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

Resumen va aquí 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 anexo como se hace?

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

Esta flag se utiliza para poder utilizar funciones de medición de tiempos de ejecución para 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.

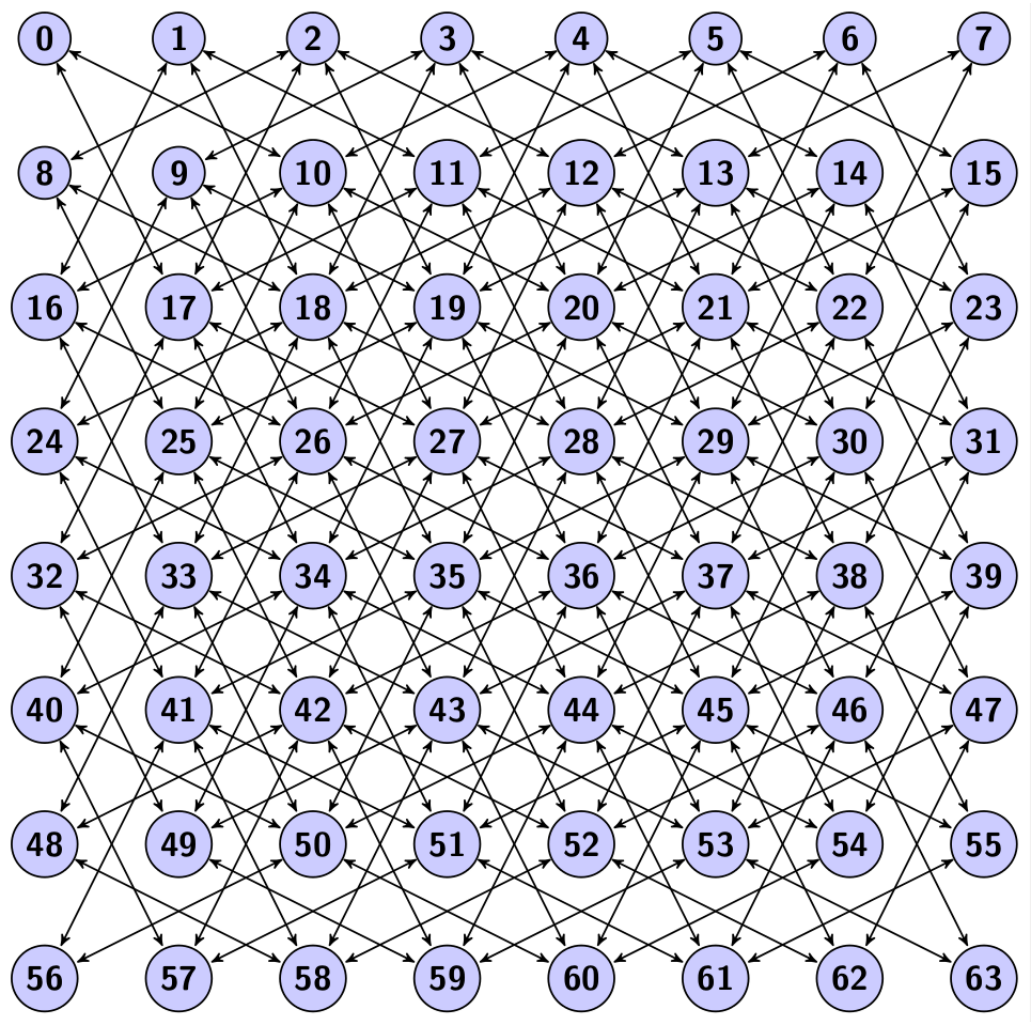
Por otro lado, un conjunto $I \subseteq V$ es un *conjunto independiente* de G si no existe ningún eje de E entre dos vértices de I.

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 al tablero inicial como vacío (?) y 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 la pieza de un 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 caballo.

1.3. Todo conjunto independiente maximal es un conjunto dominante

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. ... completar

Y las estrategias fueron ... completar

Poner el pseudocodigo...

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. demostrar o no hace falta? Para mi es suficiente (AGUS)

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

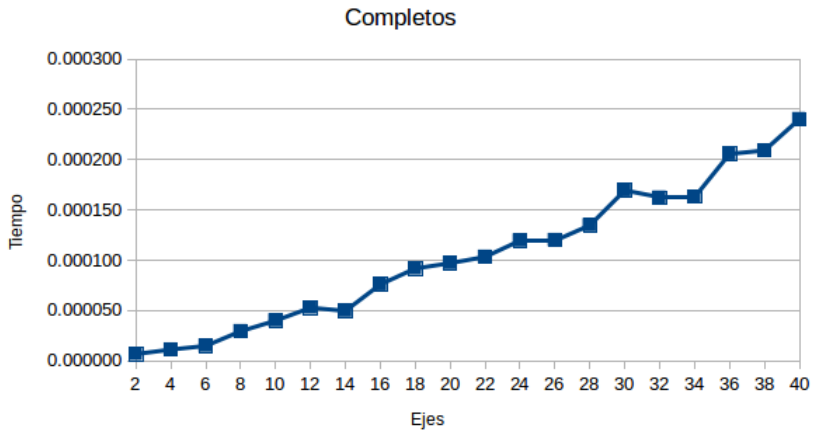
Para realizar la experimentación, se genereron grafos específicos, que nos permitieron ver el funcionamiento de nuestro algoritmo. Las conexiones entre los nodos de cada grafo son aleatorias, pero respetando la cantidad de ejes que nosotros definamos previamente.

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 debe ser necesariamente una solución óptima.

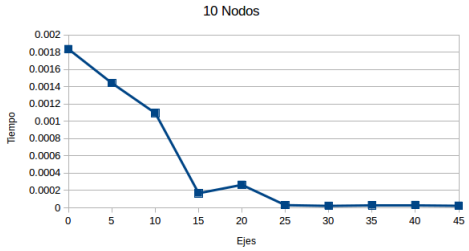
Tabla

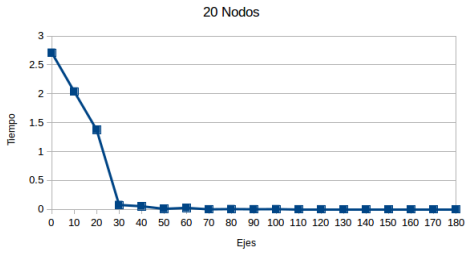


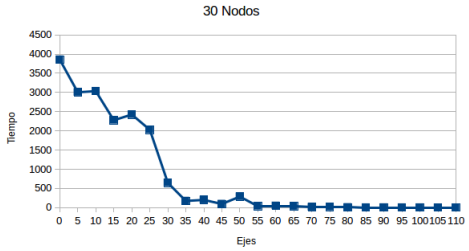
Se puede ver que el gráfico tiende a una recta, lo que era esperable ya que encontrará la solución en el primer nodo, pero luego deberá descartar los siguientes n nodos, teniendo una complejidad de $O(n)$.

2.3.2. Nodos fijos

Para continuar viendo el comportamiento del algoritmo, decidimos realizar experimentos fijándo 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 y tienden a ser más lineales, ya que gracias a la poda, no es necesario recorrer todas las posibles soluciones.

3. Heurística Constructiva Golosa

3.1. Explicación

La heurística constructiva golosa busca, dado un grafo, determinar un conjunto independiente dominante mínimo.

Para ello, el algoritmo ordena los nodos de acuerdo al grado de cada uno de ellos, de mayor a menor. Este orden se establece al comienzo de la ejecución y no se actualiza.

Luego, recorre dichos nodos, y por cada uno de ellos, lo agrega al conjunto solución, y va eliminando a sus vecinos, hasta que no quedan más nodos.

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.

Falta una explicacion mas detallada (un pseudocodigo, o algo), y aclarar que es goloso porque en cada paso elige la mejor opcion y nosostros definimos MEJOR como el mayor grado total

3.2. Complejidad Temporal

Aca no me meto, todavia...
En lo que respecta a la complejidad temporal, demostraremos a continuación que la misma es $O(n^2)$.

Recordemos que el algoritmo toma los nodos con mayor grado, lo agrega al conjunto solución, y lo elimina junto a sus adyacentes. Dicho esto, definiremos a $f(a, b) : \mathbb{N} \to \mathbb{N}$ como:
$f(a, b)$ = Cantidad de operaciones en el peor caso, teniendo b nodos y faltando eliminar a.
Es decir,
$f(a, b) = h * a + f(a - h, b)$, con h la cantidad de vecinos del nodo que estoy viendo, y $f(i - h, b)$ es el llamado recursivo, habiendo tachado dichos h vecinos. Esto esta bien definido pues en el peor de los casos, debo recorrer la l
Entonces, querremos demostrar que $f(a, b) = a * b + k$ (con k alguna constante), pues al momento de ejecutar el algoritmo, se tienen n nodos, y faltan eliminar n nodos, en cuyo caso, la complejidad del mismo será $O(f(n, n)) = O(n^2)$.

Teorema
Dado $f(a, b) : \mathbb{N} \to \mathbb{N}$.
$f(a, b)$ = Cantidad de operaciones teniendo b nodos y faltando eliminar a.
Luego, $f(a, b) = a * b + k$.

Demostración Realizaremos la demostración por inducción en la cantidad de nodos que falta eliminar. Luego,
Casos base:

- $f(0, b) = k$

  Dado que tengo b nodos, y ya no me quedan más por eliminar, el algoritmo terminó. k es alguna cantidad constante de operaciones.

- $f(1, b) = k$

  Dado que tengo b nodos, y solo falta eliminar uno k es alguna cantidad constante de operaciones para eliminar el nodo y finalizar el algoritmo.

- $f(2, b) = h + k$

  Dado que tengo b nodos, y me falta eliminar 2. Tomo uno, y recorro su lista de adyacencia, que en este casos tendra h elementos.

Paso inductivo:
Supongamos que $f(r, b) = r * b$, con $r \leq a - 1$, queremos ver que $f(i, b) = i * b$.
$f(i, b) = h * b + f(i - h, b)$, donde h es la cantidad de adyacentes al nodo que estoy viendo, y $f(i - h, b)$ es la llamada recursiva luego de haber eliminado los h vecinos.
$\Rightarrow f(i, b) = h * b + f(i - h, n) \overset{HI}{=} h * b + (i - h) * b = (h + i - h) * b = i * b \square$

3.3. Comparación de resultados con solución óptima

3.4. Experimentación

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 I 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.

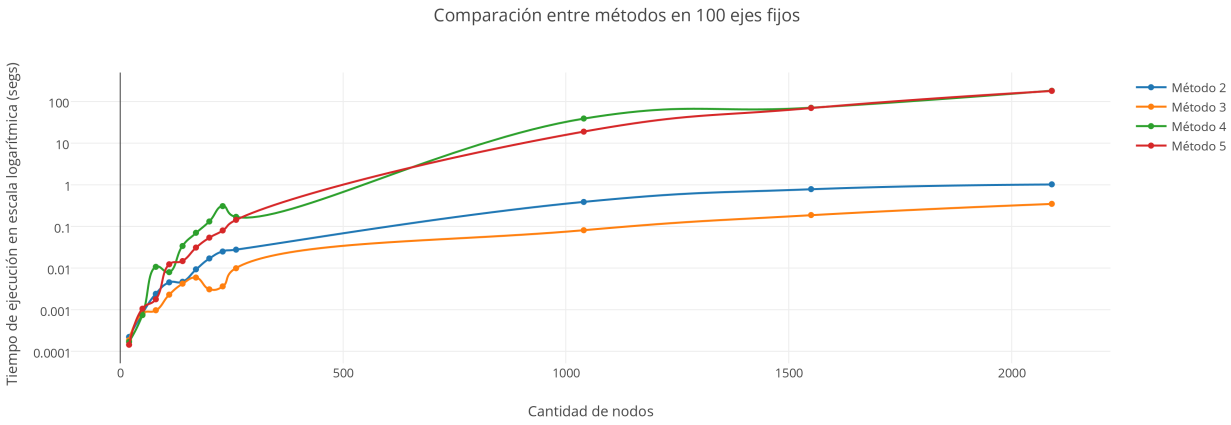
Las opciones que elegimos son: 2, 3, 4 y 5 y son BLABALLABLA

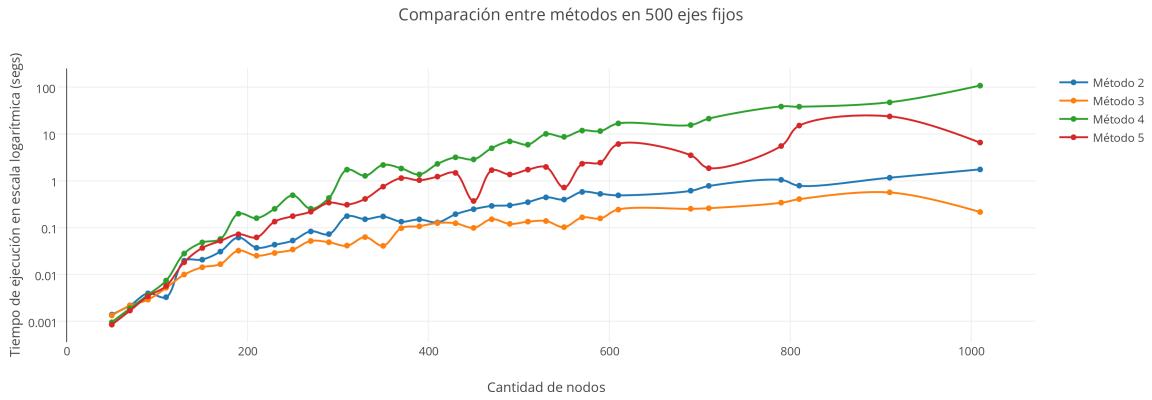
4.2. Complejidad Temporal

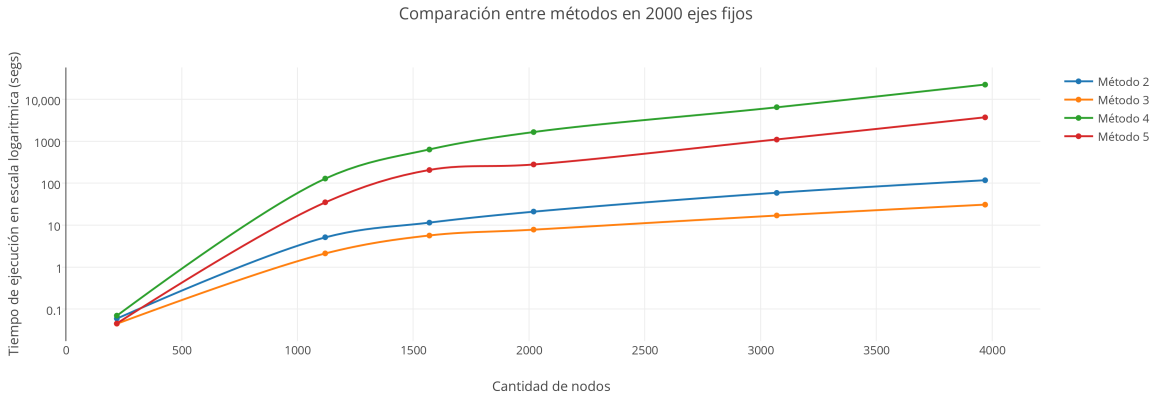
```
/**
V,E
2E = E(0) + E(1) + ... + E(V-1)
(0,1) => E(0) + E(1)
(0,2) => E(0) + E(2)
...
(0,V-1) => E(0) + E(V-1)
(1,2) => E(1) + E(2)
(1,3) => E(1) + E(3)
...
(1,V-1) => E(1) + E(V-1)
E(0) * (V-1) + E(1) * (V-1) + ... + E(V-1) * (V-1) = 2E * (V-1) = o(EV) = o(V^3)


**/
```

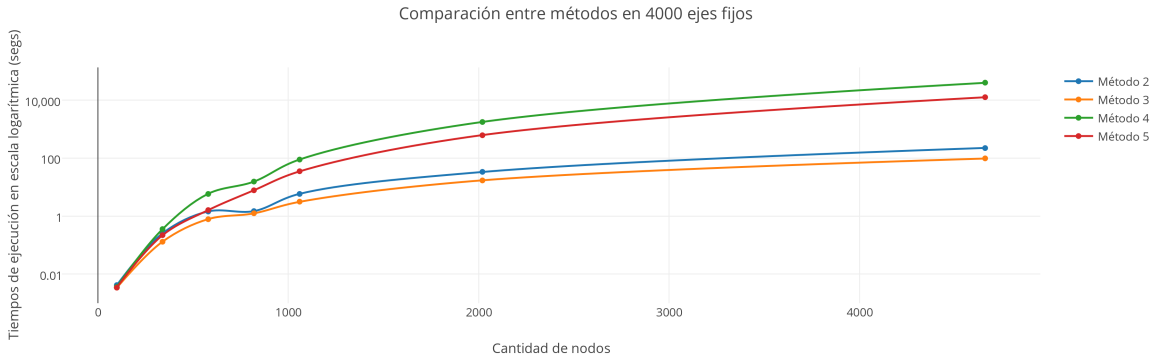
4.3. Experimentación

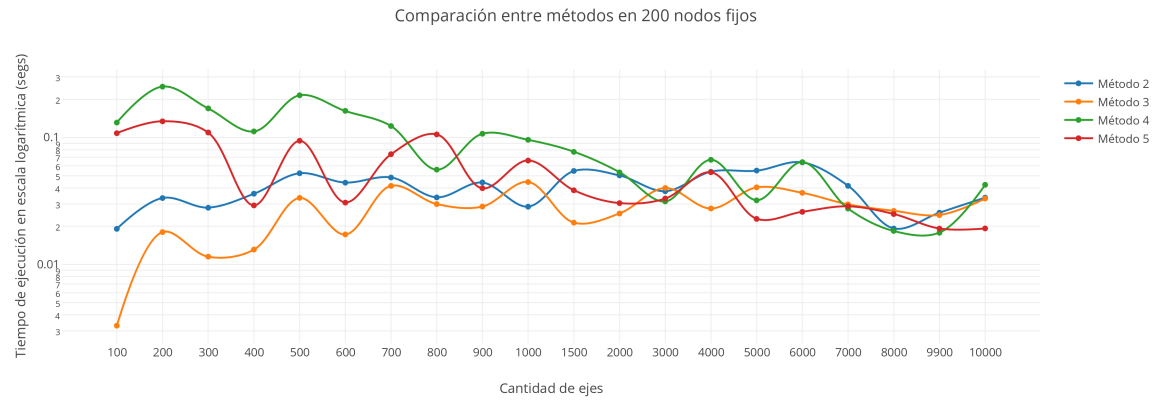
4.3.1. Análisis de tiempos de ejecución

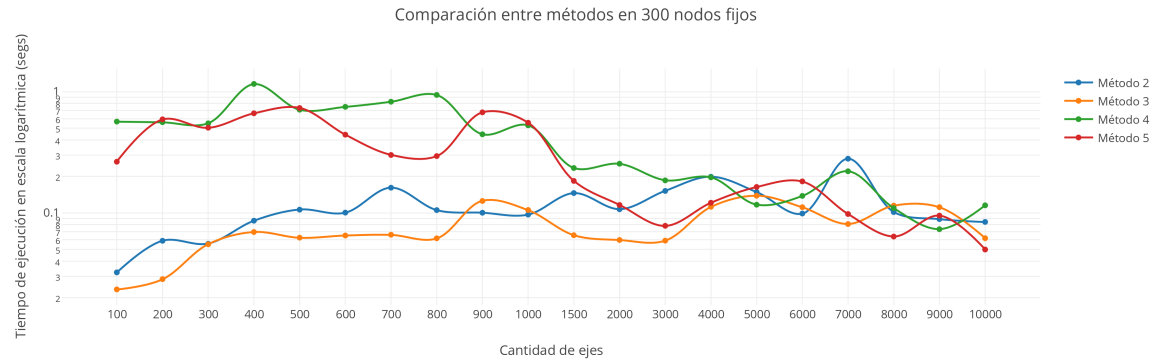


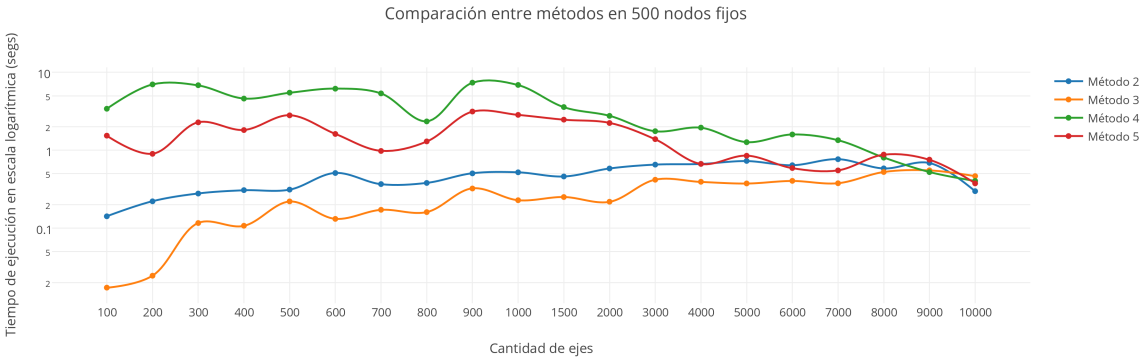


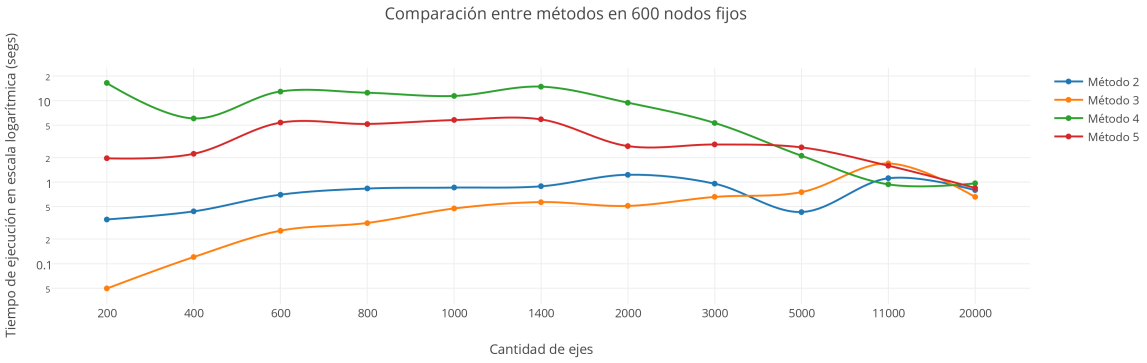


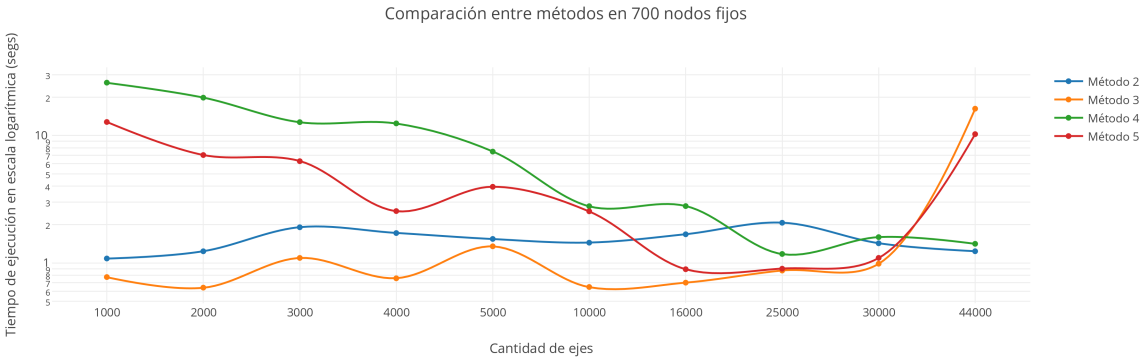

Comparación entre métodos en 4000 ejes fijos
Tiempos de ejecución en escala logarítmica (segs)
10,000
100
1
0.01
0
1000
2000
3000
4000
Cantidad de nodos
Método 2
Método 3
Método 4
Método 5


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

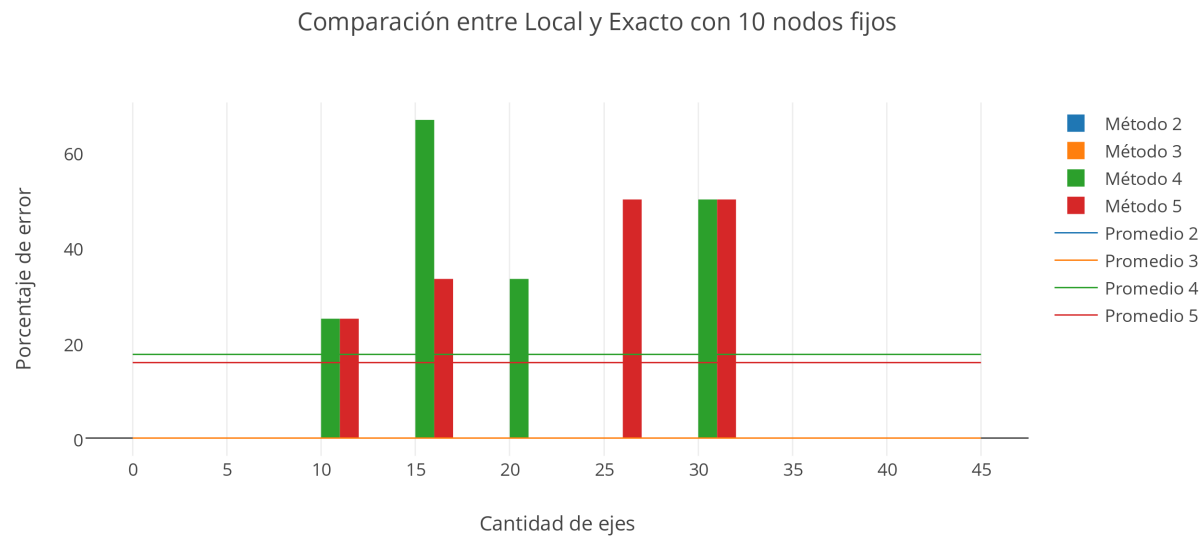


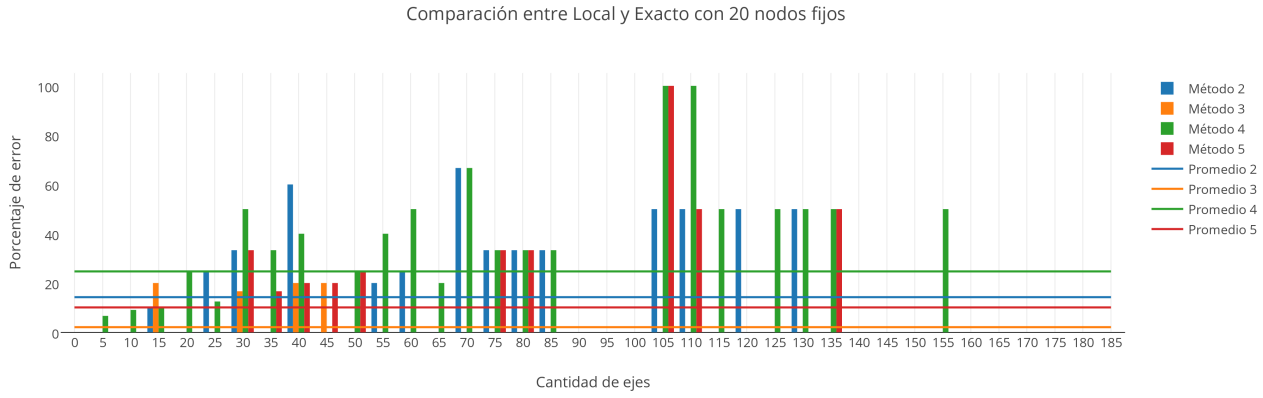


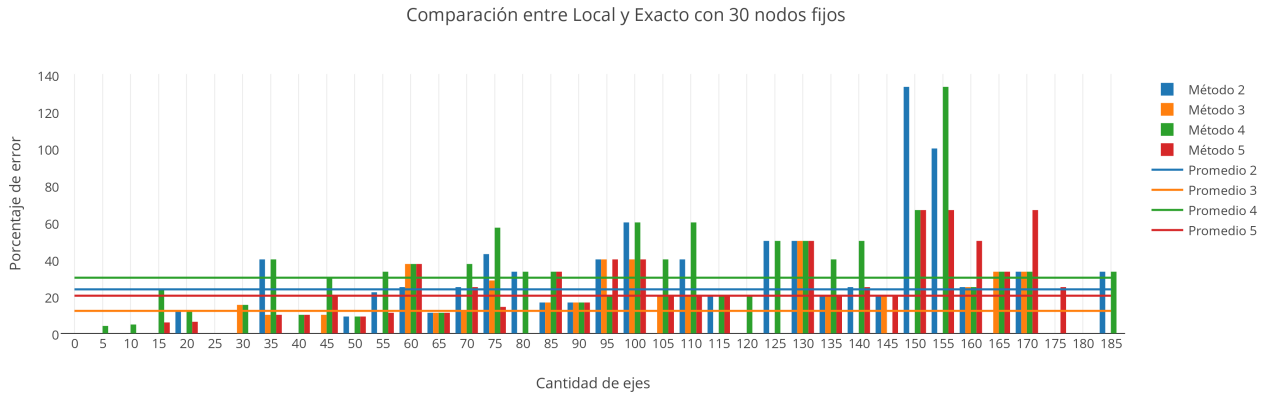


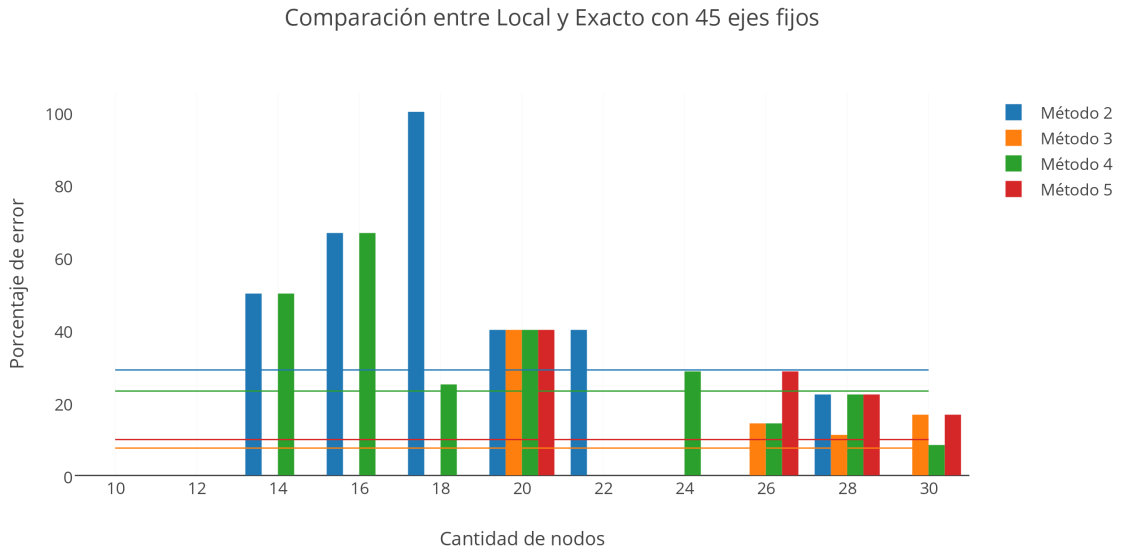
4.3.2. Contrastación empírica de la complejidad

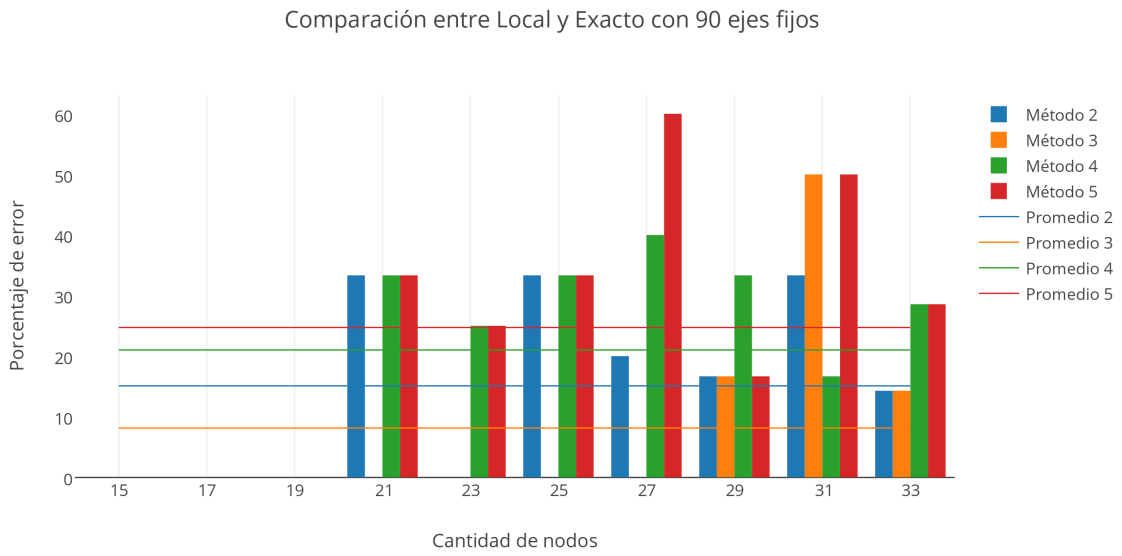
4.3.3. Comparación soluciones Local vs Exacto

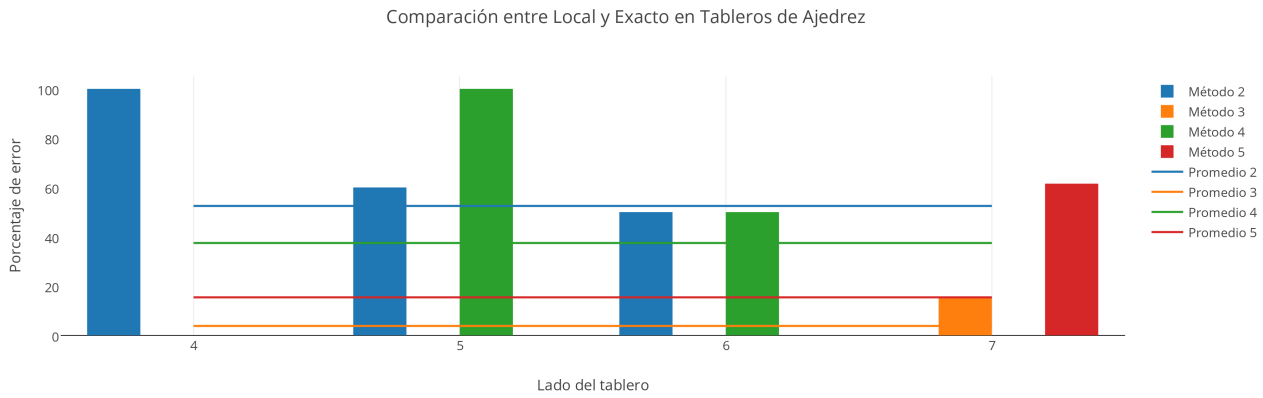







Comparación entre Local y Exacto con 45 ejes fijos
Porcentaje de error
Cantidad de nodos
Método 2
Método 3
Método 4
Método 5
0
20
40
60
80
100
10
12
14
16
18
20
22
24
26
28
30


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 la solución inicial 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.
Poner como correr 6 7 8 9 y que correnc on q parametros

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*.

La siguiente tabla muestra los valores obtenidos, las columnas 2, 3 y 4 indican el criterio aplicado al grafo.

Sory, la tabla no la puedo corregir, no la entiendo bien
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 %.

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

Para correr las heurísticas:

- 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;