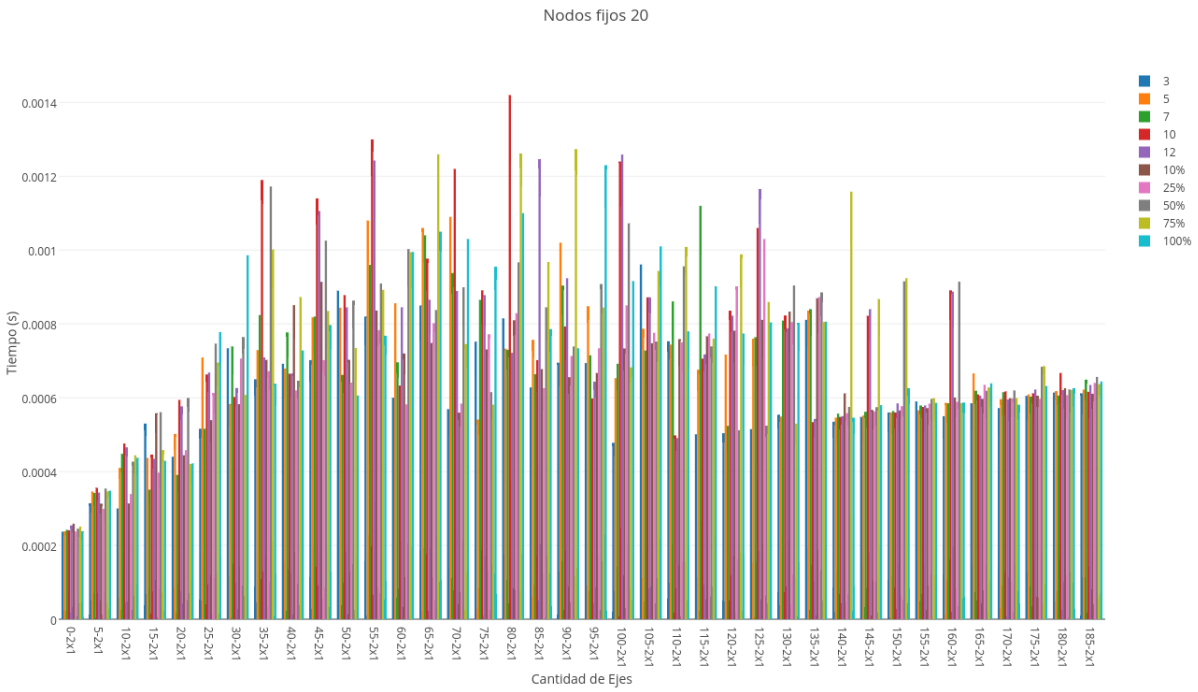


Figura 10: Comparación tiempos de ejecución de la heurística GRASP para grafos aleatorios con 20 nodos, aumentando su cantidad de ejes. Criterio de parada = 5 iteraciones. Vecindad 2x1

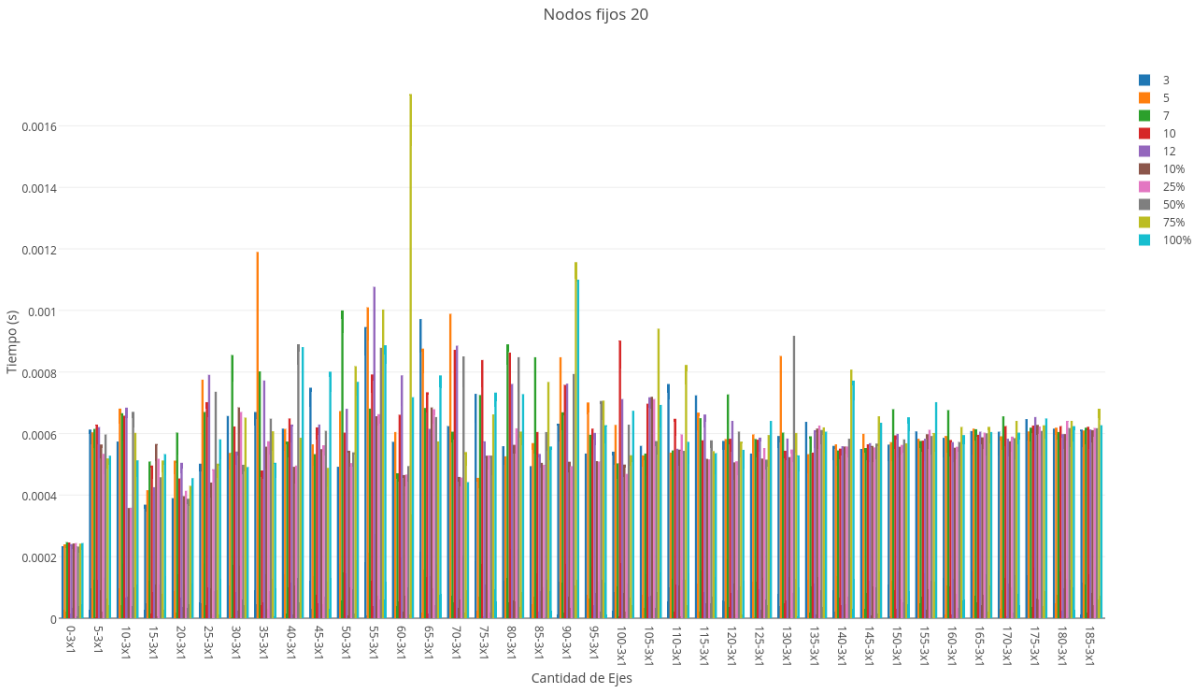


Figura 11: Comparación tiempos de ejecución de la heurística GRASP para grafos aleatorios con 20 nodos, aumentando su cantidad de ejes. Criterio de parada = 5 iteraciones. Vecindad 3x1

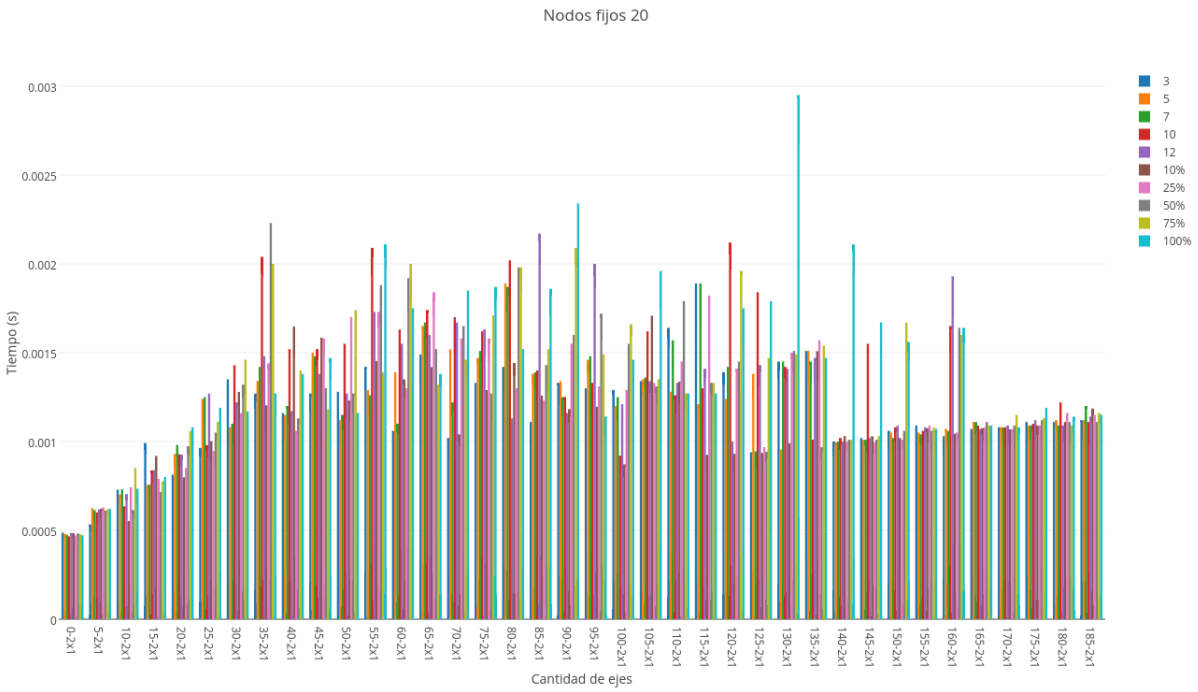


Figura 12: Comparación tiempos de ejecución de la heurística GRASP para grafos aleatorios con 20 nodos, aumentando su cantidad de ejes. Criterio de parada = 10 iteraciones. Vecindad 2x1

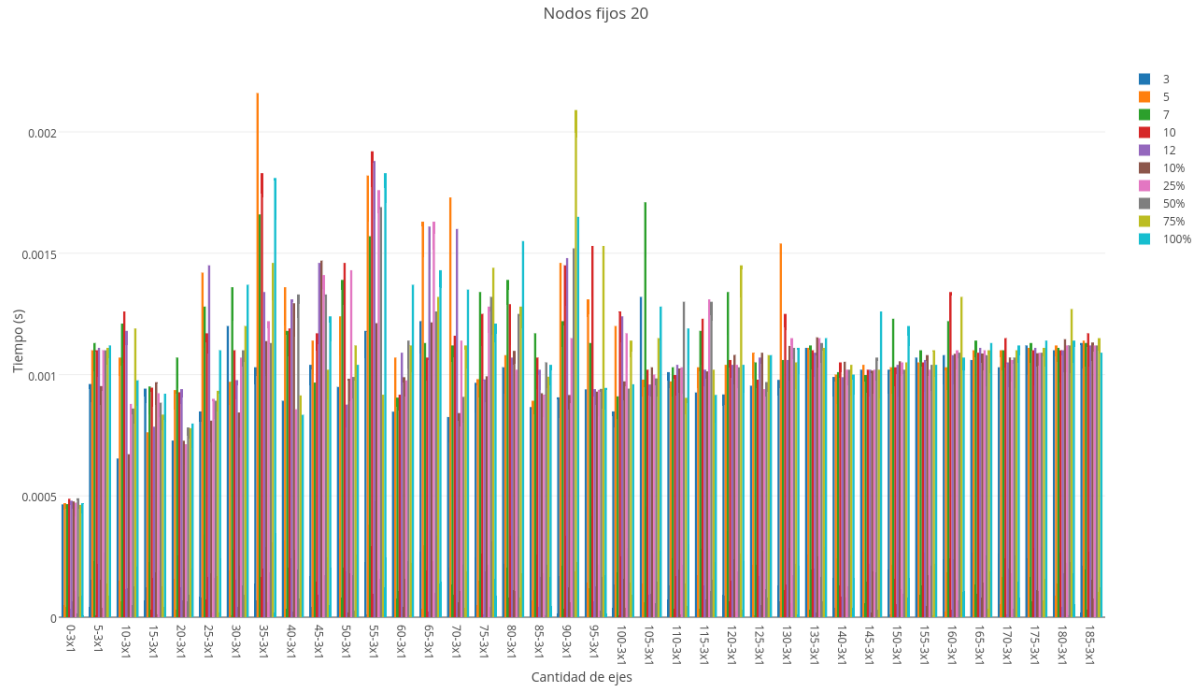


Figura 13: Comparación tiempos de ejecución de la heurística GRASP para grafos aleatorios con 20 nodos, aumentando su cantidad de ejes. Criterio de parada = 10 iteraciones. Vecindad 3x1

Estos últimos cuatro gráficos tiene la característica de tener un tiempo de ejecución aproximadamente igual. No obstante en la zona media hay oscilaciones en función del alpha elegido que no se presentan en los extremos izquierdo y derecho.

Esto se debe a que el primer caso es el grafo vacío (solución trivial) y los siguentes poseen muchos nodos aunque no todos, haciendo que la vecindad tarde más en recorrerse completamente.

Comparando los tiempos hallados, viendo que las fluctuaciones, entre el alpha elegido y la vecindad tomada no presentan aumentos descomunales en los tiempos de ejecución, concluímos que salvo para el caso de alpha igual a 100 %, las ejecuciones son de tiempos semejantes.

Por lo tanto decidimos que la mejor configuración para ejecutar la heuerística GRASP es la que mejor resultados provee en cuánto a qué tan cerca del óptimo está. Que como se mencionó anteriormente es con alpha = 10 %, y la vecindad 2x1.

6. Comparación entre todos los métodos

Una vez elegidos los mejores valores de configuracion para cada heurıstica implementada (si fue posible), realizar una experimentacion sobre un conjunto nuevo de instancias para observar la performance de los metodos comparando nuevamente la calidad de las soluciones obtenidas y los tiempos de ejecucion en funcion de los parametros de la entrada. Para los casos que sea posible, comparar tambien los resultados del algoritmo exacto implementado. Presentar todos los resultados obtenidos mediante graficos adecuados y discutir al respecto de los mismos.

7. Anexo

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 pocentaje de mejores (conAlpha = true), vecindad 2x1 (vecindad = true), alpha = input);
- **7**: *Heurística GRASP* (solución inicial por pocentaje 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;