DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 3

Informe y análisis de resultados.

Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Benitti, Raul	592/08	raulbenitti@gmail.com
Scarpino, Gino	392/08	gino.scarpino@gmail.com
Vallejo, Nicolás	500/10	nico_pr08@hotmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Descripción de situaciones reales

Para ciertos tipos de problemas que se suelen modelar con grafos, se usa coloreo de grafos. Estos problemas, en general, suelen ser donde para ciertas cosas chocan interesas o hay incompatibilidades. Modelandolo, un nodo que representa un objeto está relacionado a otro, es decir, son adyacentes, si pasa lo anteriormente dicho. Coloreando el grafo, los nodos vecinos tienen distinto color, donde el color representa algo oportuno.

En el caso de este problema, se aplica cuando uno modela un problema con coloreo para el grafo G, cambian las relaciones y se obtiene un grafo H. Se quiere averiguar cómo impacta el cambio, buscando el máximo valor posible. Sirve, de cierta forma, para medir esto.

2. Algoritmo Exacto

2.1. Algoritmo

La idea de este algoritmo es simple. Recorremos la cantidad de colores posibles del grafo g, calculamos su impacto en h y nos quedamos con el máximo.

La dificultad del problema radicaba en calcular la cantidad de colores posibles del grafo, algo que a priori puede ser infinito.

Consideramos a los colores como números enteros positivos. De aquí se extrae que hay infinitos colores y por lo tanto infinitos coloreos de G válidos.

Sin embargo, se pueden hacer algunas consideraciones para acotar considerablemente la cantidad de colores a usar, sin perder generalidad.

En primer lugar, intuitivamente se extrae que con n colores es suficiente, donde n es la cantidad de nodos para analizar todos los casos posibles. Entonces consideraremos como colores los primeros n números enteros positivos. Esto es porque en realidad este caso sirve para analizar aquellos donde se tienen n colores distintos, independientemente de que esos n colores no sean los primeros n números positivos (considerando un color como un número).

Sin embargo, todavía se puede acotar un poco más las instancias a analizar.

Esto tiene que ver con el hecho de que, aunque definamos n colores, uno podría establecer una biyección entre dos conjuntos de colores con los mismos elementos dónde, por ejemplo, en el conjunto 2 se renombra al color 1 del primer conjunto como el color n y viceversa. Esto provocaría, de no tenerse en cuenta, que se estén analizando casos ya analizados. Si además, en vez de renombrar 2 colores los renombrasemos a todos, estaríamos analizando muchísimos casos que, si bien se corresponden a coloreos distintos, son en el fondo, casos equivalentes.

Para evitarnos estos análisis de más, determinamos los coloreos de la siguiente manera:

Eligimos un nodo cualquiera como el primer nodo para pintar. A ese nodo lo pintamos con el color 1, y por lo que mencionamos anteriormente, de esta manera se estarían analizando también los casos dónde ese nodo tiene otro color entre 2 y n.

Ahora se elije otro nodo para pintar en segundo lugar. Para pintarlo tenemos dos posibilidades: o pintarlo del mismo color que el nodo 1 o pintarlo con un color nuevo. Luego, se están generando dos coloreos distintos, y cada uno se sigue generando por separado.

La idea es que al pintar el nodo k, con $k <= n$, uno podría elegir 2 caminos : o pintarlo de un color ya usado, o pintarlo con un color nuevo.

Definamos max como el color de máximo valor utilizado hasta ahora en el coloreo. Luego, existen $max + 1$ maneras de continuar ese coloreo, pintando el nodo k de 1,2...max o de un color nuevo, $max + 1$.

Debe notarse entonces, que por cada paso del coloreo se generan muchos nuevos coloreos más.

A medida que se generan los coloreos, generamos otra poda: aquella que determina si al pintar a un nodo de un color se está generando un coloreo válido. En caso de no serlo se deja de analizar ese caso.

Luego, de todos los colores posibles válidos se calcula el impacto en H y nos

quedamos con aquél que proporciona un máximo impacto.

El pseudocódigo del algoritmo es el siguiente:

Algoritmo 2.1

```
MAXIMOIMPACTOEXACTO(Grafo g, Grafo h)
1  vector<unsigned int> res(n + 1)
2  int res[0] = 0
3  vector<unsigned int> coloreo(n, 0)
4  coloreo[0] = 1
5  colorear(1, g, h, coloreo, res)
6  return res
```

Algoritmo 2.2

```
COLOREAR(unsigned int nodo, Grafo g, Grafo h,vector<unsigned int>
coloreo,vector<unsigned int> solucion)
 1  if (nodo > n)
 2      int temp = impacto(h, coloreo)
 3      if temp > solucion[0]
 4          solucion[0] = temp
 5          for i desde 0 hasta n
 6              solucion[i + 1] = coloreo[i]
 7  else
 8      int maxColor = maximo color usado hasta el momento
 9      for c desde 1 hasta maxColor
10          if es legal pintar el nodo nodo del color c
11              vector<unsigned int> nuevoColoreo(coloreo)
12              nuevoColoreo[nodo] = c
13              colorear(nodo + 1, g, h, nuevoColoreo, solucion)
```

2.2. Análisis de complejidad

Al momento de hacer este análisis de complejidad se tuvieron en cuenta algunas consideraciones.

En primer lugar, llamaremos m al máximo entre la cantidad de aristas del grafo G y del grafo H y n a la cantidad de nodos de dichos grafos.

En segundo lugar, en el análisis de complejidad de la función *colorear* se definirá k como la cantidad de nodos que quedan por pintar hasta ese paso de la recursión, en

contraste con la implementación donde la recursión es , por así decirlo *haciaarriba*, significando esto que se inicia desde el primer nodo y se va hacia el último.

Analicemos primero la función *colorear*. Como mencionamos anteriormente, consideraremos la recursión en la cantidad de nodos que quedan por colorear.

El caso base será cuando no hayan más nodos por pintar. En dicho caso, el algoritmo simplemente calcula el impacto de dicho coloreo en el grafo H y en caso de ser el de máximo impacto hasta el momento se reemplaza la solución anterior por la nueva. Esto cuesta O(n+m), que es lo que cuesta calcular el impacto en H.

Para el caso en el que la cantidad de nodos a pintar sea distinta de cero, el algoritmo calcula los posibles colores con los que pintar el nodo. Esto lo hace buscando cuál es el máximo color usado hasta el momento. El nuevo nodo podrá ser pintado de los colores usados anteriormente o del máximo color usado hasta el momento + 1, es decir pintándolo de un nuevo color. Esto se calcula en tiempo O(n).Luego se llamará a la función recursivamente una cantidad de veces igual al máximo color a utilizar. Ese valor se puede acotar para todos los casos por n-k+1.

Además se chequea que agregar ese color genere un coloreo válido, y eso cuesta O(cantidad de vecinos del nodo), que lo podemos acotar por la cantidad de aristas de G, es decir O(m). Luego se hace la llamada recursiva para la instancia inmeditamente menor.

Pasando en limpio, en el paso k el algoritmo cuesta (n-k+1)*(n+m + T(k-1)).
Es decir:

T(0)=n+m
T(k)=(n-k+1)*[(n+m)+ T(k-1)]

donde k es la cantidad de nodos que quedan por pintar.

Veamos entonces cuánto cuesta pintar todos los nodos.

Basado en la definición que dimos antes, si tenemos que pintar todos los nodos, estamos en el caso T(n).

T(n)=(n-n+1)*(n+m + T(n-1))

Desarrollemos T(n):

$$\begin{gathered}
\text{T(n)=(n-n+1)*(n+m + T(n-1))=} \\
\text{= n+m + [2*(n+m) + 2*T(n-2)]=} \\
\text{=(n+m)+2*(n+m) + 2*[3*(n+m)+ 3* T(n-3)]=} \\
\text{=(n+m)+2*(n+m)+ 2*3*(n+m) + 2*3 T(n-3)=} \\
\text{=(n+m)+2*(n+m)+ 2*3*(n+m) + 2*3*[4*(n+m)+4*T(n-4)]=} \\
\text{=(n+m)+2*(n+m)+2*3*(n+m)+2*3*4*(n+m)+ 2*3*4*T(n-4)=} \\
\text{=...=} \\
\text{=[} \textstyle\sum_{i=1}^{n} i! * (n+m)] \text{ + n! * T(0)=}
\end{gathered}$$

$$=[\sum_{i=1}^{n} i! * (n+m)] + \text{n! * (n+m)}$$

Es decir que T(n)= $[\sum_{i=1}^{n} i! * (n+m)]$ + n! * (n+m)

Conjeturamos entonces que T(n) es $O(\sum_{i=1}^{n} i! * (n+m))$.

Veámoslo por inducción en la cantidad de nodos por pintar:

Queremos ver que existe un d real positivo y n_0 natural positivo tales que para todo $n \geq n_0$ vale que $T(n) \leq d * [\sum_{i=1}^{n} i! * (n+m)]$.

Caso base: n=1

$$\text{T(1)}= \sum_{i=1}^{1} i! * (1+m) 1! * (1+m) = \text{2*1!*(1+m)} =$$
$$=\text{2*(1+m)} \leq \text{2*} \sum_{i=1}^{1} i! * (1+m)$$

Es decir que con un $d = 2$ nos alcanza.

Paso inductivo: Suponiendo que vale que $T(n-1) \leq d * [\sum_{i=1}^{n-1} i! * (n-1+m)]$ quiero ver que vale $T(n) \leq d * [\sum_{i=1}^{n} i! * (n+m)]$

$$\text{T(n)=(n-n+1)*(n+m + T(n-1))=}$$
$$= \text{(n+m) + T(n-1)} \leq$$
$$\leq \text{por hipótesis inductiva} \leq$$
$$\leq n + m + d * [\sum_{i=1}^{n-1} i! * (n-1+m)] \leq$$
$$\leq n + m + d * [\sum_{i=1}^{n-1} i! * (n+m)] \leq$$
$$\leq (n+m) * [(d * \sum_{i=1}^{n-1} i!) + 1] \leq$$
$$\leq (n+m) * [(d * \sum_{i=1}^{n-1} i!) + n!] \leq$$
$$\leq (n+m) * [(d * \sum_{i=1}^{n} i!)] \leq$$
$$\leq d * \sum_{i=1}^{n} i! * (n+m)$$

que es lo que queríamos ver.

Luego, colorear cuesta $O(\sum_{i=1}^{n} i! * (n+m))$.

maximoImpactoExacto cuesta entonces $O(\text{n+1 + n + 1} + \sum_{i=1}^{n} i! * (n+m))$ que es $O(\sum_{i=1}^{n} i! * (n+m))$.

2.3. Experimentación y Resultados

Trabajamos con los siguientes 3 casos: grafos al azar, grafos densos, G y H complementos.

Se midieron los tiempos en corridas de 5 a 100 nodos con 100 repeticiones para cada cantidad de nodos.

2.3.1. Grafos al azar

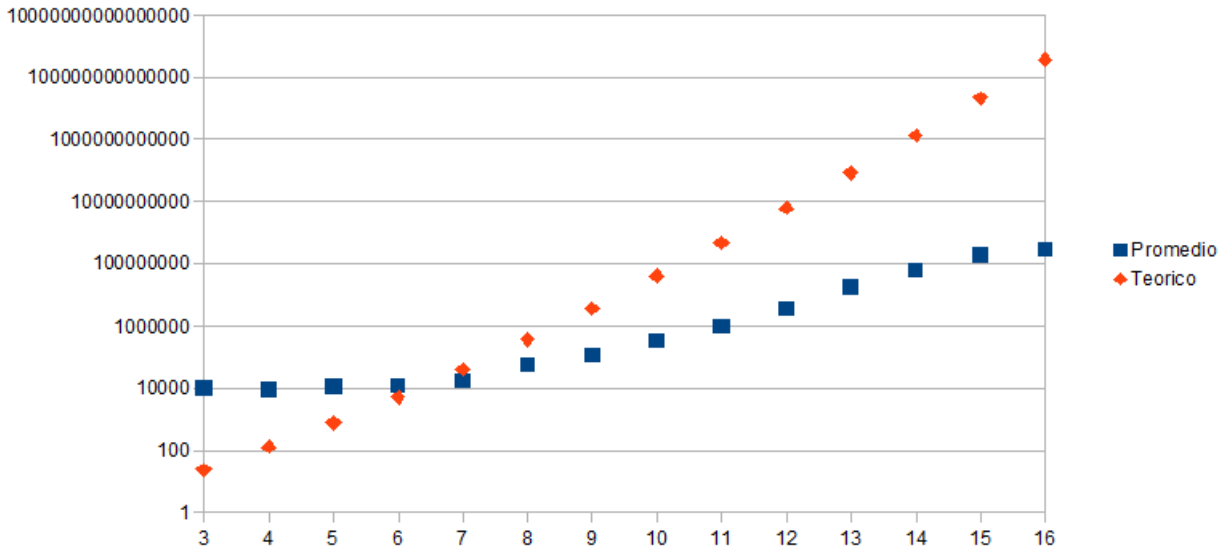

Figura 1: Costos

2.3.2. Grafos densos

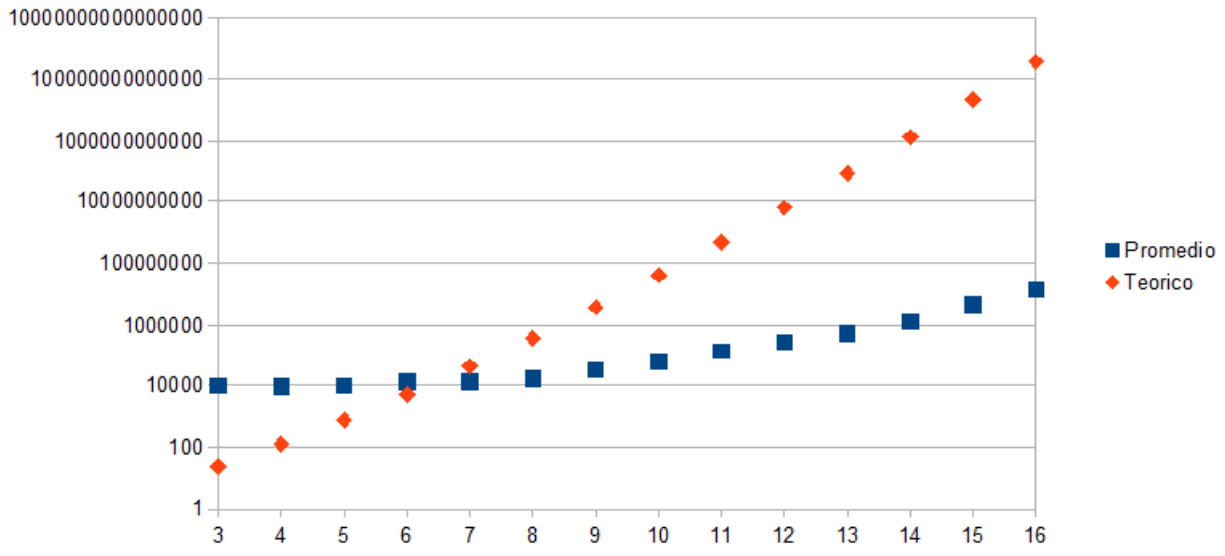

Figura 2: Costos

2.3.3. G y H complementos

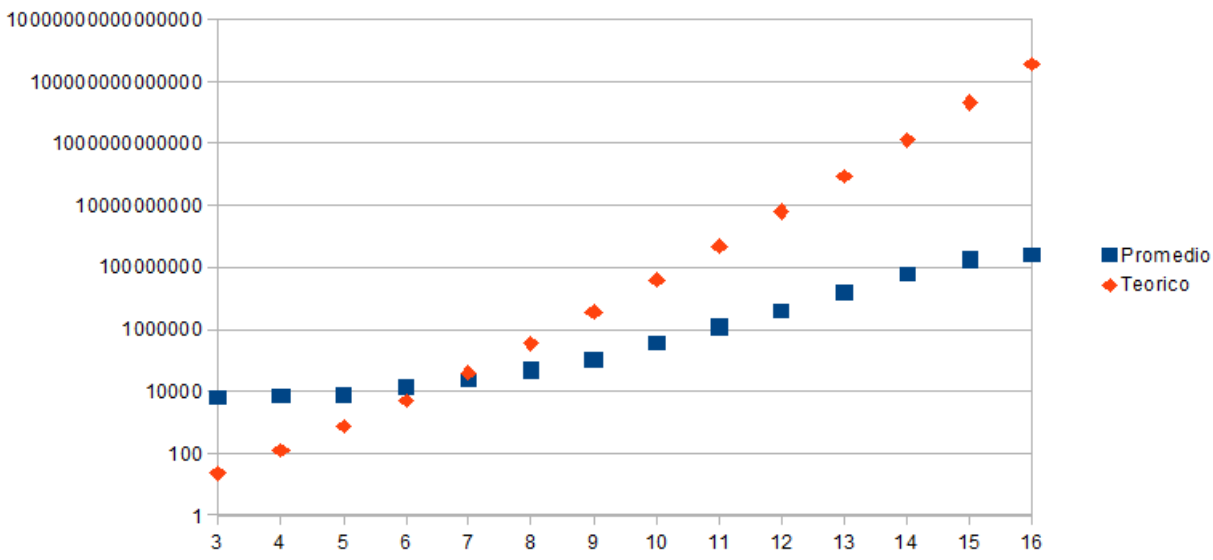

Figura 3: Costos

3. Heurística Constructiva Golosa

3.1. Algoritmo

Analizando las características del problema implementamos una heurística constructiva golosa con los siguientes objetivos:

- Empezar con el impacto máximo posible (cantidad de aristas de H)

- Modificar la menor cantidad de nodos posibles

- El nodo que se modifique, tendría que bajar lo menos posible el impacto

Pseudocódigo:

Algoritmo 3.1

```
MAXIMOIMPACTOGOLOSO(Grafo g, Grafo h, doubleporcentaje)
 1  vector<unsigned int> res(n + 1)
 2  int solucion[0] = 0
 3  vector<unsigned int> coloreo(n, 1)
 4  vector<bool> modificados(n, false)
 5  while not G.coloreoLegal(coloreo))
 6        nodo = siguienteModificable(G,H,modificados,porcentaje)
 7        for c desde 1 hasta colores.size()
 8              if G.colorLegalDelNodo(nodo,coloreo,c)
 9              coloreo[nodo] = c
10              exitFor
11  solucion[0] = impacto(h, coloreo)
12  for i desde 0 hasta n
13  solucion[i + 1] = coloreo[i]
14  return solucion
```

Algoritmo 3.2

siguienteModificable($Grafo$ g, $Grafo$ h,vector<bool> modificados, double porcentaje)

```
1  vector<pair< unsigned int, unsigned int > > posibles
2  for n nodo in V(G)
3        if not modificados[ nodo ]
4        agregar(posibles,<G.impactoNodo(nodo,H,coloreo),nodo>)
5
6  sort(posibles)
7
8  unsigned int res = random(| posibles | * porcentaje)
9
10 return res
```

En *maximoImpactoGoloso* primero se colorea a todos los nodos con el mismo color, por lo que, sin tener en cuenta al grafo G, el impacto en H es máximo. Mientras que no sea legal este coloreo en G, se va a ir modificando los nodos. Apenas se consiga un coloreo legar se detiene. Por lo tanto, no necesariamente el ciclo principal itera la cantidad de nodos (en el peor caso sí).

En *siguienteModificable* se obtiene el nodo a modificar. Se crea una lista de nodos que no hayan sido modificados y se calcula para cada uno su aporte al impacto en H. Se ordena la lista por los siguientes criterios:

- Menor impacto en H aportado: se desea modificar al que menos aporta para ver si cambiando de color aporta más.

- Mayor grado en G: se busca que cuanto antes el coloreo de G sea legal modificando lo menos posible, y modificando un nodo de grado grande se podria conseguir esto.

- Menor grado en H: por lo que dijimos, en caso de empate de los anteriores criterios, queremos modificar lo menos posible en G y H

3.2. Análisis de complejidad

Comencemos analizando la complejidad de la función impactoNodo.

Esta función mira para un nodo el impacto que aporta en H, comparando su color con el de sus vecinos. Dicho nodo tiene en H a lo sumo n-1 vecinos. Luego, impactoNodo cuesta O(n).

Analicemos ahora siguienteModificable. Al inicio comienza iterando sobre la can-

tidad de nodos de H y si dicho nodo no fue modificado o si no tiene vecinos, se calcula el impacto de cada nodo y se lo agrega a un vector de nodos candidatos a ser modificados.En el peor caso, todos los nodos están sin modificar y tienen vecinos,por lo tanto esto cuesta O(n^2).

Luego, se ordena de manera creciente el vector de candidatos de acuerdo al impacto de cada nodo.En el peor caso dicho vector tiene n elementos, pues todos los nodos son modificables y ordenarlos cuesta entonces O(n*log(n)).

Luego, se itera sobre la cantidad de elementos de ese vector, esta vez para desempatar los nodos. En el peor caso todos los nodos empatan en el impacto que generan. Desempatarlos a todos cuesta en el peor caso O(n^2) , que el caso en el que se invirtió el orden del vector por desempates.

A continuación se elige pseudoaleatoriamente en O(1) uno de los primeros elementos del vector.

Pasando en limpio, siguienteModificable cuesta O(n^2 + n*log(n)+ n^2), que es O(n^2).

Ahora analicemos maximoImpactoGoloso.

Al principio realiza unas cuantas operaciones en O(n). De estas es destacable la creación de un vector de tamaño igual al grado del nodo con grado máximo de G, que refiere a la cantidad de colores a usar. Pero el grado máximo de cada nodo es a lo sumo n-1. Luego crear ese vector cuesta O(n).

Luego, se ejecuta un while que a lo sumo itera n veces. Esto es porque en el peor caso tuve que pintar todos los nodos de distinto color hasta obtener un coloreo válido.

Dentro de ese while está implícito el chequeo de si el coloreo es válido, que cuesta O(n+m), donde vamos a acotar a m como el máximo entre las aristas de G y de H. Se ejecuta siguienteModificable y se itera luego en la cantidad de colores, costando cada iteración en la cantidad de colores O(n) que es lo que cuesta ver si pintar un nodo de ese color es no coincide con el color de uno de los vecinos de ese nodo, que como mencionamos antes pueden ser n-1.

Luego, lo de adentro del while cuesta O(n+m+ n^2) y el costo total del while es de O(n(n+m +n^2)), que es O(n*(n+m)+ n^3).

Luego de iterar se calcula el impacto de dicho coloreo en O(n+m).

Es decir que en total maximoImpactoGoloso cuesta O(n+n*(n+m)+ n^3 + n+m).

Por lo tanto, maximoImpactoGoloso cuesta O(n*(n+m)+ n^3).

3.3. Experimentación y Resultados

Trabajamos con los siguientes 3 casos: grafos al azar, grafos densos, G y H complementos.

Se midieron los tiempos en corridas de 5 a 100 nodos con 100 repeticiones para cada cantidad de nodos.

3.3.1. Grafos al azar

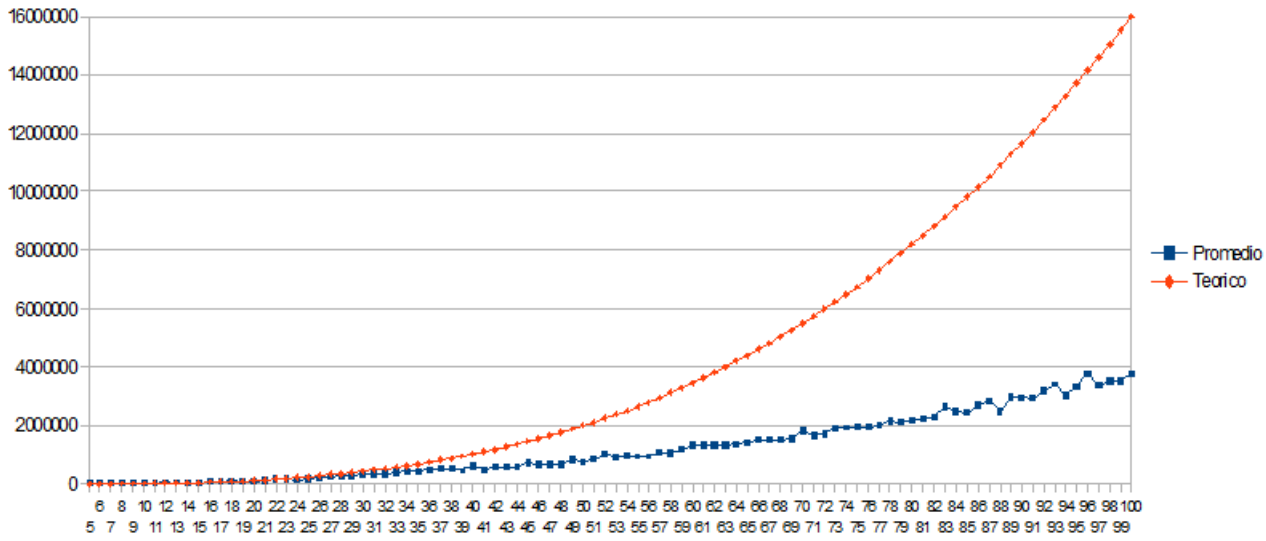

Figura 4: Costos

3.3.2. Grafos densos

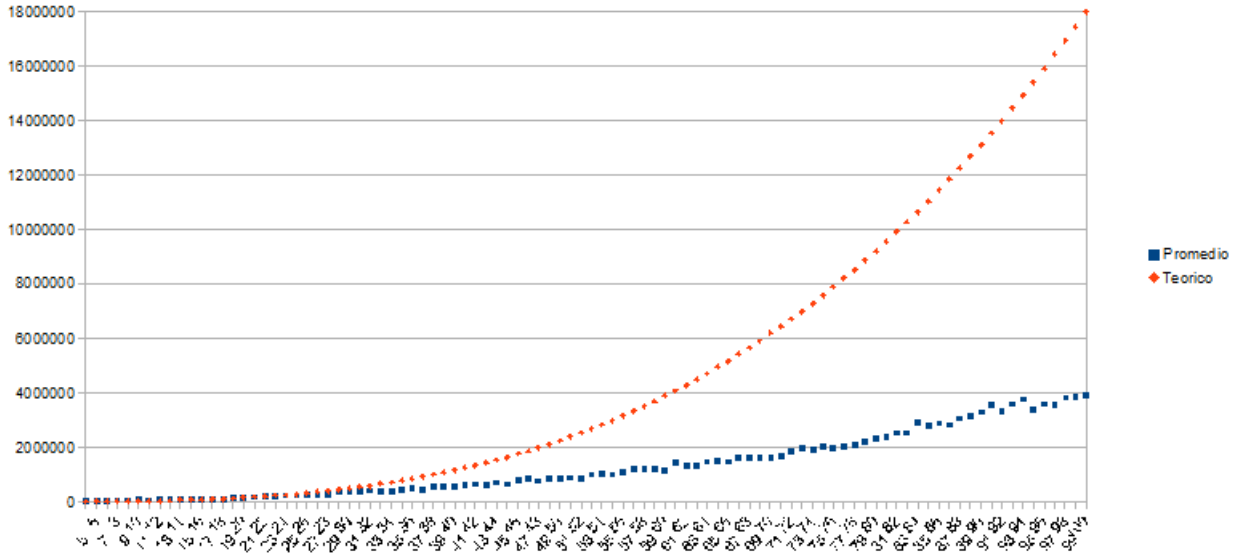

Figura 5: Costos

3.3.3. G y H complementos

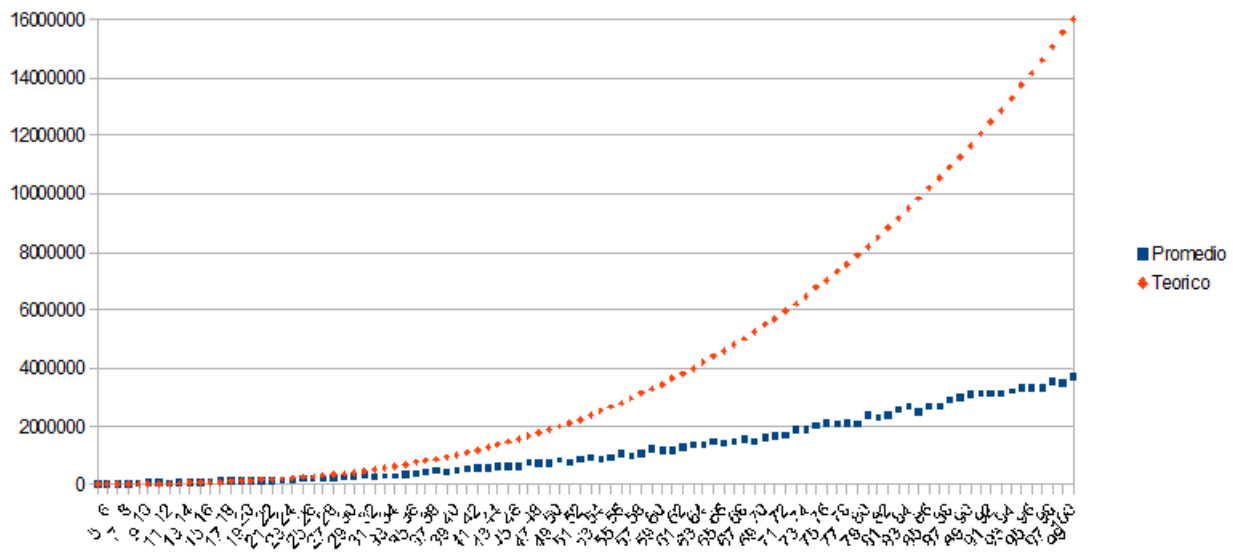

Figura 6: Costos

4. Heurística de Búsqueda Local

4.1. Algoritmo

El algoritmo de búsqueda local que implementamos parte de una solución obtenidad por el algoritmo contructivo goloso aleatorio que desarrollamos y detallamos en la sección anterior de este Trabajo Práctico. Se recibe un parámetro que será el valor *porcentaje*, utilizado a la hora de aplicar el algoritmo goloso como solución inicial.

Una vez obtenida esta solución base, se procede a iterar explorando el espacio de las soluciones vecinas.

Para ello, fue menester definir la vecindad entre soluciones. Definimos la vecindad de una solución en el contexto del algoritmo de búsqueda local como aquellos coloreos válidos de G que se obtienen de modificar el color de algún nodo de dicha solución por el color de alguno de los vecinos de ese nodo en el grafo H. Resultó natural definir la vecindad de esta manera pensando en el significado de Impacto de un coloreo de G en H, dado que lo que buscamos es que para un coloreo válido de G la mayor cantidad de nodos vecinos entre sí en H estén coloreados del mismo color.

Una vez hechos estos comentarios, nos abocamos a describir el funcionamiento de nuestra implementación del algoritmo de búsqueda local para resolver el problema del coloreo de máximo impacto de G en H.

Como mencionamos anteriormente, partimos de una solución obtenida mediante nuestra implementación del algoritmo constructivo goloso, ejecutado con el parámetro *porcentaje* que está búsqueda local recibe por parámetro. Una vez obtenida esa solución, se comienza a recorrer sus vecinas.

Para ello, se comienza a iterar en los nodos de H. Por cada nodo, se recorren sus vecinos en H y se analiza si se mejora el impacto cambiándole el color a ese nodo por el de alguno de sus vecinos (siempre y cuando dicho coloreo sea válido en G). En caso de que se obtuviese un mejor impacto, se actualiza la solución con el cambio propuesto y se continúa iterando en los vecinos de ese nodo, esta vez comparando la solución parcial con la nueva solución obtenida. Una vez recorridos todos los vecinos de ese nodo en H, se pasa a otro nodo y se repite el proceso, actualizando la solución en caso de ser necesario con los criterios mencionados.

El resultado es entonces una solución que se obtuvo de ir modificando la primera solución obtenida con el algoritmo goloso. Por la manera en que fuimos operando, podemos estar seguros de que si la solución final es diferente a la solución inicial de la que se partió, entonces el impacto del tal coloreo de G en H es mayor al de la solución golosa. En el caso en que ninguna de las soluciones vecinas de la solución golosa mejore el impacto se devuelve entonces la solución inicial.

Algoritmo 4.1

```
maximoImpactoLocal(Grafo g, Grafo h, double porcentaje)
 1
 2  vector<unsigned int> impactoGoloso = maximoImpactoGoloso(g,h,porcentaje)
 3  unsigned int impactoParcial = impactoGoloso[0];
 4  vector<unsigned int> coloreo(n);
 5
 6  vector<unsigned int> solucionFinal(n+1);
 7
 8  for i desde 1 hasta n
 9        coloreo[i] = impactoGoloso[i]
10
11  unsigned int nuevoImpacto = 0
12
13  for i desde 1 hasta n
14
15        vector<unsigned int> vecinos = vecinos del nodo i en h
16
17        for j desde 1 hasta la cantidad de vecinos de i en h
18              unsigned int color = coloreo[vecinos[j]]
19
20              if pintar al nodo i de color es legal
21                    vector<unsigned int> nuevoColoreo =    coloreo
22                    nuevoColoreo[i]=    color
23                    nuevoImpacto =    h.impacto(nuevoColoreo)
24
25                    if nuevoImpacto > impactoParcial
26                          coloreo[i] =    color
27                          impactoParcial =    nuevoImpacto
28
29
30  solucionFinal[0] =    impactoParcial
31
32  for i desde 0 hasta n
33        solucionFinal[i+1]=coloreo[i]
34
35  return solucionFinal
```

4.2. Análisis de complejidad

Veamos la complejidad de maximoImpactoLocal. Primero se calcula una solución con maximoImpactoGoloso. Como mencionamos en el apartado correspondiente eso cuesta O(n*(n+m)+ n^3).

Luego, se copia el coloreo que se obtuvo de maximoImpactoGoloso en O(n).

A continuación se itera sobre la cantidad de nodos de H. En cada iteración se copian los vecinos del nodo en el que estamos ahora. Como dicho nodo puede tener a lo sumo n-1 vecinos, eso cuesta O(n). Luego,se itera sobre los vecinos del nodo y por cada vecino del nodo se decide en O(n+m) si se va a pintar el nodo del mismo color que su vecino. Dicha decisión se fundamenta en si cambiar el color genera un coloreo válido en G y si aumenta el impacto H. Entonces,el costo del ciclo interior cuesta O(n*(n+m)). Por lo tanto, el ciclo que lo engloba cuesta O(n^2*(n+m)).

Luego de terminar de iterar se guarda el coloreo parcial con un costo de O(n).

Entonces, maximoImpactoLocal cuesta O(n*(n+m)+ n^3 +n^2*(n+m)).

4.3. Experimentación y Resultados

Trabajamos con los siguientes 3 casos: grafos al azar, grafos densos, G y H complementos.

Se midieron los tiempos en corridas de 5 a 100 nodos con 100 repeticiones para cada cantidad de nodos.

4.3.1. Grafos al azar

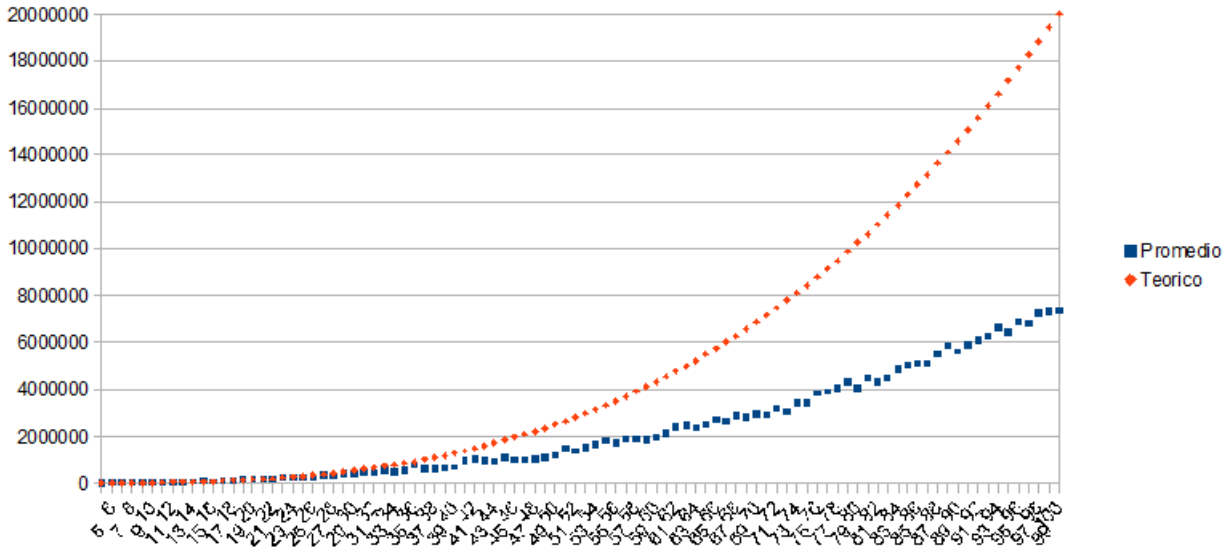

Figura 7: Costos

4.3.2. Grafos densos

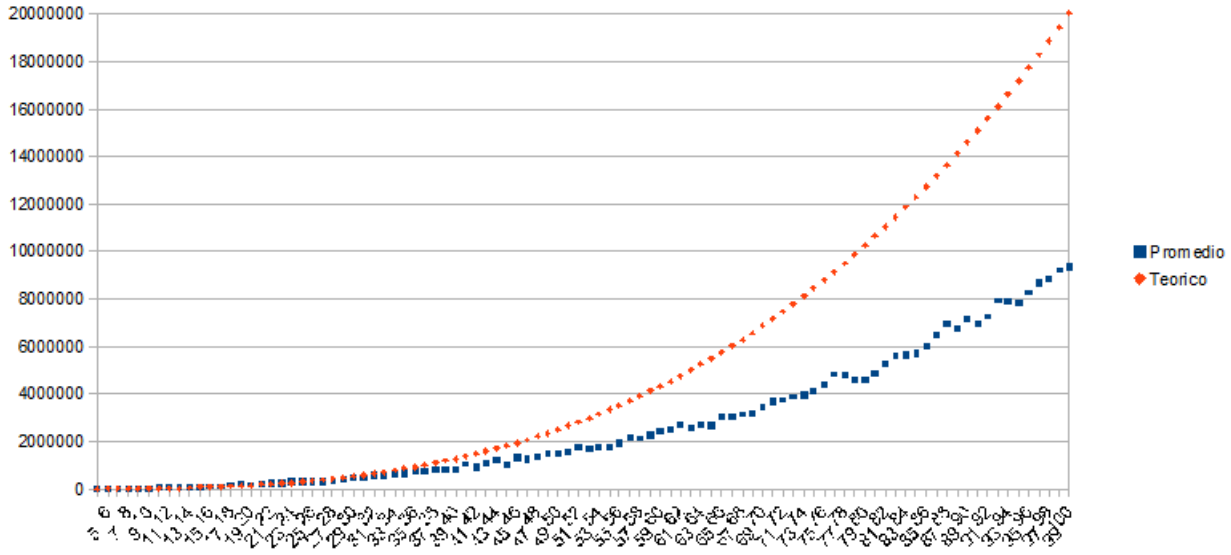

Figura 8: Costos

4.3.3. G y H complementos

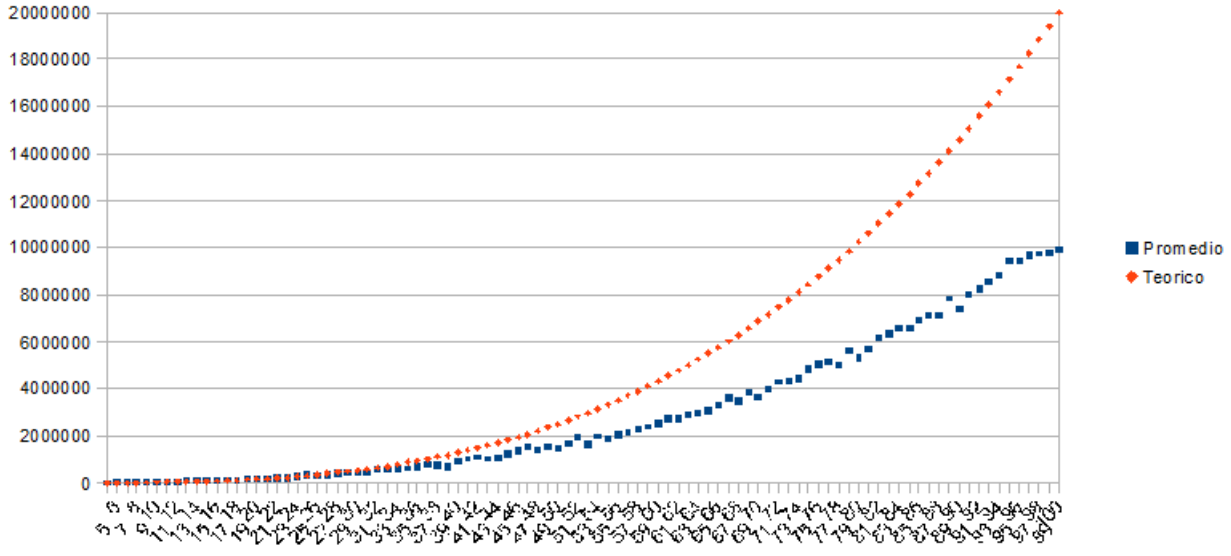

Figura 9: Costos

5. Metaheurística de Grasp

5.1. Algoritmo

Nuestra implementación de Grasp opera de la siguiente manera: Se generan una cantidad de veces determinada por parámetro de soluciones con nuestra implementación de la heurística golosa. Luego se elige una de ellas pseudoaleatoriamente y se le aplica nuestra implementación de búsqueda local. Si esa solución obtenida con búsqueda locar mejora la que teníamos anteriormente, nos quedamos con ella. Este proceso se iterará una cantidad de veces máxima determinada por parámetro. Sin embargo, puede ocurrir que se deje de iterar antes, si no se encontraron mejores en una cantidad determinada de iteración, también provista por parámetro. A continuación detallamos más detenidamente nuestra implementación.

El comportamiento de nuestra implementación de Grasp se ve fuertemente influenciado por ciertos parámetros. A saber, estos son: $porcentaje$, que se usará para obtener las soluciones con nuestras implementaciones del algoritmo goloso y de búsqueda local; $maxRCL$, que determinará la cantidad de candidatos obtenidos mediante la aplicación del algoritmo goloso; $maxIteraciones$ que determinará la cantidad de veces máxima que iterará el algoritmo en el peor caso; y finalmente $maxIterSinMejora$ que determinará la máxima cantidad de iteraciones que permitiremos ejecutarse sin que se obtenga una mejor solución antes de dejar de iterar (donde una mejor solución es aquella que mejora el impacto del coloreo de G en H).

Como se puede observar, los últimos dos parámetros descriptos se utilizarán como criterios de parada: a lo sumo el algoritmo iterará $maxIteraciones$ de veces en busca de soluciones, a menos que en una cantidad de iteraciones consecutivas igual a $maxIteracionesSinMejora$ no se obtenga una solución que implique un mayor impacto del coloreo de G con esa solución en H.

El algoritmo iterará, en el peor caso, hasta la máxima cantidad de iteraciones permitidas. En cada iteración, se calculan $maxRCL$ (RCL proviene de Restrictive Candidate List) soluciones con el algoritmo goloso que implementamos, cada una de ellas usando el valor de $porcentaje$, es decir que se calculan $maxRCL$ candidatos golosos a los cuales se les puede aplicar el algoritmo de búsqueda local que diseñamos. De esos candidatos se elige uno pseudoaleatoriamente (en nuestro caso, al implementarlo en C++, hicimos uso de la función rand()). A ese candidato elegido, se le aplica nuestra implementación de búsqueda local, obteniendo así una nueva solución. Si esta solución mejora el impacto de G en H en comparación con la solución que se manejaba hasta el momento, se reemplaza a esa solución vieja por esta nueva y se resetea un contador que contiene la cantidad de veces que se iteró sin lograr una mejora. Caso contrario se aumenta dicho contador y se chequea que no se haya alcanzado la máxima cantidad de iteraciones sin mejoras permitidas, en cuyo caso se dejará de iterar y se devolverá la mejor solución que se obtuvo hasta el momento.

Este procedimiento se ejecutará hasta que se cumpla con algunos de los criterios de parada. Cuando uno de ellos se cumpla se devuelve la mejor solución que se obtuvo hasta el momento. Es de notar que en cada iteración se generan $maxRCL$ candidatos

golosos nuevos de los cuáles se elige uno para continuar con la búsqueda local.

La aleatorización de los candidatos golosos se obtiene mediante una combinación de los parámetros *porcentaje* y *maxRCL*. El primer parámetro impacta en la solución que devuelve el algoritmo goloso que implementamos como describimos en la sección correspondiente. El segundo parámetro determina la cantidad de soluciones candidatas que se calcularán en un principio. Luego, además, de esos *maxRCL* candidatos se elegirá uno pseudoaleatoriamente (como mencionamos antes, en nuestra implementación usamos la función rand() de C++).

Algoritmo 5.1

MAXIMOIMPACTOGRASP(*Grafo* g, *Grafo* h, *double* porcentaje, *unsigned int* maxIteraciones, *unsigned int* maxIterSinMejora,*unsigned int* maxRCL)

```
 1  vector<unsigned int> res(n + 1)
 2  res[0] = 0
 3  unsigned int sinMejora = 0
 4
 5  vector<unsigned int> coloreo(n,1) // Todos los elementos valen 1
 6
 7  for i desde 0 hasta maxIteraciones
 8       vector<vector<unsigned int>> rcl(maxRCL)
 9       for k desde 0 hasta maxRCL
10            rcl[k] = maximoImpactoGoloso(g, h, porcentaje)
11
12       unsigned int e = índice de uno de los elementos de rcl elegido al azar
13
14       vector<unsigned int> solBusqLocal = maximoImpactoLocal(g,h,porcentaje,rcl[e])
15
16       if solBusqLocal[0]>res[0]
17            res[0] =solBusqLocal[0]
18
19            for k desde 1 hasta n
20                 res[k]=solBusqLocal[k]
21
22            sinMejora= 0
23       else
24                 sinMejora++;
25                 if sinMejora == maxIterSinMejora
26                      salir del ciclo
27
28       return res
```

5.2. Análisis de complejidad

Analicemos la complejidad de maximoImpactoGrasp. Los primeros pasos del algoritmos son crear unos vectores igual a la cantidad de nodos de los grafos. Eso cuesta O(n) para cada creación de vector.

Luego se itera maxIteraciones veces. El costo de cada iteración es el siguiente:

Primero se calcular maxRCL veces soluciones con maximoImpactoGoloso, donde maxRCL la cantidad de restrictive candidates list, es decir la cantidad máxima de candidatos golosos a utilizar.Ese ciclo cuesta entonces O(maxRCL*(n*(n+m)+ n^3) de acuerdo a nuestro análisis de complejidad de maximoImpactoGoloso.

A continuación, se elige pseudoaleatoriamente uno de esos candidatos.

Una vez elegido un candidato, se aplica maximoImpactoLocal con dicha solución golosa.

Por lo que analizamos en la sección correspondiente, esto cuesta O(n*(n+m)+ n^3 $+n^2$*(n+m)).

Una vez hecho esto, se decide si se va a quedar con la nueva solución obtenida con maximoImpactoLocal y esto cuesta O(n).

Luego, el ciclo cuesta :

O(maxIteraciones* [(maxRCL*(n*(n+m)+ n^3))+ (n*(n+m)+ n^3 $+n^2$*(n+m))])

que además es la complejidad de maximoImpactoGrasp.

5.3. Experimentación y Resultados

Trabajamos con los siguientes 3 casos: grafos al azar, grafos densos, G y H complementos.

Se midieron los tiempos en corridas de 5 a 100 nodos con 100 repeticiones para cada cantidad de nodos.

5.3.1. Grafos al azar

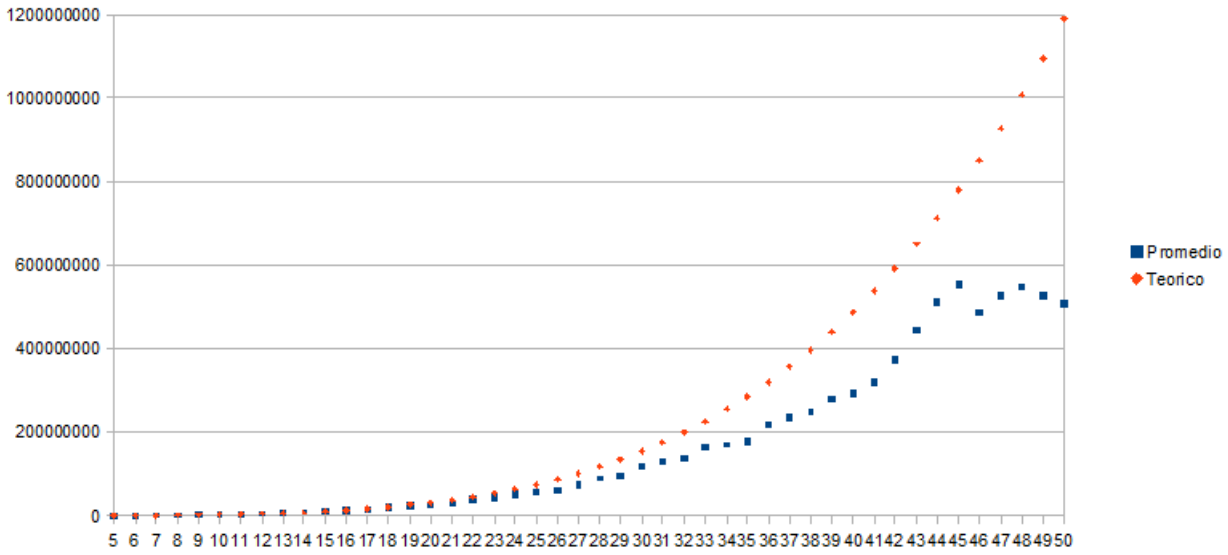

Figura 10: Costos

5.3.2. Grafos densos

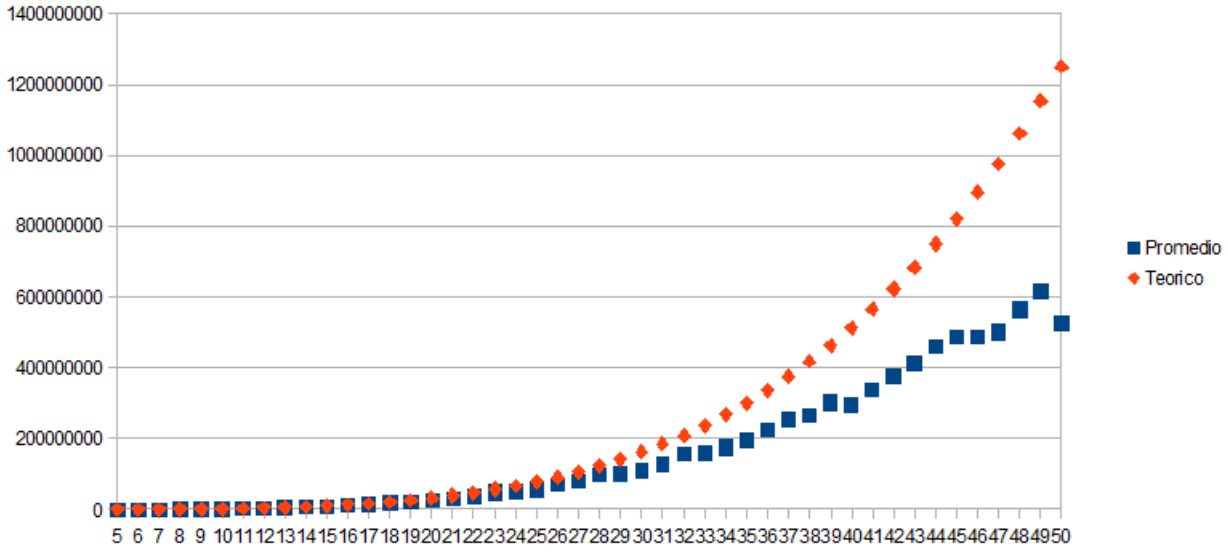

Figura 11: Costos

5.3.3. G y H complementos

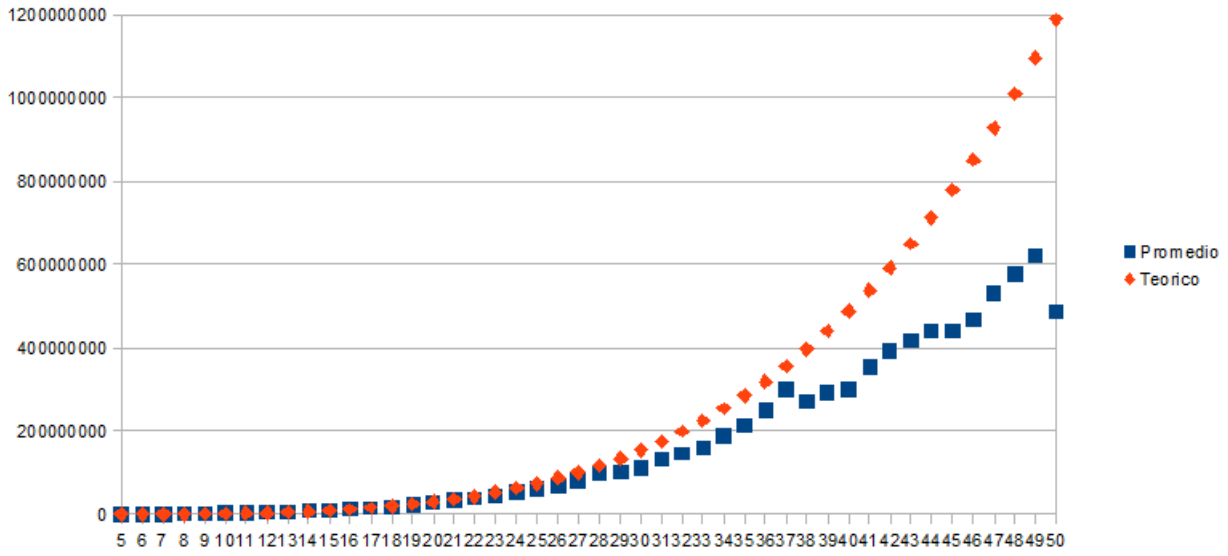

Figura 12: Costos

6. Experimentación General

6.1. Comparando con el Exacto

Debido al costo del exacto se realizaron tests de 3 nodos a 19 con 100 repeticiones para cada cantidad.

6.1.1. Distancia

Comparamos los resultados del algoritmo exacto, el goloso, la búsqueda local y GRASP. Calculamos las distancias a la solución del exacto.

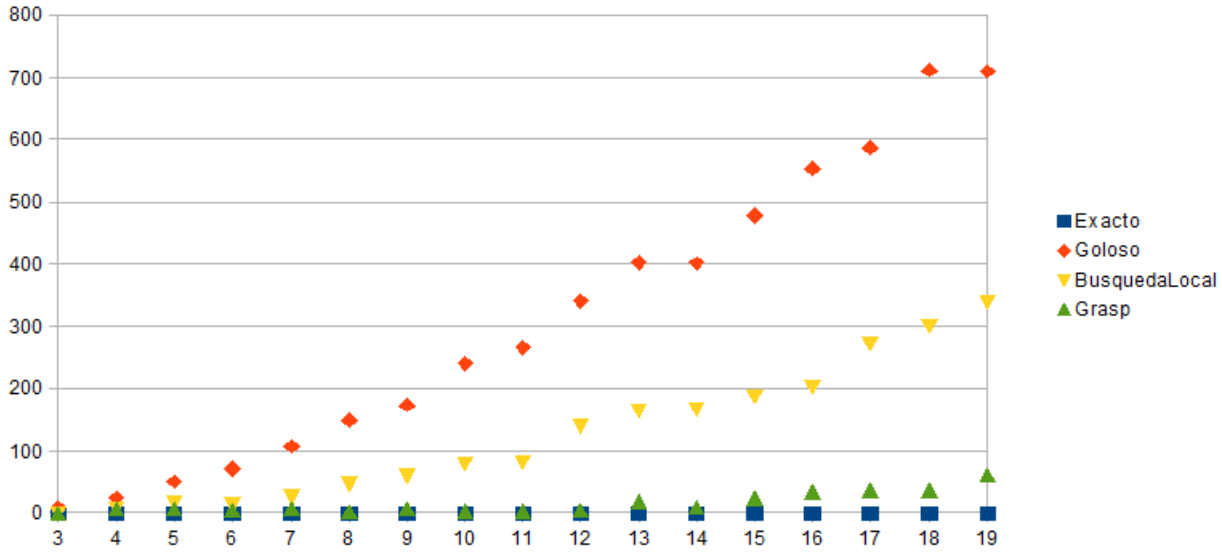

Figura 13: Grafos al azar

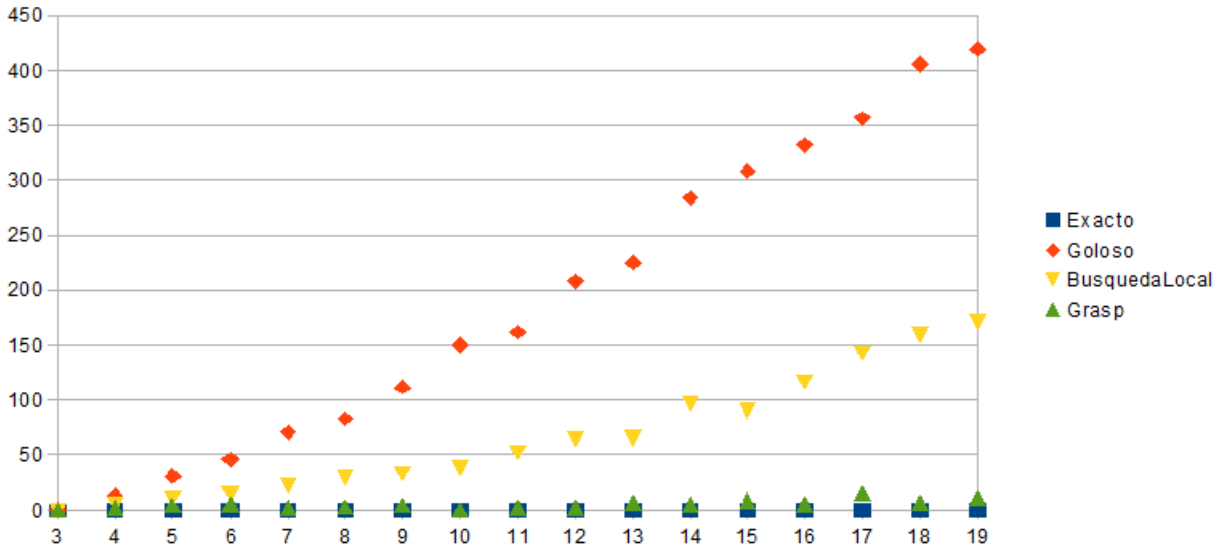

Figura 14: G y H densos

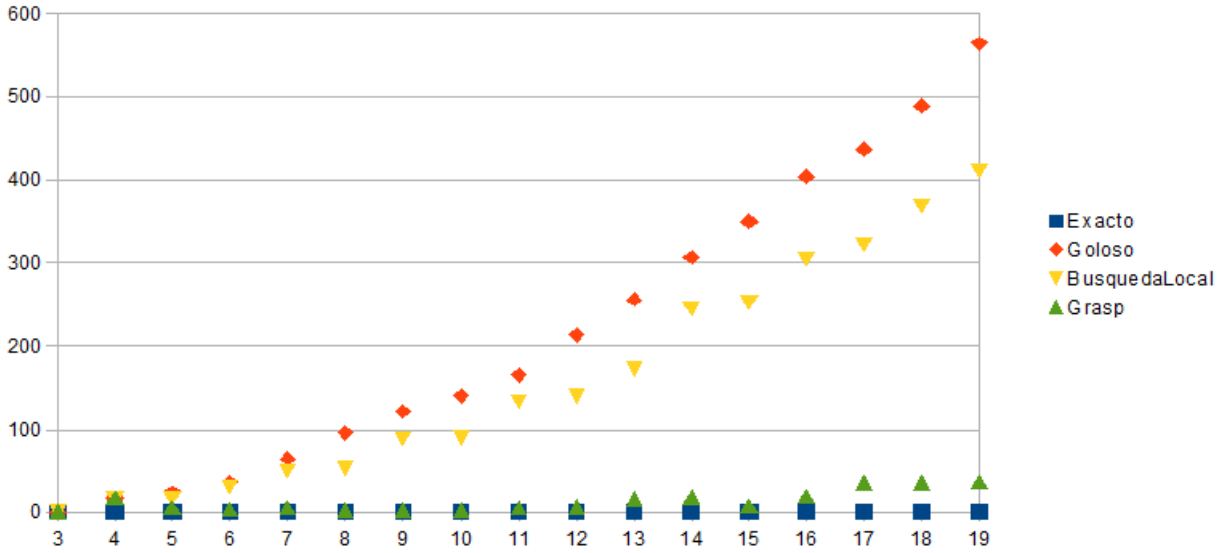

Figura 15: H es el complemento de G

Como se puede observar, casi no hay diferencias entre grafos aleatorios y densos. Aunque con los grafos donde G y H son complementos uno del otro, la búsqueda local aumentó considerablemente su distancia.

6.1.2. Efectividad

Ahora consideramos que un resultado es efectivo si dista menos del 10 % del resultado exacto.

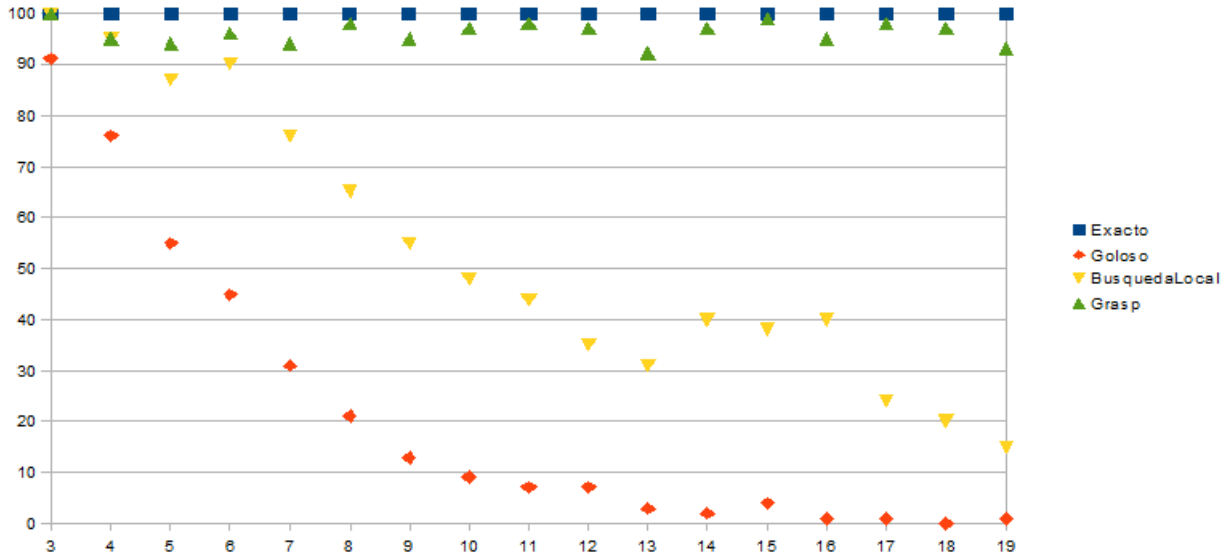

Figura 16: Grafos al azar

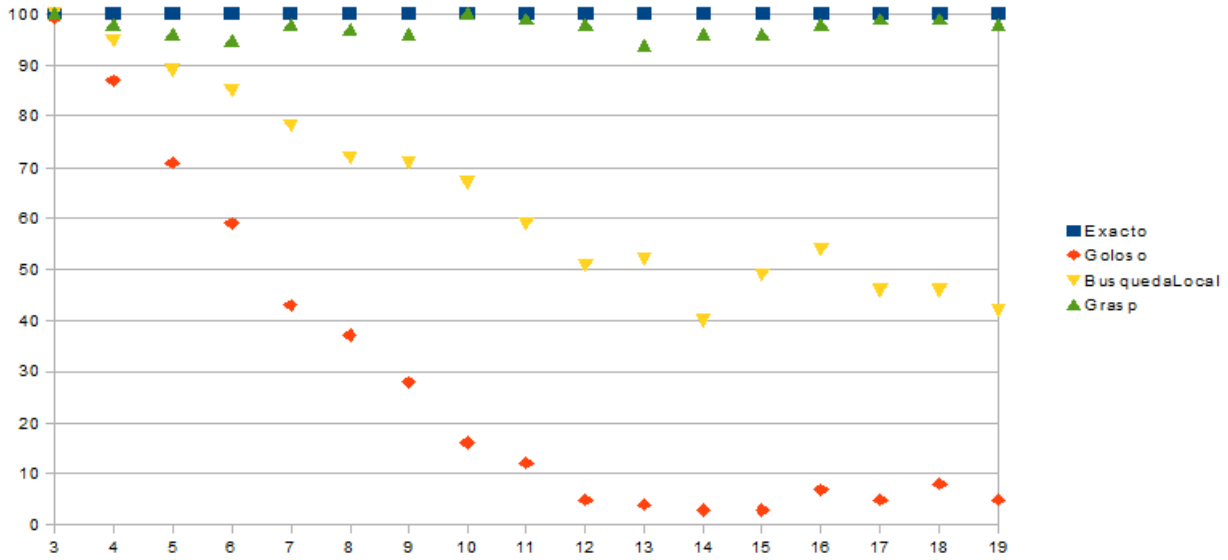

Figura 17: G y H densos

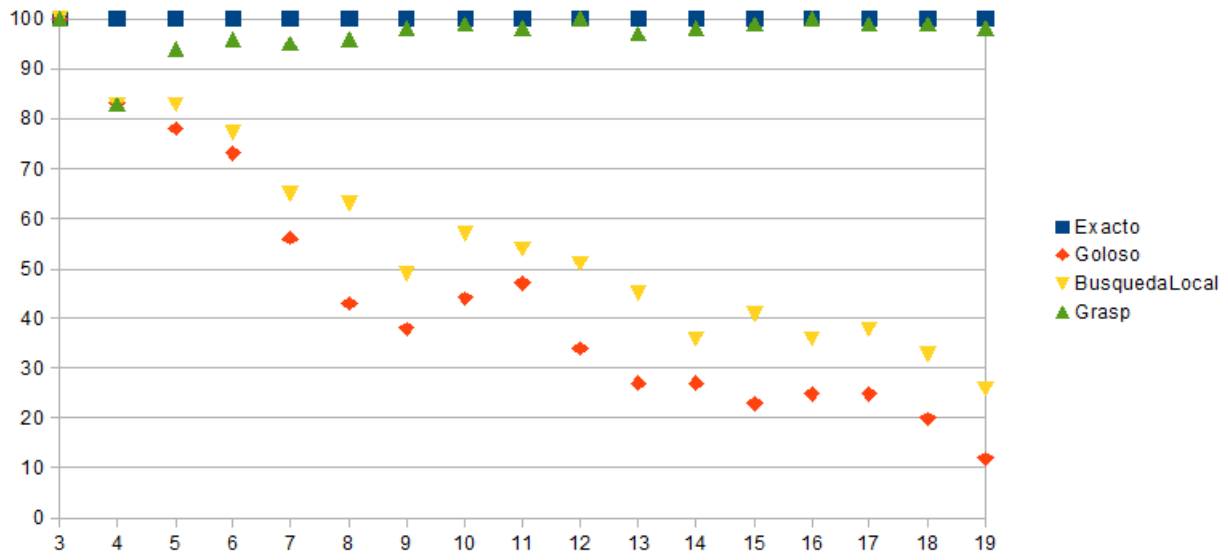

Figura 18: H es el complemento de G

Se observa que para estas cantidades de nodos la efectividad de GRASP no decae, a diferencia de las otras heurísticas.

6.2. Comparando con Grasp

Dejando de lado el algoritmo exacto, podemos experimentar con una mayor cantidad de nodos. Comparamos la heuristica golosa, la búsqueda local y GRASP

6.2.1. Efectividad

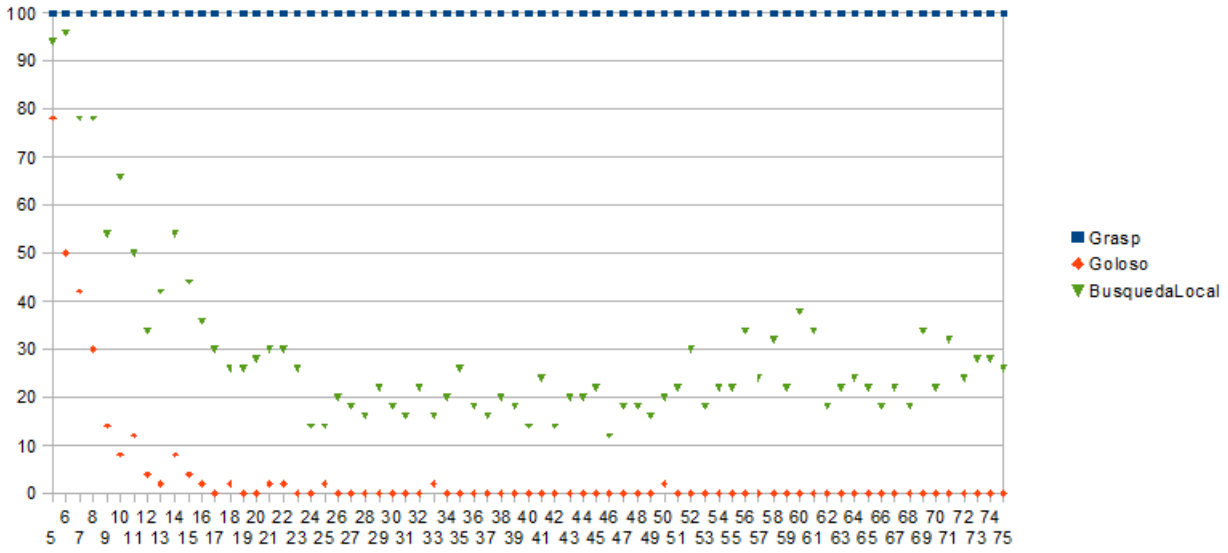

Figura 19: Comparación de efectividad contra Grasp

Podemos ver como a partir de cierto punto, la heurística golosa no es para nada efectiva (según el criterio anteriormente mencionado). En cambio, la búsqueda local se *estanca* entre el 10 % y 40 % de efectividad.

6.2.2. Costo

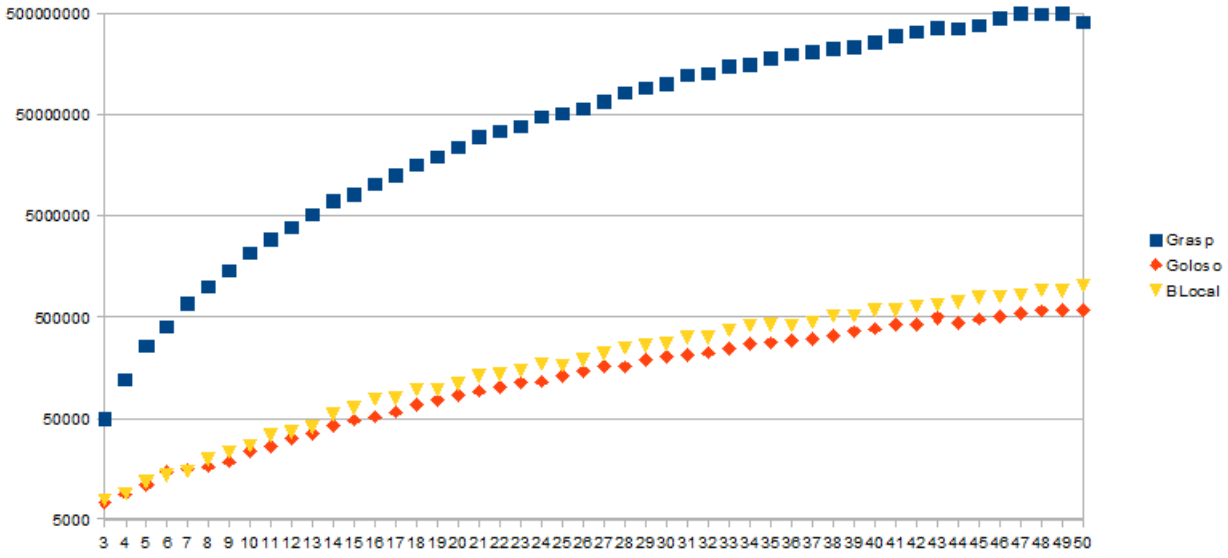

Figura 20: Costo temporal de las heurísticas, en escala logaritmica, contra Grasp

En este gráfico podemos observar como GRASP a pesar de obtener resultados mucho mejores que las otras heuristicas, consume mucho más tiempo de procesamiento.

Si bien la búsqueda local cuesta apenas más que la golosa, en comparación con GRASP, se obtiene muchos mejores resultados.