# Algoritmos y Estructuras de Datos III

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

## Trabajo Práctico 3



| Integrante | LU | Correo electrónico |
|---|---|---|
| Ortiz de Zarate, Juan Manuel | 403/10 | jmanuoz@gmail.com |
| Martelletti, Pablo | 849/11 | pmartelletti@gmail.com |
| Kujawski, Kevin | 459/10 | kevinkuja@gmail.com |
| Carreiro, Martin | 45/10 | carreiromartin@gmail.com |



**Reservado para la cátedra**

| Instancia | Docente | Nota |
|---|---|---|
| Primera entrega | | |
| Segunda entrega | | |

# Índice

**1. Casos de la Vida Real** **3**
1.1. Situación 1 . . . 3
1.2. Situación 2 . . . 3

**2. Algoritmo Exacto** **4**
2.1. Enunciado . . . 4
2.2. Solución . . . 4
2.3. Pseudocódigo . . . 4
2.4. Análisis de Complejidad . . . 5
2.5. Peor Caso . . . 5
2.6. Tests y análisis . . . 5

**3. Algoritmo Goloso** **7**
3.1. Solución . . . 7
3.2. Pseudocódigo . . . 7
3.3. Análisis de Complejidad . . . 7
3.4. Mejor Caso . . . 8
3.5. Peor Caso . . . 8
3.5.1. Familias . . . 8

**4. Algoritmo Local Search** **11**
4.1. Solución . . . 11
4.2. Pseudocódigo . . . 11
4.3. Análisis de Complejidad . . . 12
4.4. Peor Caso . . . 13
4.5. Tests y análisis . . . 13

**5. Algoritmo GRASP** **15**
5.1. Solución . . . 15
5.2. Pseudocódigo . . . 15
5.3. Análisis de Complejidad . . . 15
5.4. Tests y análisis . . . 16

**6. Resultados Finales** **20**
6.1. Introducción . . . 20
6.2. Por Cantidad De Nodos . . . 20
6.3. Por Tiempo . . . 20

**7. Conclusiones** **20**

# 1. Casos de la Vida Real

## 1.1. Situación 1

Supongamos que se organiza una fiesta a la que asiste mucha gente (es una fiesta democrática e inclusiva y cada uno puede invitar a todos los amigos que quiera). Como la fiesta es en un parque de diversiones jurásico en el que hay dinosaurios vivos y mutantes sueltos es necesario capacitar a la gente por cualquier accidente o imprevisto que pueda ocurrir. Ahora, resulta que la fiesta tuvo tanta difusión que asistieron alrededor de 70 mil personas. Esto desconcertó profundamente a los organizadores, ya que al ser tanta gente es muy difícil capacitarla. Por suerte el presidente del parque conocía a un grupo de investigadores de exactas (grupo 10) a los que les delegó la tarea, aclarándoles que con la única información que contaban era con una tabla de relaciones (obtenida de facebook) personas a personas, en la que una persona se relaciona con otra si y solo si son amigos. Los investigadores abstrayeron el problema y realizaron el siguiente planteo: -Planteemos el problema como un problema de grafos! -Cada persona es un nodo -Un nodo se relaciona con otro si y solo si esas personas son amigas -Busquemos la menor cantidad de nodos para capacitar (podría pensarse como los más populares), para que luego estos nodos (o personas) capaciten a sus amigos y se logre así capacitar a toda la gente que concurrió a esta zarpada fiesta.

Buenísimo pero ahora ¿Cómo obtenemos la menor cantidad de gente a capacitar?.... Conjunto dominante!, este conjunto dominante representaría a la mínima cantidad de gente a capacitar, tal que luego esas personas al capacitar a sus amigos lograsen capacitar a toda la gente que concurrió a la fiesta.

## 1.2. Situación 2

Supongamos que en un tablero de ajedrez queremos poner algunas reinas de manera que todas las casillas estén amenazadas por al menos una de ellas y queremos usar la menor cantidad posible de damas. ¿Cuántas reinas son necesarias? y ¿cómo hay que ubicarlas? Se puede plantear este ejemplo como un problema de conjunto dominante. Para ello consideramos el grafo donde los vértices representan las casillas de un tablero de ajedrez y dos vértices son adyacentes si una reina ubicada en la casilla u amenaza la casilla v. Un conjunto dominante mínimo D nos dice dónde hay que ubicar las reinas en el tablero para amenazar a todas las casillas.

# 2. Algoritmo Exacto

## 2.1. Enunciado

En este ejercicio se nos solicita buscar el conjunto dominante mínimo y óptimo dado un grafo cualquiera. Este es un problema del tipo NP completo para el cual aún no se encontró forma de resolverlo polinomialmente pero tampoco se demostró que no sea posible solucionarlo con dicha complejidad.

## 2.2. Solución

Como todavía no se halló algoritmo alguno para resolverlo polinomialmente y nosotros no somos investigadores/iluminados (aún) decidimos resolverlo de manera exponencial. Para esto utilizamos el popular método de backtracking (dicho en criollo) de quedarme con la mejor opción entre poner o no un nodo en el conjunto dominante. Este procedimiento lo que hace, básicamente, es analizar todas las soluciones posibles y agarrar la mejor de ellas.
No nos pareció significativo agregarle memorization ya que su complejidad no mejoriría de forma considerable debido a que la complejidad del algoritmo reside en calcular cada solución posible y no en el recalculo de las mismas, además de que ocuparía mas memoria. Tampoco nos pareció imperante preocuparnos por la complejidad de la función que chequea si el conjunto recibido es dominante, ya que, mientras sea polinimal, va a ser despreciable al lado de la complejidad de analizar todas las soluciones (que como dijimos es exponencial).
Finalmente queremos resaltar que cualquier optimización conocida para este algoritmo no haría mas que mejorar la complejidad para ciertos tipos de casos, como el ejercicio no especifica que los grafos a recibir cumplan ciertos parametros o restricciones, todo tipo de perfeccionamiento que le implementemos va a dar lo mismo para las consignas del ejercicio.

## 2.3. Pseudocódigo

global grafoOriginal

obtenerConjuntoDominanteMinimo (**in** $Grafo$) $\longrightarrow$ *conjuntoDom* : Conj

```
1  c = crearConj()
2  grafoOriginal = Grafo
3  return buscarMinimo(Grafo,c)
```

---

buscarMinimo(**in** $Grafo$, **in** *conjuntoDom*) $\longrightarrow$ *conjuntoDom* : Conj

```
1  Si esDominante(conjuntoDom) Hacer:
2      return conjuntoDom
3  Si no
4      vertice = grafo.obtenerVertice();
5      grafo = grafo.sinUno(vertice);
6      conjunto1 = buscarMinimo(grafo, conjuntoDom);
7      if(|conjunto1| == 1) return conjunto1
8      conjunto2 = buscarMinimo(grafo, conjuntoDom + vertice);
9      if(|conjunto2| == 1) return conjunto2
10     return min(conjunto1, conjunto2)
```

---

esDominante(**in** $Grafo$, **in** *conjuntoDom*) $\longrightarrow$ *esDominante* : Boolean

```
1  Grafo grafoDom = crearGrafo()
2  Para cada v en conjuntoDom
3      grafoDom.agregarVertice(v)
4      grafoDom.agregarVertices(v.adyacentes())
5  Para cada v en V(grafoOriginal)
6      Si grafoDom.esta?(v) Hacer:
7          continue
8      Si no
9          return false
10 return true
```

---

## 2.4. Análisis de Complejidad

La complejidad de mi algoritmo es de $2^n$ * $n^2$. Ya que hace $2^n$ recursiones y en cada una de ellas ver si el conjunto formado es dominante cuesta a lo sumo $n^2$. Voy a demostrar por inducción la parte exponencial:

Caso Base:

n = 1, si n tiene un solo nodo ver si es dominante me cuesta O(1) ya que recorrer los vertices del grafo original es una sola iteracion y no es posible recorrer sus aristas. Luego divido en el caso en el que uso a ese nodo en el conjunto dominante y el caso en que no. El caso en que no al no tener mas nodos con cual probar me va a devolver el grafo original y el otro caso también ya que el grafo original era el que contenía únicamente a ese nodo. Ambos casos ver si el conjunto es dominante cuesta O(1) ya que tiene a lo sumo una iteración para hacer. Por lo tanto el algoritmo costaría O(1) que es igual a $O(2^1 * 1^3)$.

Hipótesis inductiva:

Supongo que con n nodos la cantidad de recursiones es $2^n$

Paso inductivo:

Quiero ver que con n+1 nodos la cantidad de recursiones será $2^{n+1}$

Ver si el conjunto vacío es dominante me cuesta O(1) ya que en la primera iteración del grafoOriginal no es posible encontrar ningún vertice en el conjunto dominante o adyacente a él. Luego obtengo un nodo del grafo (grafo en el que me guarde todos los vertices, no el origianl) y busco el minimo conjunto dominante agregando ese nodo al conjunto o no, esto por hipótesis inductiva me cuesta 2 * ($2^n$), ya que tengo que calcular 2 veces el conjunto dominante mínimo para n nodos. Por lo tanto la complejidad me termina costando $2^{n+1}$. Con lo cual queda demostrada la complejidad.

Ahora voy a demostrar que verificar si el conjunto recibido es dominante cuesta O($n^2$):

Esta función cuenta con 2 iteraciones, una sobre el conjunto de nodos dominantes y otra sobre el grafo original. La iteracion sobre el conjunto dominante crea un grafo con todos los nodos del conjunto y sus adyacentes, para esto agrega cada elemento del conjunto y todos sus adyacentes al nuevo grafo (obviamente si se agrega 2 veces el mismo queda 1 vez sola en el grafo) esto nos puede dar un total de n*(n-1) iteraciones, lo que pertenece a O($n^2$).
En la iteracion sobre el grafo original, por cada nodo de este se fija si pertence al grafo recien creado, chequear si el nodo pertence al grafo cuesta O(n) ya que debe compararlo contra todos los nodos del otro grafo y como el grafo original tiene n nodos nos terminar quedando tambien una complejidad de O($n^2$).
Finalmente obtenemos que la complejidad de esta verificacion es de O($2n^2$) que es igual a O($n^2$). Por lo tanto la complejidad total del algoritmo nos termina quedando O($2^n$*$n^2$)

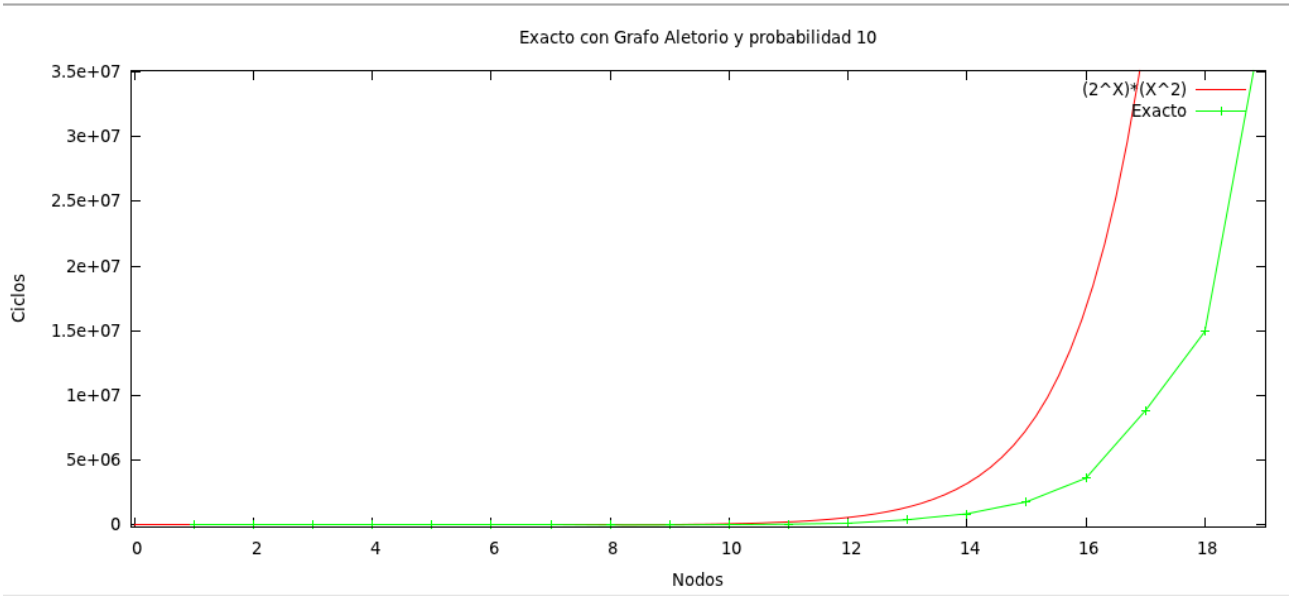
## 2.5. Peor Caso

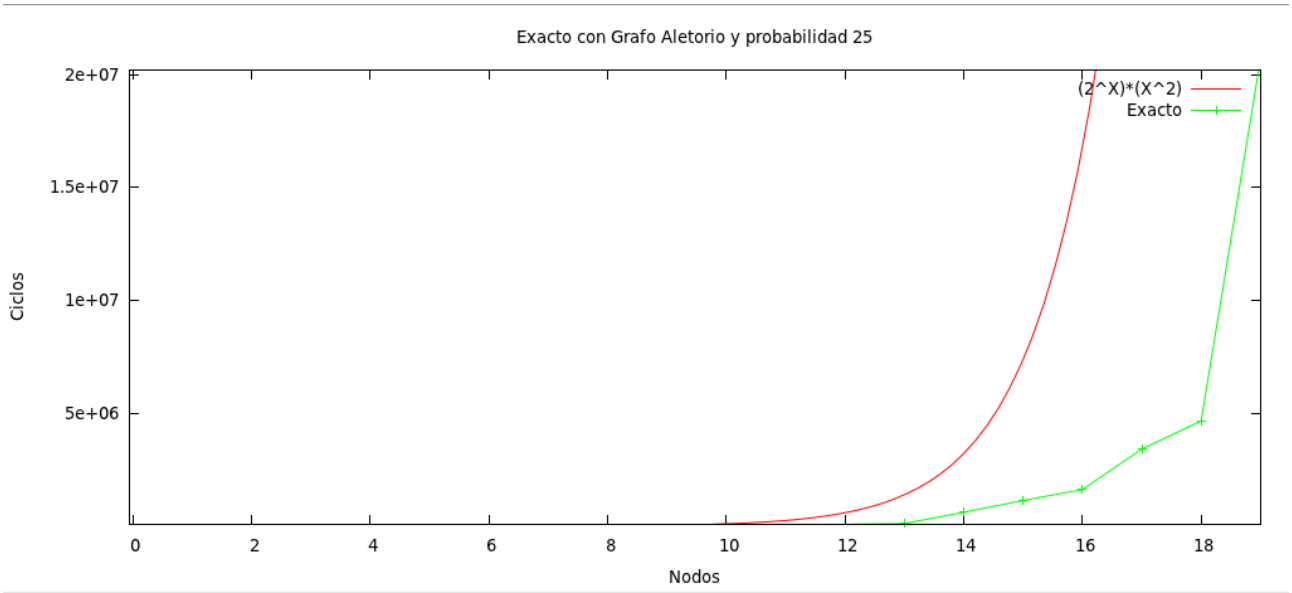
Como el algoritmo es exacto y recorre todas las soluciones posibles siempre obtiene la mejor de ellas. Por lo tanto no existe una 'peor' instancia en la que la solución devuelta sea sub-óptima, siempre devuelve la mejor.

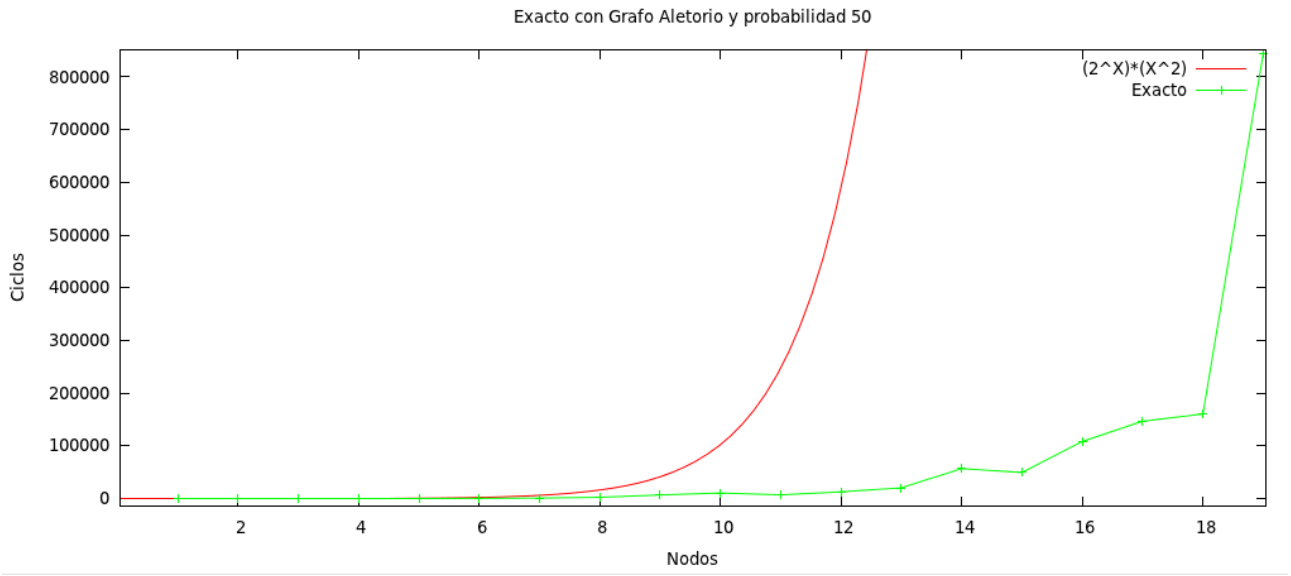
## 2.6. Tests y análisis

Como no podiamos correr el algoritmo para grafos de muchos nodos, no solo por una cuestion temporal sino porque se hace casi imposible de graficarlos por los saltos cada vez mas grandes que se producen, se nos ocurrio correr el algoritmo para grafos aleatorios. Como generamos estos grafos aleatorios? mediante un algoritmo (se encuentra en la clase grafoFactory) que recibe un n (cantidad de nodos) y un entero (probabilidad) que lo que hace basicamente es crear un grafo con n nodos y para cada par de nodos genera un numero random de 0 a 100 y si ese numero es menor a la probabilidad recibida crea la arista entre esos nodos.
Como a nuestro algoritmo le agregamos que si encuentra algun conjunto dominante de un 1 nodo lo devuelva (ya que no puede haber un conjunto dominante de menos de un nodo) al aumentar el parametro probabilidad que recibe nuestro algoritmo aumentamos la probabilidad de que haya algun nodo con grado (n-1) y por lo tanto disminuimos la cantidad de ciclos de nuestro algoritmo exacto, ya que al probar el conjunto dominante con ese nodo lo devolvera automaticamente.
En los siguientes grafos podemos ver que a menor probabilidad mas se acerca la curva a la funcion O($2^n$*$n^2$), debido a que no encuentra un punto de corte el algoritmo y analiza todas las posibilidades, en camibo a medida que se aumenta la probabilidad la curva no solo se aleja de la funcion sino que empieza a perder su forma exponencial y aparecen casos extranos en los que un grafo de x nodos requiere menor o la misma cantidad de ciclos que un grafo

de x+1 nodos. Esto se puede ver claramente en el grafo de probabilidad 50 en los puntos 14 y 15, esto pudo haberse debido a que en el grafo de 14 nodos tuvo que analizar mas casos hasta llegar al conjunto dominante de 1 nodo y en el de 15 probo antes con ese conjunto (como no agarramos los nodos con algun criterio especifico nos vienen en el orden que el grafo nos los da con la funcion obtenerVertice).



# 3. Algoritmo Goloso

## 3.1. Solución

Proponemos una solución heurística golosa para encontrar un conjunto dominante de un grafo lo mas pequeño posible con una complejidad polinomial.
Para ello nos basamos en una estrategia de selección de nodos dominantes, la cual consiste en elegir el nodo que mas adyacentes no cubiertas tenga (es decir, que todavía no fueron dominadas) y verificando en cada iteración si el conjunto actual es dominante. En base a diferentes estrategias y tipos de grafos (estrella, bipartitos, completos, estrellas unidas, caminos, ciclos) que fuimos observando, elegimos esta ya que fue la que más nos convenció y mejores resultados nos dió debido a que siempre intenta seleccionar el nodo que mas pueda dominar a otros nodos reduciendo el conjunto de los no cubiertos y acercándose a una mejor solución del problema.

## 3.2. Pseudocódigo

MCDGreedy (**in** $Grafo$) $\longrightarrow$ *conjuntoDomGoloso* : ConjDeVértices

```
1  vértices = lista de vértices del grafo                                   ▷ O(n)
2  dominantes = conjunto vacio de vértices
3  Mientras no estén TodasCubiertas(vértices)                               ▷ O(n)
4      Ordeno los vértices por la cantidad adyacentes que tenga no cubiertas (sin dominar), de mayor a menor. ▷ O(n*log(n
5      Elijo como dominante al primero de la lista y lo agrego al conjunto de dominantes   ▷ O(n)
6      Saco de la lista de vértices al elegido                              ▷ O(n)
7      ActualizarGradoSinDominar(elegido)                                   ▷ Actualizo los nodos del grafo,
   disminuyendo la cantidad de grado sin dominar de los adyacentes al elegido, y de los adyacentes a estos. O(n^2)
8  return dominantes
```

---

TodasCubiertas (**in** $ListaDeVertices$) $\longrightarrow$ *result* : Boolean

```
1  Para todos los vértices en la lista
2      Si el vértice no esta dominado
3          return falso
4  return verdadero
```

---

ActualizarGradoSinDominar (**in** $Vertice\ elegido$)

```
1  Marcar al vértice elegido como dominado.
2  unionDeAdyacentes = lista de vértices vacía, que luego se usará para actualizar.
3  Para todos los vértices en la lista de adyacentes al elegido
4      Si el vértice elegido no estaba dominado
5          Decremento en uno el grado de adyacentes sin dominar del nodo actual
6      Si el nodo actual elegido no estaba dominado
7          Agrego los nodos adyacentes a la lista unionDeAdyacentes
8      Marco al nodo actual como dominado.
9  Para todos los vértices en la lista unionDeAdyacentes
10     Decremento en uno el grado de adyacentes sin dominar del nodo actual
```

---

## 3.3. Análisis de Complejidad

La complejidad del algoritmo es de O($n^3$)
El algoritmo comienza creando una lista de vértices en O(n).
Luego entra en un ciclo que como máximo será lineal en la cantidad de vértices porque en cada ciclo saca un vértice y la cantidad de vértices es finita. En cada iteración deberá comprobar si el conjunto actual de vértices elegidos domina todo el grafo en O($n$), ordenar todos los vértices en O(n*log(n)), elegir un vértice en O(1), removerlo de la lista en O(n) y finalmente actualizar el atributo de los grados sin dominar adyacentes de cada nodo en O($n^2$), esto es porque para los adyacentes del nodo elegido y a su vez para los adyacentes de estos tengo que bajar en uno este atributo, recorrer los adyacentes me lleva O(n) y por cada adyacente recorrer sus adyacentes también me lleva O(n), O(n) * O(n) = O($n^2$).

Por lo tanto, la complejidad nos queda:
O(n) + O(n) * (O($n^2$) + O(n*log(n) + O(n) + O($n^2$)) = O($n^3$)

La complejidad final del algoritmo goloso es de O($n^3$), cumpliendo el objetivo de ser polinomial.

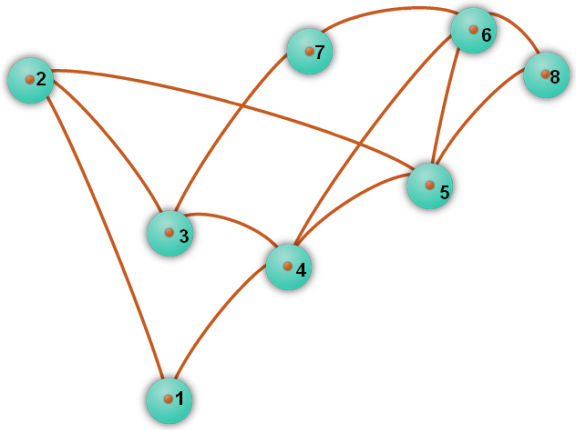
## 3.4. Mejor Caso

Las familias de grafos en donde la solución es la mejor son en los grafos bipartitos completos, triangulos unidos, estrellas, arboles binarios, arbol de cliques. Por que? Porque en cada ordenamiento y selección de nodos, el nodo con más grado sin dominar es exactemente el que hay que elegir. EXPLICAR MAS

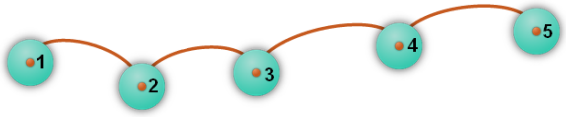
## 3.5. Peor Caso

Al ser una heurística, en cierto casos la solución no es la óptima que podría dar un algoritmo exacto, pero aún asi es correcta. Los peores casos, donde se produce una diferencia en el tamaño de conjunto dominante respecto a la optima, se suelen dar en grafos en los que varios nodos tienen el mismo grado o hay pequeña diferencia, y esto es debido que para la elección del vertice dominante en cada iteración nos quedamos con el que mayor grado de nodos adyacentes no cubiertos tenga y si hay varios con esta misma característica puede pasar que el nodo seleccionado no sea conveniente a futuro para llegar a una solución óptima.

En los siguientes ejemplos de grafos se puede apreciar mejor:




La solución optima proporcionada por el algoritmo exacto es 6,2, mientras que el goloso devuelve 4,3,5, ya que como los nodos 6,4 y 5 tienen el mismo grado, al momento de la elección se decide por el 4 provocando esta diferencia en el tamaño del conjunto con respecto a la solución óptima.



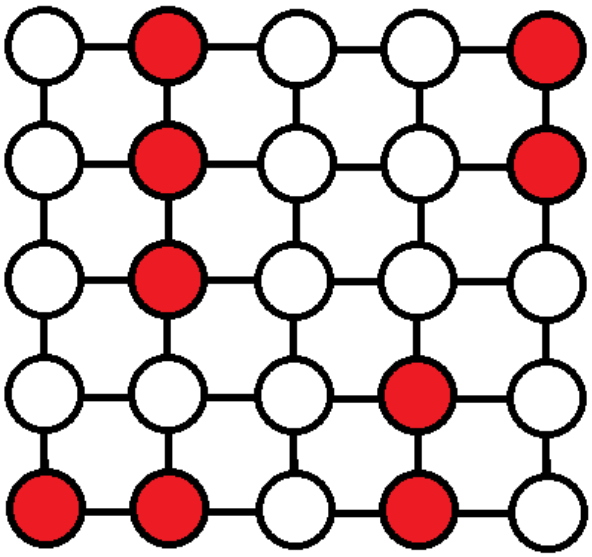

La solución óptima proporcionada por el algoritmo exacto es 2,4, mientras que el goloso podría devolver 3,2,5, ya que como los nodos 2, 3 y 4 tienen el mismo grado, si se decide por el 3, dejaría a todos los demás nodos con 1 grado sin dominar y faltando los dos extremos, provocando que si o si se necesite cubrirlos o elegirlos como dominantes, haciendo que el conjunto tenga tamaño 3 y no 2 como el exacto.

### 3.5.1. Familias

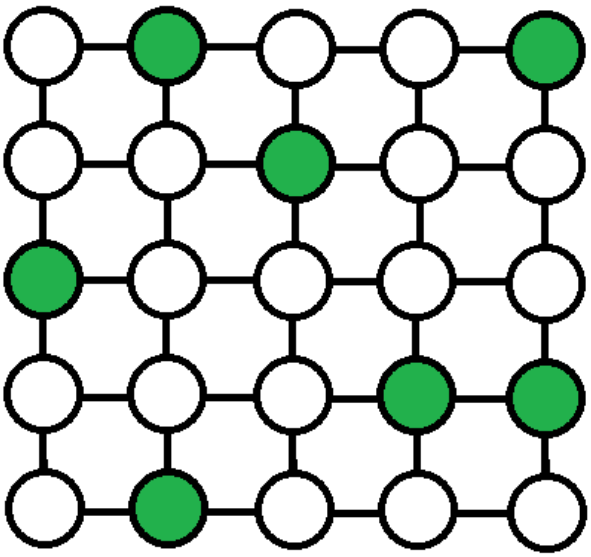
Las familias de grafos que encontramos que el algoritmo goloso da un resultado no optimo frente al exacto fue con los Möbius–Kantor y Grid. Esto suele pasar por que son regulares (3-regular) y casi regulares (4-regular internamente, 3-regular en los bordes) respectivamente y al momento de la elección del mejor vértice, la estrategia solo distingue por grado de adyacentes sin dominar por lo tanto seleccionará por el orden de la etiqueta de los nodos, las cuales son arbitrarias.

Considerando que Möbius–Kantor se puede reducir a un circuito, nos pareció mas interesante enfocarnos en los grafos Grid, ya que es más amplia la diferencia de resultados entre los dos algoritmos, además esta familia de grafos son de más utilidad, ya que equivalen tableros representando cada celda con un nodo.

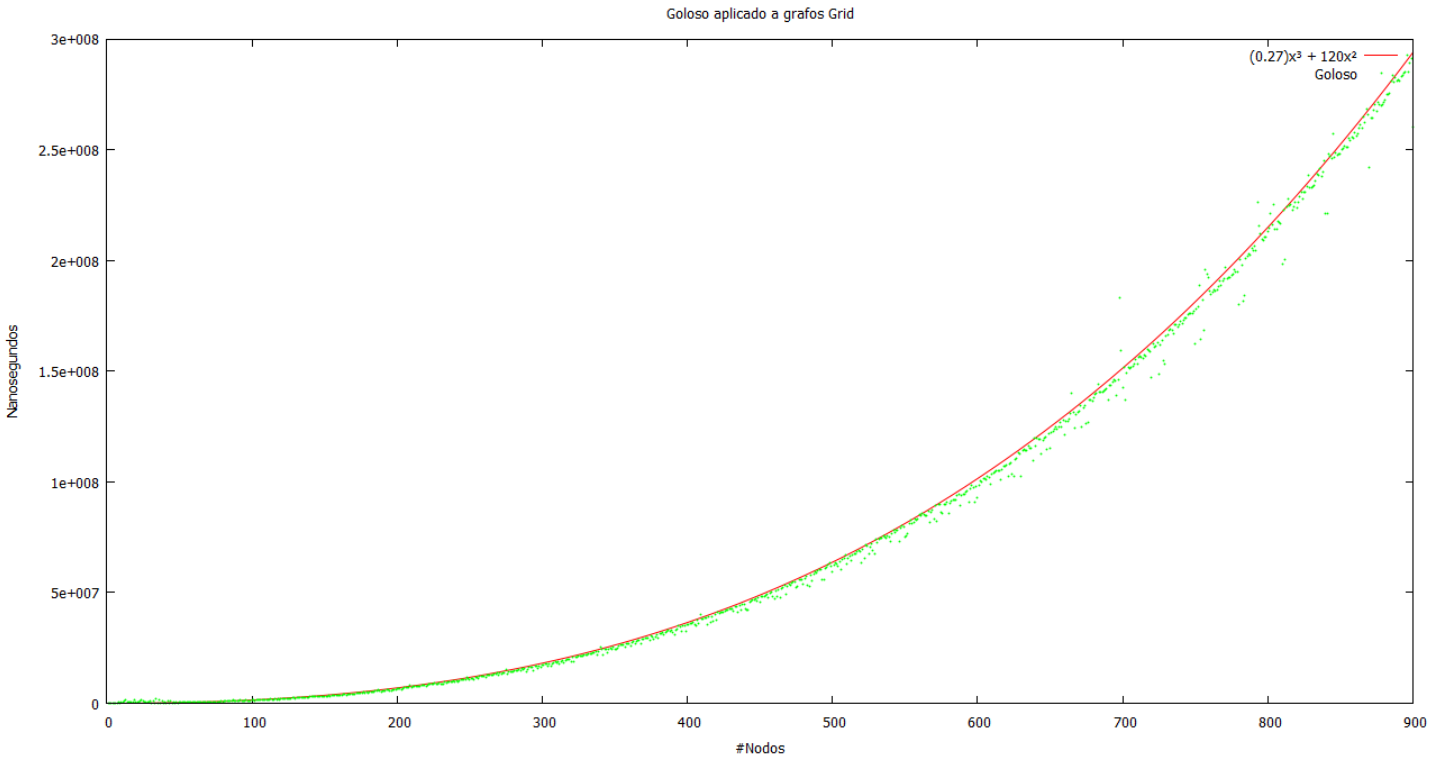
En el siguiente ejemplo, para un grafo Grid de 5x5 con 25 nodos, se puede notar la diferencia entre cantidad de elementos de los conjunto dominante resultantes, el goloso devuelve 9 nodos 6, 18, 21, 9, 1, 11, 20, 4, 23, mientras que el exacto 7 nodos 19, 18, 10, 7, 4, 1, 21, y esto se debe principalmente a que el goloso eligirá primero los nodos internos de grado 4 hasta que tengan el mismo grado sin dominar que los del borde y luego seleccionará los que faltan para dominar a todos, mientras que la solución exacta dependia más de los nodos en los bordes que los interiores.



Goloso en Grid 5x5



Exacto en Grid 5x5



| $m \setminus n$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 2 | 1 | 2 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 3 | 1 | 2 | 3 | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 4 | 2 | 3 | 4 | 4 | | | | | | | | | | | | | | | | | | | | | | | | | |
| 5 | 2 | 3 | 4 | 6 | 7 | | | | | | | | | | | | | | | | | | | | | | | | |
| 6 | 2 | 4 | 5 | 7 | 8 | 10 | | | | | | | | | | | | | | | | | | | | | | | |
| 7 | 3 | 4 | 6 | 7 | 9 | 11 | 12 | | | | | | | | | | | | | | | | | | | | | | |
| 8 | 3 | 5 | 7 | 8 | 11 | 12 | 14 | 16 | | | | | | | | | | | | | | | | | | | | | |
| 9 | 3 | 5 | 7 | 10 | 12 | 14 | 16 | 18 | 20 | | | | | | | | | | | | | | | | | | | | |
| 10 | 4 | 6 | 8 | 10 | 13 | 16 | 17 | 20 | 22 | 24 | | | | | | | | | | | | | | | | | | | |
| 11 | 4 | 6 | 9 | 11 | 14 | 17 | 19 | 22 | 24 | 27 | 29 | | | | | | | | | | | | | | | | | | |
| 12 | 4 | 7 | 10 | 12 | 16 | 18 | 21 | 24 | 26 | 29 | 32 | 35 | | | | | | | | | | | | | | | | | |
| 13 | 5 | 7 | 10 | 13 | 17 | 20 | 22 | 26 | 29 | 31 | 35 | 38 | 40 | | | | | | | | | | | | | | | | |
| 14 | 5 | 8 | 11 | 14 | 18 | 21 | 24 | 28 | 31 | 34 | 37 | 40 | 44 | 47 | | | | | | | | | | | | | | | |
| 15 | 5 | 8 | 12 | 15 | 19 | 22 | 26 | 29 | 33 | 36 | 40 | 43 | 47 | 50 | 53 | | | | | | | | | | | | | | |
| 16 | 6 | 9 | 13 | 16 | 20 | 24 | 27 | 31 | 35 | 38 | 42 | 46 | 49 | 53 | 57 | 60 | | | | | | | | | | | | | |
| 17 | 6 | 9 | 13 | 17 | 22 | 26 | 29 | 33 | 37 | 41 | 45 | 49 | 53 | 56 | 60 | 64 | 68 | | | | | | | | | | | | |
| 18 | 6 | 10 | 14 | 18 | 23 | 27 | 31 | 35 | 39 | 43 | 47 | 51 | 55 | 60 | 64 | 68 | 72 | 76 | | | | | | | | | | | |
| 19 | 7 | 10 | 15 | 19 | 24 | 28 | 32 | 37 | 41 | 45 | 50 | 54 | 58 | 63 | 67 | 71 | 75 | 80 | 84 | | | | | | | | | | |
| 20 | 7 | 11 | 16 | 20 | 25 | 30 | 34 | 39 | 43 | 48 | 52 | 57 | 62 | 66 | 70 | 75 | 79 | 84 | 88 | 92 | | | | | | | | | |
| 21 | 7 | 11 | 16 | 21 | 26 | 31 | 36 | 41 | 45 | 50 | 55 | 60 | 64 | 69 | 74 | 78 | 83 | 88 | 92 | 97 | 101 | | | | | | | | |
| 22 | 8 | 12 | 17 | 22 | 28 | 32 | 37 | 43 | 47 | 52 | 57 | 62 | 67 | 72 | 77 | 82 | 87 | 92 | 96 | 101 | 106 | 111 | | | | | | | |
| 23 | 8 | 12 | 18 | 23 | 29 | 34 | 39 | 44 | 49 | 54 | 60 | 65 | 70 | 75 | 80 | 86 | 91 | 96 | 101 | 106 | 111 | 116 | 121 | | | | | | |
| 24 | 8 | 13 | 19 | 24 | 30 | 36 | 41 | 46 | 52 | 57 | 63 | 68 | 73 | 79 | 84 | 89 | 94 | 100 | 105 | 110 | 115 | 120 | 126 | 131 | | | | | |
| 25 | 9 | 13 | 19 | 25 | 31 | 37 | 42 | 48 | 54 | 59 | 65 | 71 | 76 | 82 | 87 | 93 | 98 | 104 | 109 | 114 | 120 | 125 | 131 | 136 | 141 | | | | |
| 26 | 9 | 14 | 20 | 26 | 32 | 38 | 44 | 50 | 56 | 62 | 68 | 74 | 79 | 85 | 91 | 96 | 102 | 108 | 113 | 119 | 124 | 130 | 136 | 141 | 146 | 152 | | | |
| 27 | 9 | 14 | 21 | 27 | 34 | 40 | 46 | 52 | 58 | 64 | 70 | 76 | 82 | 88 | 94 | 100 | 106 | 112 | 117 | 123 | 129 | 135 | 141 | 146 | 152 | 158 | 164 | | |
| 28 | 10 | 15 | 22 | 28 | 35 | 41 | 47 | 54 | 60 | 66 | 73 | 79 | 85 | 91 | 97 | 104 | 110 | 116 | 122 | 128 | 134 | 140 | 146 | 152 | 158 | 164 | 170 | 176 | |
| 29 | 10 | 15 | 22 | 29 | 36 | 42 | 49 | 56 | 62 | 69 | 75 | 82 | 88 | 94 | 101 | 107 | 113 | 120 | 126 | 132 | 138 | 144 | 151 | 157 | 163 | 169 | 175 | 182 | 188 |

| m/n | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 2 | 1 | 2 | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 3 | 1 | 2 | 3 | | | | | | | | | | | | | | | | | | | | | | | | | |
| 4 | 2 | 3 | 4 | 6 | | | | | | | | | | | | | | | | | | | | | | | | |
| 5 | 2 | 4 | 5 | 6 | 9 | | | | | | | | | | | | | | | | | | | | | | | |
| 6 | 2 | 4 | 6 | 8 | 10 | 10 | | | | | | | | | | | | | | | | | | | | | | |
| 7 | 3 | 5 | 6 | 9 | 13 | 13 | 15 | | | | | | | | | | | | | | | | | | | | | |
| 8 | 4 | 5 | 8 | 12 | 12 | 16 | 15 | 21 | | | | | | | | | | | | | | | | | | | | |
| 9 | 3 | 6 | 9 | 11 | 13 | 16 | 21 | 23 | 28 | | | | | | | | | | | | | | | | | | | |
| 10 | 4 | 7 | 10 | 14 | 15 | 19 | 20 | 27 | 28 | 30 | | | | | | | | | | | | | | | | | | |
| 11 | 5 | 8 | 11 | 15 | 17 | 21 | 23 | 29 | 31 | 33 | 39 | | | | | | | | | | | | | | | | | |
| 12 | 6 | 8 | 11 | 16 | 17 | 21 | 26 | 31 | 34 | 36 | 43 | 45 | | | | | | | | | | | | | | | | |
| 13 | 6 | 8 | 14 | 18 | 18 | 22 | 25 | 34 | 39 | 43 | 45 | 49 | 53 | | | | | | | | | | | | | | | |
| 14 | 6 | 9 | 14 | 16 | 21 | 23 | 28 | 34 | 39 | 43 | 49 | 51 | 54 | 57 | | | | | | | | | | | | | | |
| 15 | 7 | 9 | 15 | 18 | 21 | 28 | 31 | 40 | 41 | 47 | 51 | 53 | 61 | 64 | 65 | | | | | | | | | | | | | |
| 16 | 7 | 10 | 16 | 19 | 22 | 31 | 32 | 40 | 45 | 51 | 54 | 59 | 63 | 67 | 69 | 80 | | | | | | | | | | | | |
| 17 | 7 | 11 | 17 | 24 | 24 | 29 | 36 | 41 | 48 | 56 | 56 | 61 | 65 | 69 | 77 | 84 | 83 | | | | | | | | | | | |
| 18 | 6 | 11 | 18 | 24 | 25 | 33 | 37 | 40 | 50 | 58 | 61 | 64 | 69 | 76 | 81 | 87 | 89 | 95 | | | | | | | | | | |
| 19 | 7 | 12 | 19 | 27 | 25 | 35 | 39 | 45 | 52 | 61 | 64 | 69 | 74 | 79 | 85 | 94 | 94 | 99 | 109 | | | | | | | | | |
| 20 | 8 | 12 | 19 | 28 | 29 | 35 | 41 | 46 | 53 | 62 | 66 | 73 | 77 | 84 | 90 | 97 | 94 | 106 | 112 | 119 | | | | | | | | |
| 21 | 7 | 13 | 22 | 30 | 29 | 37 | 46 | 49 | 58 | 67 | 69 | 75 | 81 | 84 | 91 | 101 | 100 | 109 | 121 | 124 | 131 | | | | | | | |
| 22 | 8 | 13 | 22 | 31 | 31 | 40 | 49 | 53 | 61 | 69 | 74 | 77 | 83 | 88 | 95 | 104 | 105 | 112 | 127 | 131 | 137 | 142 | | | | | | |
| 23 | 9 | 13 | 23 | 33 | 31 | 38 | 51 | 56 | 64 | 71 | 76 | 83 | 90 | 93 | 97 | 109 | 110 | 119 | 133 | 139 | 139 | 153 | 158 | | | | | |
| 24 | 9 | 13 | 24 | 31 | 33 | 42 | 51 | 58 | 64 | 77 | 79 | 87 | 92 | 92 | 105 | 114 | 115 | 133 | 136 | 144 | 151 | 162 | 165 | 176 | | | | |
| 25 | 9 | 14 | 25 | 33 | 36 | 42 | 53 | 57 | 66 | 76 | 80 | 87 | 96 | 100 | 108 | 118 | 119 | 133 | 142 | 149 | 158 | 163 | 169 | 180 | 182 | | | |
| 26 | 10 | 16 | 26 | 38 | 39 | 46 | 55 | 58 | 72 | 83 | 85 | 91 | 102 | 105 | 110 | 122 | 125 | 140 | 151 | 152 | 164 | 172 | 183 | 189 | 189 | 203 | | |
| 27 | 10 | 16 | 27 | 38 | 43 | 45 | 56 | 60 | 71 | 83 | 89 | 97 | 107 | 110 | 114 | 126 | 129 | 141 | 156 | 157 | 166 | 177 | 183 | 197 | 200 | 217 | 217 | |
| 28 | 10 | 17 | 28 | 37 | 44 | 47 | 61 | 65 | 75 | 87 | 88 | 102 | 109 | 116 | 117 | 129 | 135 | 151 | 162 | 165 | 170 | 182 | 190 | 200 | 204 | 216 | 222 | 232 |
| 29 | 11 | 18 | 29 | 40 | 40 | 51 | 59 | 68 | 78 | 78 | 94 | 106 | 114 | 120 | 121 | 139 | 139 | 156 | 170 | 183 | 179 | 189 | 192 | 213 | 211 | 219 | 232 | 245 |

— Mejores casos: ESTRELLA Peores casos: GRID

# 4. Algoritmo Local Search

## 4.1. Solución

La idea de un algorítmo de búsqueda local (o local search) es, a partir de una solución (no óptima) a un problema dado, trabajar sobre ella, realizar distintas operaciones y estrategias de modo tal de mejorarla e intentar acercarse lo más posible a una solución óptima. La distinción de éstos algoritmos está en que cada modificación a la solución original, nos genera una o un conjunto de soluciones al problema, siempre c̈ercanasä la solución original. En éste sentido, definimos una *vecindad* de soluciones, que está formada por éstas nuevas soluciones cercanas a la original. En nuestro caso en particular, la idea es que, a partir de un grafo, hallemos un cojunto dominante de alguna forma (todos los nodos del grafo son, en efecto, un cojunto dominante. O bien también podemos usar la solución obtenida con el algoritmo greedy) y, a partir de ella, intentar construir, mediante estrategias y funciones creadas por nostros, una nueva solución, mejor o igual a la que nos proveyeron como base.
Es así que nos planteamos lo siguiente: ¿qué operaciones podemos realizar de forma tal que, a partir de un conjunto dominate, no necesariamente mínimo, podamos reducir la cardinalidad del conjunto y que continúe siendo dominante? Las posibilidades que encontramos fueron orientadas siempre a la posibilidad de reducir o, al menos mantener, la cantidad de nodos en el conjunto dominante. Es por eso que las estrategias que utilizamos en la búsqueda local son las siguientes:

**Reemplazar 2 nodos del Conjunto** dominante por uno que se encuentre dominado. Ésta es la forma más directa de reducir la cardinalidad del CD. La forma de seleccionar los nodos intercambiables la realizamos verificando si la unión de nodos vecinos de los dos nodos a quitar del conjunto está incluida en el conjunto formado por los vecinos del vértice a insertar en el CD. Ésta es la forma más sencilla de encontrar dichos nodos, ya que de otra forma, el costo de encontrar estos posibles reemplazos sería muy costoso. Además, existe la posibilidad de encontrar varios posibles candidatos, por lo que ideamos 3 funciones de comparación para ordenar éstas tuplas y, en cada caso, utilizar la más c̈onvenientesegún dicha función. Luego, para el método GRASP, eligiremos una de éstas a partir de las pruebas realizadas.

**Eliminar 1 nodo del CD:** Para éste caso, analizamos si eliminando alguno de los nodos del CD, el mismo sigue dominando a todos los nodos del grafo. Para ello, nos fijamos si el conjunto de vecinos del nodo que queremos quitar, está incluido en la unión de todos los vecinos de los nodos del CD menos el nodo que queremos quitar. Si es así, quitando el nodo el conjunto seguirá siendo dominante. En éste caso, es claro que la cardinalidad disminuye.

**Reemplar 1 nodo del CD** por otro del conjunto dominado. La estrategia aplicada para encontrar los posibles candidatos es similar a la que utilizamos en la técnica del 2x1, es decir, nos fijamos los nodos que estén en el CD, y buscamos aquellos dominados cuyos conjunto de vecinos contenga a los vecinos del nodo quitado (no tienen que ser los mismos exactamente, podria tener mas vecinos). Es claro que ésta técnica no reduce la cardinalidad del conjunto, pero la utilizamos con la esperanza de que éste cambio de nodos nos modifique la solución parcial y, a partir de ésta, si podamos utilzar algunas de las técnicas anteriores. Es decir, lo que intentamos realizar con ésta procedimiento es que al cambiar por otro nodo con un número de nodos adyacentes posiblemente mayor, al agregarse nuevas aristas, podamos luego sí, en el próximo paso, reducir la cardinalidad del CD.

El algoritmo de búsqueda local intentará, a partir de una solución dada (en la primera iteración, será la solución que se le otorgue a la hora de invocarla), mejorarla a partir de alguna de las tres técnicas mencionadas anteriormen. Primero, intentará quitar 2 nodos del conjunto e ingresar uno. Si no puede, intentará con la siguiente estrategia, que es quitar uno del conjunto dominante. Si tampoco puede reducir la cardinalidad con ésta técnica, intentará reemplazar 1 nodo del CD por un nodo dominado, para así al menos, cambiar la solución.
Si alguna de éstas técnicas tuvo éxito, el algoritmo no probará con las siguientes, y a partir de la solución hallada, intentará volver a crear una nueva ejecutando todas las técnicas nuevamente para la nueva solución. Evidentemenete, llegará una solución para la que nuestro algoritmo no podrá llevar a cabo ninguna de las técnicas especificadas, y es ahí donde termina la búsqueda local, y nuestro algoritmo, dando como solución a la última encontrada, y que tendrá una cardinalidad igual o menor a la solución inicial.
A continuación, mostramos el pseudocódigo de nuestra algoritmo:

## 4.2. Pseudocódigo

MejorarSolucion (**ConjuntoDominante** *cd*, **Comparator** *funcion*) $\longrightarrow$ *cd* : Conjunto Dominante

```
1  mejoroSolucion = true
2  hasta mejoroSolucion no sea false:
3        mejoroSolucion = ElegirYEjecutarEstrategia(cd, funcion)
4  return cd
```

---

ElegirYEjecutarEstrategia (**ConjuntoDominante** *cd*, **Comparator** *f*) $\longrightarrow$ *res* : Boolean

```
1  Si es posible llevar a cabo la estrategia 2x1 con la funcion f:
2      devuelvo true
3  Si es posible llevar a cabo la estrategia quitar un nodo:
4      devuelvo true
5  Si es posible llevar a cabo la estrategia 1x1:
6      devuelvo true
7  return false
```

---

TryEstrategia2x1 (**ConjuntoDominante** *cd*, **Comparator** *f*) $\longrightarrow$ *res* : Boolean

```
1  obtengo los posibles nodos intercambiables, ordenados por la funcion f
2  Si existen posibles nodos intercambiables:
3      obtengo la primera tupla de nodos intercambiables
4      intercambio los nodos de la tupla en el cd
5      devuelvo true
6      sino devuelvo false
7  return false
```

---

TryEstrategiaQuitarUno (**ConjuntoDominante** *cd*, **Comparator** *f*) $\longrightarrow$ *res* : Boolean

```
1  para cada vertice v del conjunto dominante:
2      quito el vertice v del cd
3      Si el conjunto sigue siendo dominante:
4          devuelvo true
5  return false
```

---

TryEstrategia1x1 (**ConjuntoDominante** *cd*, **Comparator** *f*) $\longrightarrow$ *res* : Boolean

```
1  para cada vertice v del conjunto dominante:
2      para cada vertice d del conjunto dominado:
3          si no realize el intercambio v,d y los nodos son reemplazables:
4              intercambio los vertices en el cd
5              devuelvo true
6  return false
```

---

## 4.3. Análisis de Complejidad

Para todos los casos, se corre al menos k veces, y para cada una de esas k corridas, se utilizan 3 métodos para intentar hallar una nueva solucion:

**2x1:** Se crea una cola de nodos, ordenados por una funcion f (pasada por parametro). Para ello, se recorren todos los posibles pares de nodos dominantes, y para cada par, se analiza si es posible reemplazarlo por un nodo del conjunto dominado. La complejidad de crear esta cola es de $O(n^2)$, siempre y cuando se considere que $n^2$ es mayor a m, siendo m la cantidad de vertices del conjunto dominado. Luego, para hacer el intercambio, se realizan operaciones en ArrayList, que son $O(n)$. Por lo que la complejidad de éste método es de $O(n^2)$.

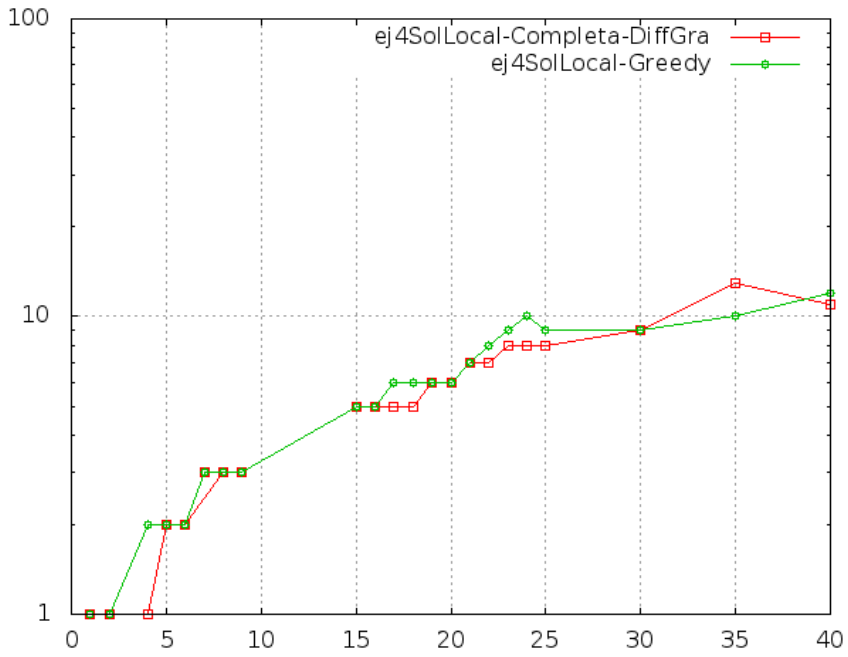
**QuitarUno:** en éste caso, se recorren todos los nodos del conjunto dominante, y se pregunta si, quitando el nodo en cuestion, el conjunto continúa siendo dominante. Ésta método tiene una complejidad de $O(nxm)$, donde n es la cantidad de nodos del cd, y m es la cantidad de aristas de cada nodo.

**UnOxUnO** ésta estrategia requiere de recorrer todos los vértices dominantes, y para cada uno de ellos, compararlo con todos los vértices dominados, y fijarme si puedo intercambiar éstos nodos (en cuestión, si tienen los mismos vecinos) de forma tal que el conjunto continue siendo dominante. Esta verificación es de complejidad $O(nxm)$, donde n es la cantidad de nodos del conjunto dominante, y m la cantidad de nodos del conjunto dominado.

Es decir, que la complejidad en todos los casos, no es mayor a $O(n^2)$, ya que en todos los casos se analiza primero la opción de hacer el 2x1. Si puede, la complejidad será esa, y si no puede, seguirá con los otros métodos que, a rasgos generales, nunca superan ésta cota de $O(n^2)$. Por lo que la complejidad general del algoritmo es de $O(n^2)$.

## 4.4. Peor Caso

Es claro que, al ser una heurística, en muchos casos la solución obtenida por la búsqueda local (o local search) no es la óptima, pero de todas formas, se trata de soluciones çercanas.ª ella. En éste sentido, se dan cuando los nodos de la solución de la cual se parte la búsqueda loca, tienen muchas aristas conectadas entre si o a otros vértices ya dominados (los grados de los nodos son muy altos). Ésto hace que, por ejemplo, las estrategias elegidas no puedan intercambiar nodos, o quitar nodos de la solución de la que se parta y, sin embargo, en la práctica si exista una solución con menor cantidad de nodos.
En particular, hemos encontrado dos casos generales distintos: por un lado, aquellos en que el algorítmo de búsqueda local mejora la solución de la cual se parte pero, sin embargo, no llega a una solución óptima (calculada a partir del mismo grafo con el algoritmo exácto). Y por otro lado están aquellas instancias que, a partir de una solución, no pueden ser mejoradas por la búsqueda local, a pesar de haber una solución óptima y exacta con una menor cantidad de nodos. Éstos dos casos se pueden ver en los tests que hemos corrido en el jUnit, y que lamentablemente, no hemos podido correr para instancias de más de 25 nodos, ya que la solución exacta (que es de orden exponencial) no llega a terminarse y, por tanto, no podemos comparar los valores de la búsqueda local con los valores exactos.

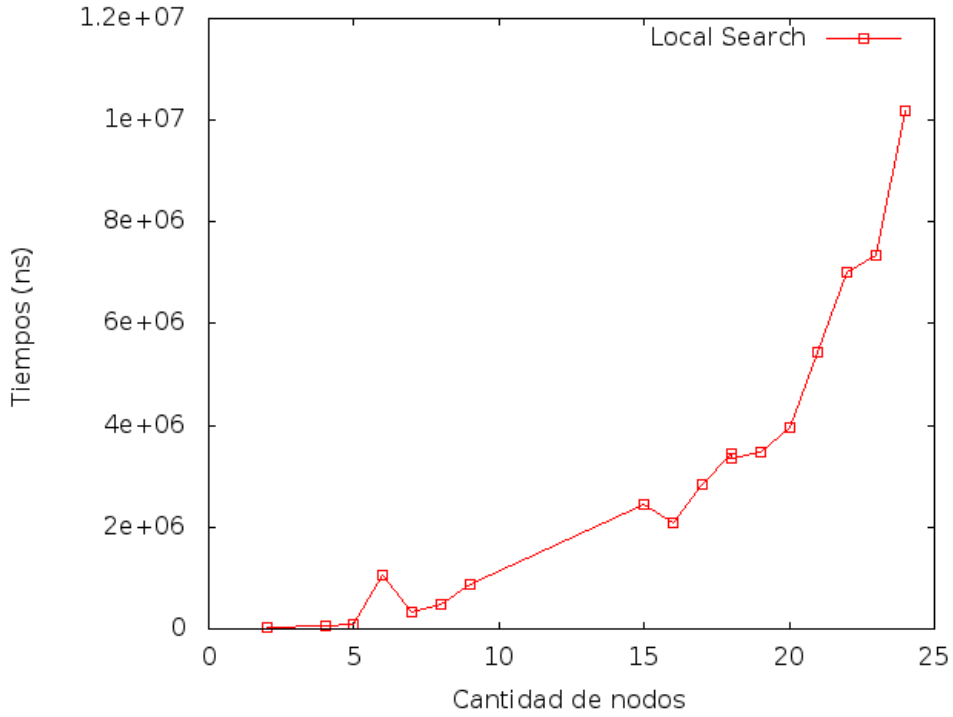



Éstos resultados, eran de esperarse, ya que se trata de una heurísitca y sólo en algunos casos partículares se consiguen los mismos resultados que en los algoritmos exáctos. La idea, entonces, es intentar de construir las mejores estrategias para poder llegar a un resultado lo más exacto posible, para instancias en que sea imposible (al menos hoy en día) correr el algoritmo exácto de orden exponencial. En éste caso particular, de la búsqueda local, seguramente haya muchas estrategias para mejorar o agregar, que nos aproximen a una solución un poco más exacta de la que encontramos hasta ahora, pero no pudimos dar con ellas.

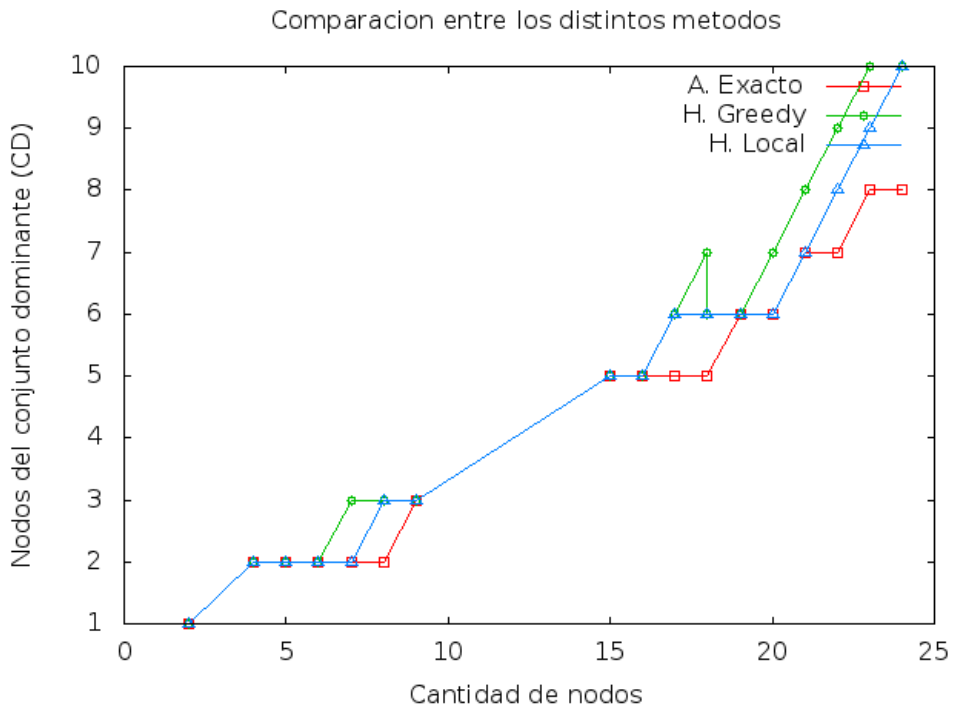
## 4.5. Tests y análisis

En primer lugar, hemos hecho test para encontrar cuál de todas las funciones encuentra los nodos óptimos para el intercambio de nodos 2x1. En la teoría, todas analizan factores distintos, pero en la práctica, al ser muy pocos los nodos que pudimos intercambiar, las distintas funciones no han aportados cambios significativos en la cardinalidad de las soluciones. A lo sumo, ésta han diferido en 1 nodo, no más, y la que ha resultado más efectiva ha sido la de encontrar la diferencia de grados entre la tupla a quitar y el nodo a insertar. Es por eso que ésta función ha sido la que utilizamos para las pruebas restantes.
Una vez establecida qué función deberíamos utilizar para el 2x1, nos abocamos a testear el método de búsqueda local de acuerdo a la cantidad de nodos del grafo y el tiempo que le consume encontrar una solución. Vale aclarar que no tomamos en consideración el tiempo que nos consume encontrar una solución para darle a la búsqueda loca, ya que consideramos que ello no es parte de nuestro algoritmo. De ésta forma, podemos ver que nuestro algoritmo se comporta de forma polinómica en función del tiempo, como lo analizamos en la sección de complejidad. Es decir, mientras mayor cantidad de nodos tenga la solución que nos proveen, y más relacionados estén entre sí dichos nodos (es decir, a través de aristas hacia nodos del mismo conjunto o del conjunto de los dominados), mayor tiempo requerirá calcular nuestra solución. Además, y de forma conjunta con éste crecimiento, aumenta la cantidad de operaciones requeridas por el algortimo para finalizar cada iteración $k$, incremento que se ve directamente

relacionado con el aumento de la cardinalidad de conjunto dominante. Ésto puede verse en el gráfico 1, de forma tal que a mayor cantidad de nodos que le pasamos a nuestra solución, mayor será el tiempo que le toma resolverlo, debido a que debe realizar más operaciones.




Además, creemos conveniente probar / mostrar la mejoría que realiza nuestro algoritmo a las soluciones que se le dan. Para ello, comparamos la ejecución de la misma instancia para el algoritmo exacto, el algoritmo greedy y el algoritmo de solución local, y la calidad de la solución (la cardinalidad de las mismas), mostrando cómo mejoran las mismas de acuerdo a cada método. Es válido aclarar que en todas las corridas, hemos utilizado como instancia de entrada para el local search la solución dada por el algoritmo greedy, por lo que en todos los casos, la cardinalidad de las soluciones de la búsqueda loca, será por lo menos, igual a la greedy y como máximo, si tuvimos suerte, igual a la exacta.



# 5. Algoritmo GRASP

## 5.1. Solución

La idea de este algoritmo es ejecutar una serie de veces nuestro algoritmo GREEDY randomizado y buscar en cada solución una mejora local, guardando la mejor solución hasta el momento. Gracias a la parte aleatoria del algoritmo Greedy, aplicar repetidas veces este algoritmo nos otorga soluciones obtenidas de distinta forma en cada caso, de distintas partes del espectro de soluciones, las cuales llevamos a mínimos locales para allí comparar sus cardinales y quedarnos con la menor. Nuestra esperanza es, entonces,poder caer en un mínimo global. Esa randomización se debe a que el método por el cuál dicho algoritmo goloso elige siempre el mejor valor bajo su criterio, es ahora aleatorizada seleccionando al azar entre los primeros k elementos de la lista de nodos a ser considerado dominante.
Hablándo un poco más en lenguaje de GRASP, la RCL (Restricted Candidate List) es la lista que contiene un conjunto de posibles mejores candidatos según el criterio de nuestro Greedy y de ella eligiremos un valor al azar. Esta lista decidimos armarla desde el algoritmo Greedy. Para ello, agregamos en el algoritmo Greedy que reciba por parámetro dicho k, de forma tal que al momento de elegir el próximo nodo, en vez de elegir el primero de los nodos ordenados por la cantidad de adyacentes sin dominar, ahora elige la posición i, que será el número obtenido pseudo-aleatoriamente a través de la función random entre 0 y k. En caso de que este k sea mayor a la cantidad de nodos considerados en la lista, k se definirá como la longitud de dicha lista.
En las heurísticas GRASP es necesario elegir un criterio de parada para que el algoritmo no se ejecute infinitamente. En nuestro caso, decidimos elegir unicamente una cantidad de iteraciones, ya que esto nos permitió jugar libremente con ese parámetro y no preocuparnos por casos en los que la solución inicial fuera optima y nunca se realizara mejora.

## 5.2. Pseudocódigo

MCDGrasp (**in** $Grafo$, **in** $k$) $\longrightarrow$ $ConjDominante$ : Conj

```
1  mejorSolucion = TodosLosVertices
2  Para i=0 hasta k
3        instanciaSolucionGreedyRandomized = MCDGreedy(grafo,k)
4        nuevaSolucion = MCDLocalSearch(instanciaSolucionGreedyRandomized)
5        Si cantNodosDominantes(nuevaSolucion) < cantNodosDominantes(mejorSolucion)
6              mejorSolucion = nuevaSolucion
7              Si mejorSolucion.size() == 1
8                    return mejorSolucion
9  return mejorSolucion
```

---

Comenzamos con todos los vertices como mejor solución, de forma tal que la próxima solución siempre sera mejor o igual.
Se debe aclarar que tanto LocalSearch como Greedy devuelven ConjuntosDominantes verdaderos, por lo que lo único que importa es la cantidad de nodos devuelta.

A continuación se agrega el cambio hecho en el algoritmo greedy para aleatorizar la selección de nodos:

elegirVertice (**in** $ListaVerticesORdenada$, **in** $k$) $\longrightarrow$ $proxVertice$ : Vertice

```
1  Si k > |ListaVerticesORdenada|
2        k = |ListaVerticesORdenada|;
3  posiciónAElegir = random(0,k);
4  proxVertice = ListaVerticesORdenada[posiciónAElegir]
5  return proxVertice
```

---

## 5.3. Análisis de Complejidad

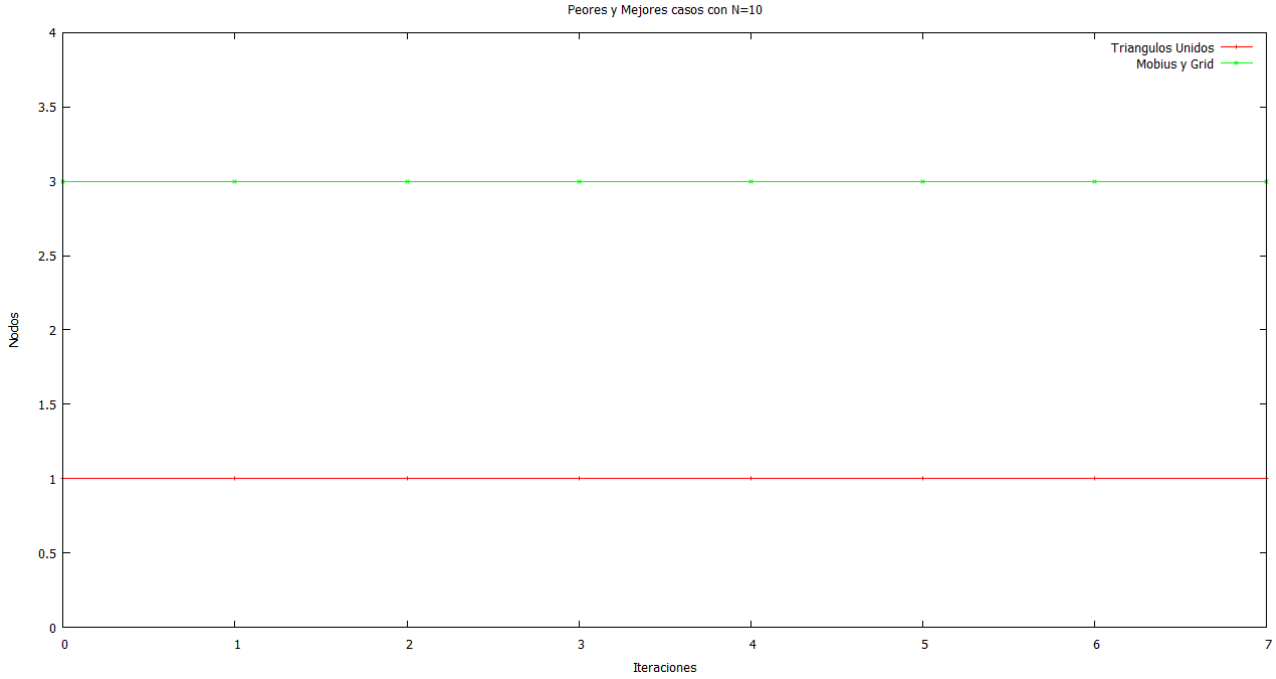
Por la demostración de la complejidad de la búsqueda local, sabemos que la complejidad es: O($n^2$);
Por la demostración de la complejidad del goloso, sabemos que la complejidad es: O($n^3$);
Dentro del GRASP se realizan k iteraciones de las siguientes operaciones:

- Ejecutar algoritmo goloso con random
- Ejecutar algoritmo localSearch a partir de la solución del goloso
- Comparar mejorSolución con nuevaSolución (esta operación es en O(1) ya que tanto conseguir el tamaño de un ArrayList en Java como la comparación de enteros es en O(1))

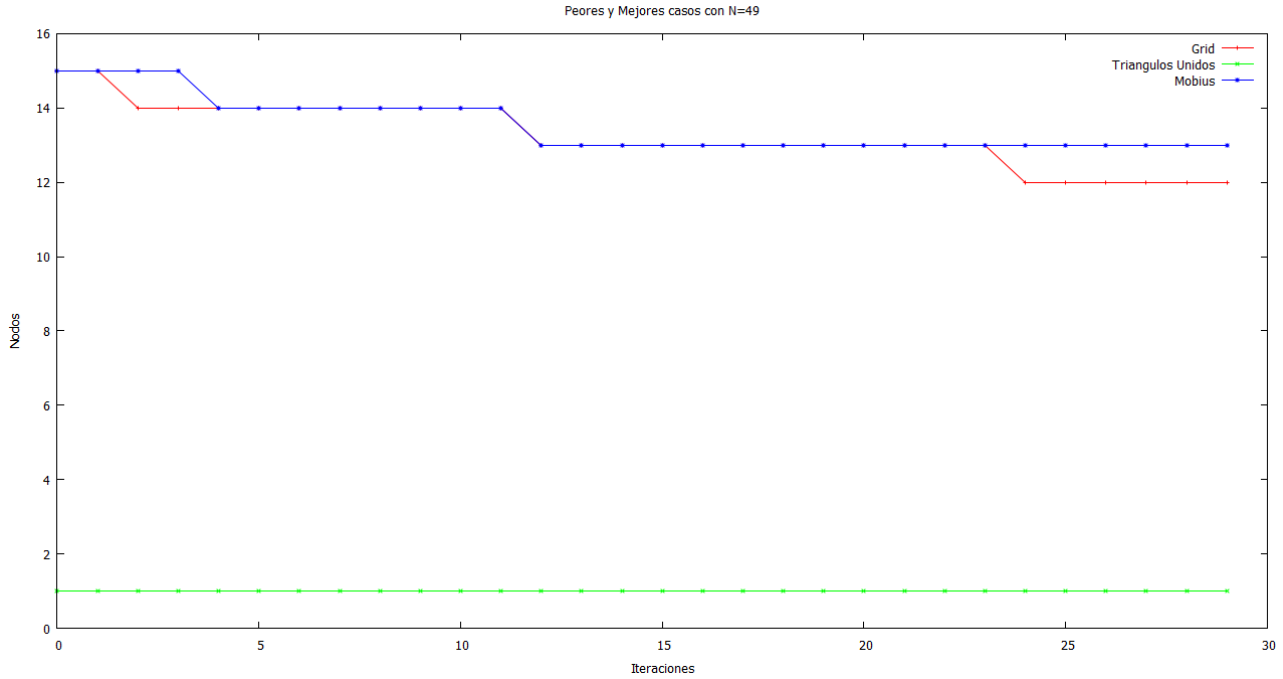
Esto es O(k*(goloso + localSearch)), que resulta polinomial.
Entonces la complejidad resulta O(k*$n^3$);

## 5.4. Tests y análisis

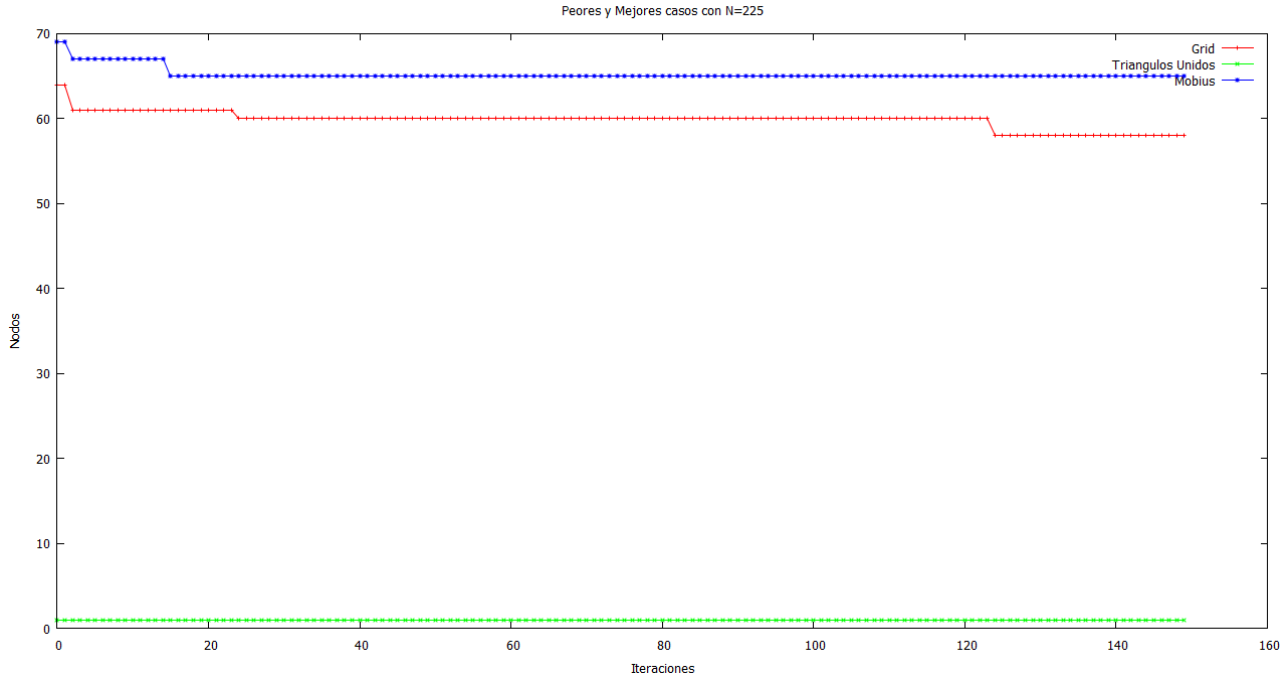
Tal como hicimos con los anteriores algoritmos, acá también se tomaron los peores y los mejores casos. Como la randomización evita encontrar un peor/mejor caso per-se, se decidió correr los mencioandos en los puntos anteriores. Retomando un comentario anterior, GRASP te da la oportunidad de recorrer más el espacio de soluciones y buscar en cada uno una mejor versión vecinal, gracias a esto podemos socorrer los casos en donde Greedy o Greedy+LocalSearch no son óptimos gracias a esta cuota de randomización. Muchas veces esto pasa porque Greedy encuentra una solución que localmente es buena, lo que reduce el marco de posibles soluciones locales. Cuando ejecutamos GRASP, el greedy ejecutado en él notamos que devuelve un peor caso que el goloso original, pero eso provee un espacio vecinal mucho más grande para que el Local Search pueda trabajar.
Al igual que en la búsqueda local, en este algoritmo tambien se incrementa la cantidad de operaciones a medida que se incrementa la cantidad de conjuntos dominantes devuelto por la solución greedy.
Cabe recordar que en cada iteración, el algoritmo greedy retorna una solución de las k*|conjuntoDominante| posibles soluciones, que después se buscará localmente una mejora en caso de existir a través del camino elegido.
Antes de hacer los gráficos, deberíamos calcular cual es el peor caso para este algoritmo. Sin embargo, el mismo depende de dos cosas: La solución random generada por el algoritmo goloso cuando se usa el valor de k. Esto es importante, ya que mientras esa solución sea generada de forma pseudo-random, menor cantidad de operaciones va a haber en la búsqueda local. A continuación se muestran algunos de los ejemplos realizados para distintas cantidades de nodos, distintas iteraciones obteniendo resultados en los peores y mejores casos.




Peores y mejores casos con n = 10.



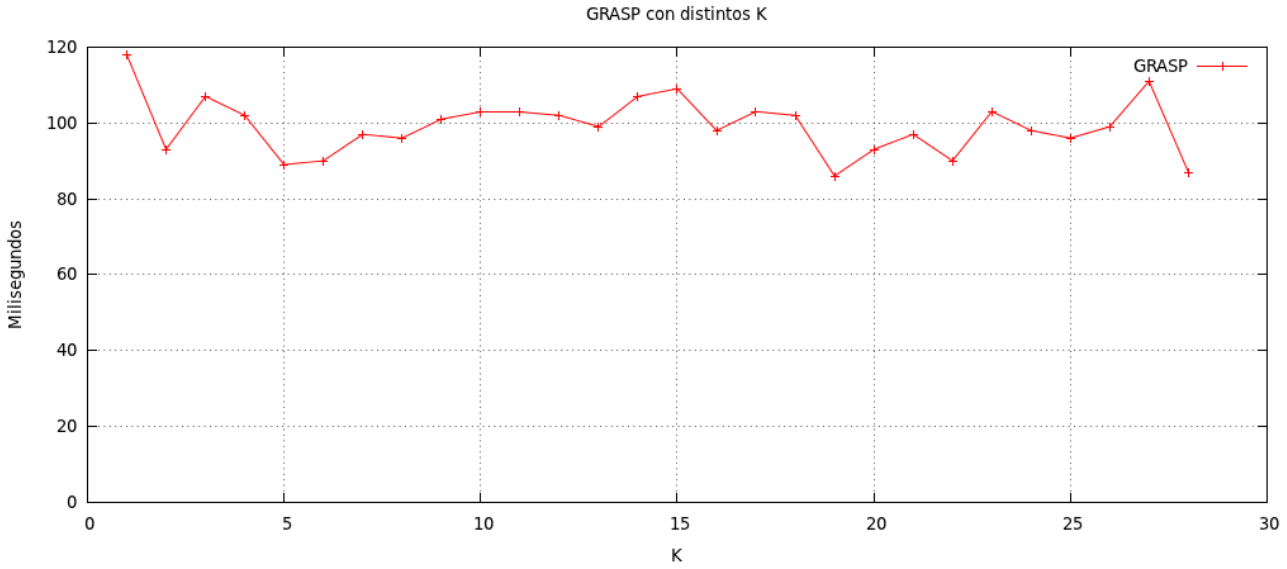

Peores y mejores casos con n = 49.



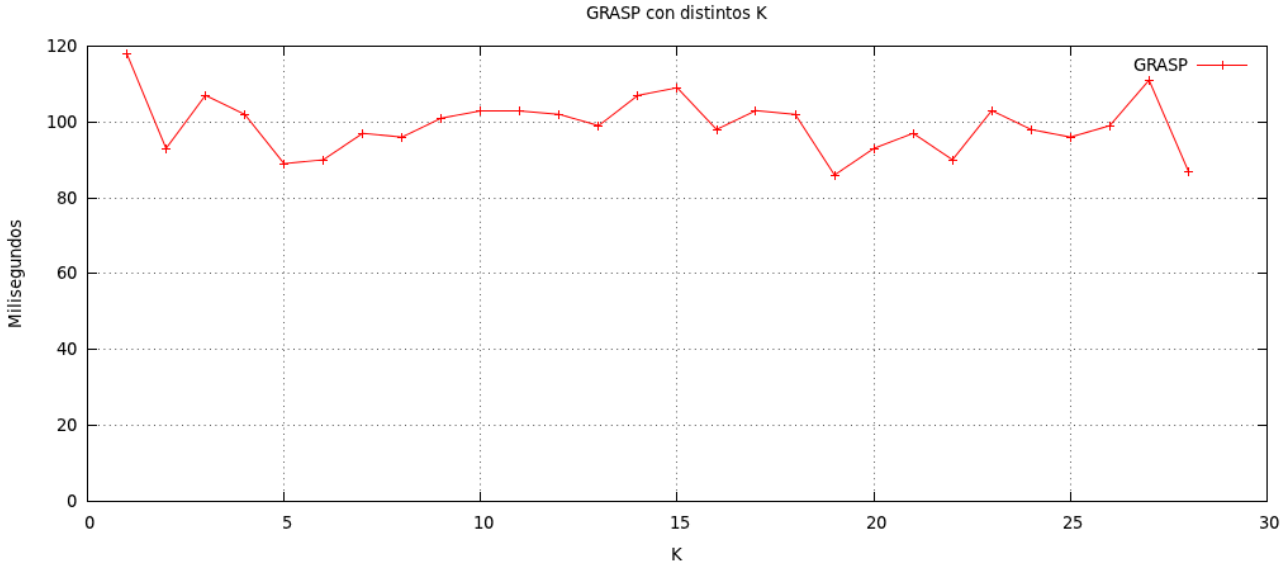

Peores y mejores casos con n = 225.

Como se puede ver en los gráficos, la cantidad de nodos no se reduce por utilizar un k más grande. Esto es importante pero veamos por qué. En la i-esima iteración, GRASP le pide a Greedy que obtenga una nueva solución tomando siempre, en cada iteración de este último, de los primeros k posibles un nodo random como próximo elemento a ser considerado como dominante, y una vez que el conjunto sea dominante, será mejorado con local search.
Ahora entonces tenemos una nueva solución, independiente de la i-1-esima iteración. Notemos que esta solución no es necesariamente mejor a la anterior, esto se debe a que el elemento obtenido de la RCL puede ser bastante malo según el criterio greedy y por lo tanto se obtiene un conjunto grande, que la búsqueda local no alcanza a mejorar del todo. Por otro lado, si el tamaño de la RCL es muy chico, la solución inicial greedy randomizada va a ser muy similar a la greedy clásico (no randomizado). Esto nos llevaría a explorar una cantidad más acotada de vecindarios de soluciones al momento de aplicar la búsqueda local. Entonces concluimos que al momento de elegir el k, notar que con pocos k ya se puede obtener lo que sería el ideal, pero sin embargo es conveniente utilizar una RCL más grande para abrir el panorama de soluciones.

A continuación se agrega un gráfico para evaluar los tiempos para distintos k con distintos nodos del conjunto de familias detallado en los gráficos anteriores:



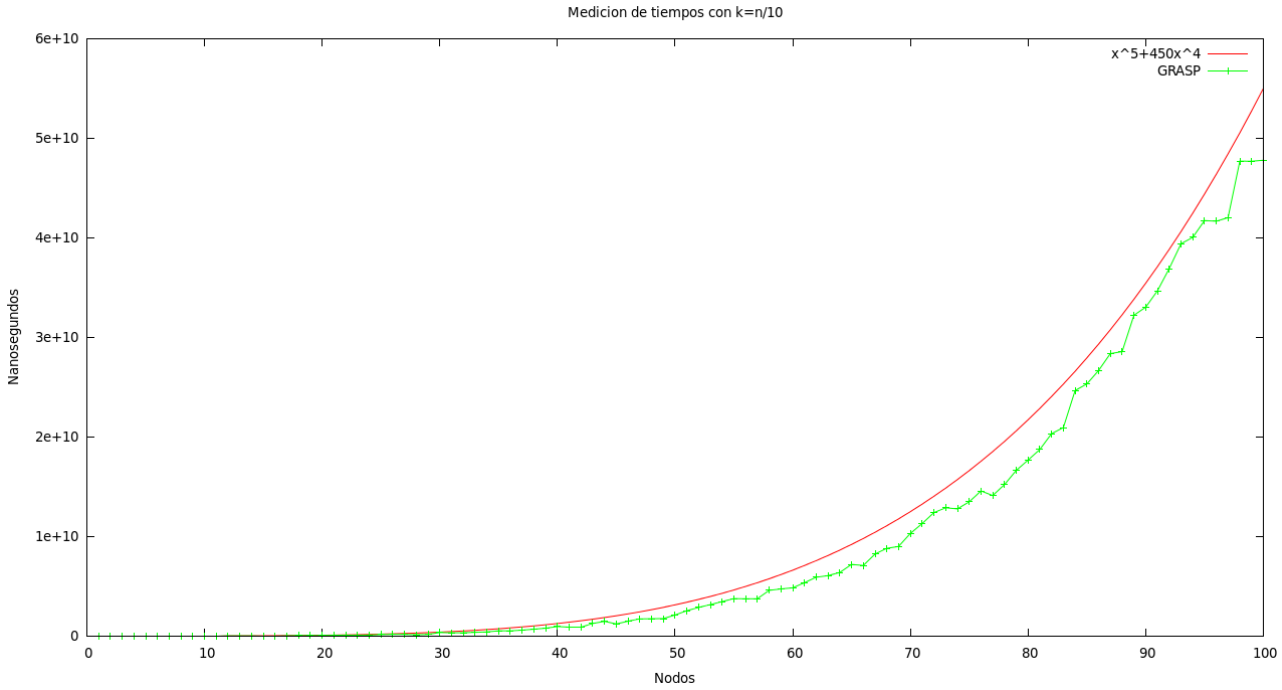

Flia de grafo Mobius perteneciente al peor caso con n=12, n=60, n=168, n=252 con k $<$ 25




Flia de grafo Triangulos Unidos perteneciente al mejor caso con n=12, n=60, n=168, n=252 con $<$ 25

Como era esperable, la relación es lineal. Esto es ya que el K multiplica como constante y esta es siempre menor a n, ya que correr con un k mayor a n no tendría sentido práctico. Las pequeñas diferencias se deben a que al depender de otros algoritmos como ser: Greedy y LocalSearch, que tienen una condición de corte distinta a la de este algoritmo, puede pasar que terminen antes o tarden más. Por ejemplo, en caso de que greedy con un k, elija en tres iteraciones un conjunto dominante minimal y sin embargo para otro k elija en más iteraciones.
Tomamos k chicos, ya que como vimos con k muy grandes no significa mejores, resultados, y de esta forma mostramos un gráfico útil al momento de elegir qué algoritmo correr.

A continuación se agrega un gráfico con los tiempos de corrida para distintos n de la familia Greed comparado con la cota superior:




Flia de grafo Triangulos Unidos perteneciente al mejor caso con n=12, n=60, n=168, n=252 con $< 25$

Como se puede ver, mantenemos la cota mencionada. Las pequeñas diferencias que hacen que no sea completamente paralela, se deben a lo mencionado en el párrafo anterior, ya que estámos trabajando con un grafo random.

# 6. Resultados Finales

## 6.1. Introducción

Realizamos diferentes gráficos generados pseudo-aleatoriamente con el fin de poder obtener una idea generalizada entre los algoritmos, ya que en las secciones anteriores pudimos detallar los peores y mejores casos de cada uno. Estos algoritmos fueron generados con una función que recibe como parámetro la cantidad de nodos, y una probabilidad. Dicha función, por cada par de nodos, genera un valor random entre 0 y 100, si este valor está por debajo de la probabilidad recibida como parámetro, unirá dichos nodos, de otra manera, no.
Los análisis a realizar serán del estilo cantidad de nodos del conjunto dominante otorgado por cada heurística, comparándolo con el Exacto, y tiempos.
Para el GRASP, elegimos correr con k=n/10, ya que después de varias pruebas, nos pareció que con esa cantidad de iteraciones en proporción a la cantidad de nodos encuentra un valor óptimo antes de que el algoritmo entre en lo que denominamos "esperando mejor solución que puede no llegar" (remitirse al análisis del GRASP en la sección anterior).
Cabe destacar que el algoritmo exacto, se deja de correr para los n > 30 debido a su complejidad. Por esto y porque a partir de ese numero pega saltos tan grandes (en cuanto a tiempos de ejecucion y ciclos de procesador) que se vuelve imposible graficarlo.
Para el LocalSearch usamos la estrategia DosPorUno por las razones ya mencionadas en la seccion de ese algoritmo.

## 6.2. Por Cantidad De Nodos

Generar dos graficos con n<200 como eje x y p = 30 (en uno) y p = 50 (en otro)
Generar dos graficos con p<100 como eje x y n = 50 (en uno) y n = 250 (en otro)
Aca hablas que GRASP siempre fue el mejor, que el greedy no se aleja mucho del exacto, y el local search mejora
Elegimos estos gráficos porque son representativos de los casos ejecutados durante el proceso de experimentación vemos que a mayor cantidad de aristas, menor cantidad de nodos en el conjunto dominante Otro aspecto a destacar es que cuanto más denso es el grafo, las heurísticas tienden a funcionar mejor. En forma intuitiva, lo que ocurre es que al aumentar la cantidad de aristas, aumenta la cantidad de soluciones optimas

El algoritmo greedy, al tomar una decisión de qué nodo agregar tiene menos probabilidad de tomar una mala decisión. En el caso de búsqueda local, al aumentar la cantidad de aristas tiene más posibilidades de intercambios de vecinos. Por ultimo, el GRASP mejora al mejorar sus dos componentes que son las heurísticas anteriores.

## 6.3. Por Tiempo

Generar dos graficos con n<200 como eje x y p = 30 (en uno) y p = 50 (en otro)
Generar dos graficos con p<100 como eje x y n = 100 (en uno) y n = 250 (en otro)
Si vemos que el GRASP tarda más que otros es por la cantidad de iteraciones El grasp tarda

Finalmente cabe mencionar que el algoritmo GRASP tarda siempre más que el de búsqueda local y este ultimo más que el greedy. Esto se debe a que el de búsqueda local ejecuta el greedy y luego intenta mejorar la solución, y el GRASP realiza una determinada cantidad de iteraciones en las que ejecuta, en cada una, el greedy y lo mejora con búsqueda local.

# 7. Conclusiones

Heurísticas descentes. Exacto no es factible para n no muy grandes Tenemos 3 heurísitcas, pero que a mayor complejidad, mejores resultados GRASP no necesita muchas iteraciones

Finalmente, para uso práctico podemos recomendar utilizar el algoritmo de búsqueda local por sobre el greedy para obtener una solución razonablemente buena en poco tiempo, ya que la puede mejorar con un costo adicional bastante bajo. Para problemas en los que se necesite mayor precisión, se puede aplicar GRASP variando los parámetros dependiendo del tiempo disponible y teniendo en cuenta lo mencionado anteriormente.