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

En este trabajo se presenta otra variante al problema de coloreo, conocida como Coloreo de Máximo Impacto (CMI). Se define el **impacto** de un coloreo C sobre un grafo G = (V, E) como el número de aristas $vw \in$ E tal que el color de $v$ es igual al de $w$. Dados dos grafos G y H definidos sobre el mismo conjunto de vértices, CMI consiste en encontrar un coloreo C de G que al ser aplicado en H maximice el impacto de C en H. En este caso, los problemas que se pueden modelar tiene las siguientes características:

- se tiene un mismo conjunto de elementos, con dos criterios para relacionarlos entre si;
- se quiere asignar (o particionar) los elementos de forma tal que se maximiza la cantidad de elementos relacionados según criterio, mientras que se respeta el otro.

Presentamos aquí algunos ejemplos de esto aplicado a situaciones que pueden darse en la vida real:

En una facultad, se tiene n materias que corresponden a alguna de m áreas de estudio. Las materias pueden tener horarios solapados, de manera que no siempre es posible utilizar el mismo aula para dar dos materias distintas el mismo dia. Ademas, cada materia necesita elementos de trabajo especificos, según el área a la que corresponda. Se desea asignar las aulas a las materias, de manera tal que se maximice la cantidad de materias del mismo area que se dicten en el mismo aula.

Podemos resolver el problema planteando dos grafos G y H con las materias como nodos, tales que :

- en G, los nodos están relacionados por una arista si las materias tiene horarios solapados;
- en H, los nodos están relacionados si pertenecen al mismo área de estudio.

En este caso, los colores a usar representaran las distintas aulas a asignar. Un coloreo de impacto máximo será aquel que en G no asigne las mismas aulas a materias con horarios solapados, mientras que en H, maximiza la cantidad de materias del mismo área que comparten el mismo aula.

conjunto de locales
g: si son del mismo rubro
h: si estan en la misma zona geografica
coloreo: visitantes
CMI: maximizar la cantidad de locales de distinto rubro que pertenecen a la misma zona de ventas, y estan en la misma zona geografica

un conjunto de experimentos
g: relacionadas si comparten ejecutores
h: relacionadas son experimentos relacionados

colores = dias en los que se tiene que realizar
CMI: asignar los dias de manera de maximizar la cantidad de experimentos del mismo area que se realizan cada dia

# 2. Algoritmo Exacto

## 2.1. Algoritmo

Antes de desarrollar la explicación, definimos algunos elementos que utilizaremos luego. Sean G = (V, $E_G$), H = (V, $E_H$) dos grafos con V el mismo conjunto de nodos, y sea $n = |V|$. Consideramos que los nodos de V se encuentran numerados de alguna manera con los números de 1 a n. Sean los n colores representados por el conjunto C = {1, 2, ..., n}; un coloreo será representado por una lista $f$ de n valores, cada uno de ellos pertenenciente al conjunto C, donde el nodo $i$ es pintado con el color en la posición $f_i$. Buscamos desarrollar un algortimo exacto que nos permita encontrar algún coloreo $f$ de G que use los colores de C, tal que el impacto (tal como se define en el enunciado) de $f$ en H sea máximo.

La idea general del algoritmo es simple: generar todos los coloreos del grafo G, y por cada uno que sea legal, calcular el impacto en H y guardar aquel con el que el valor obtenido sea máximo.

Para genererar todos los coloreos posibles, vamos a partir de la siguiente función recursiva:

**Algoritmo 2.1**

```
COLOREOS(nodo, coloreo)
if nodo ≤ LENGTH(coloreo)
      hacer algo con el coloreo
else
      for color in Colores
            coloreo[nodo] = color
            COLOREOS(nodo+1, coloreo)
```

Esta función genera todos las combinaciones posibles de los colores de C (*Colores*) en los nodos de G. Sin embargo, ésta es una tarea muy costosa, pues la cantidad de resultados posibles es de orden exponencial. Por ejemplo, si el grafo tiene n nodos, obtendríamos al menos $n!$ coloreos: podríamos colorear cada nodo de un color diferente, y después permutar los colores entre todos lo nodos. Más allá de esto, dadas las características del problema planteado, veremos algunos recursos que nos permitiran reducir drásticamente la cantidad de casos a analizar.

En primer lugar observamos que, para un grafo G dado, solo nos interesa analizar los coloreos que son legales (es decir, aquellos en los que si dos nodos son adyacentes, entonces están pintados de colores distintos). Esto nos da una pauta sobre como aplicar podas a medida que vamos generando los coloreos: al pintar un nodo i (es decir, elegir un color para el posición i del coloreo), utilizaremos solo los colores que produzcan un coloreo legal hasta ese momento.

Por ejemplo, en la figura **??** mostramos dos pasos en la generación de un coloreo para un grafo de 5 nodos. Podemos ver que si tenemos el nodo 2 pintado de color negro, al pintar el nodo 3 también de color negro obtendremos un coloreo ilegal para cualquier otra combinación de colores en los nodos siguientes.

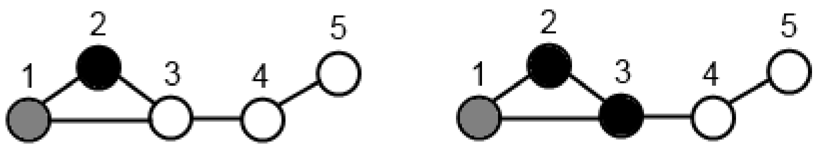



Figura 1: Ejemplo de coloreo de un nodo que lleva a coloreos ilegales.

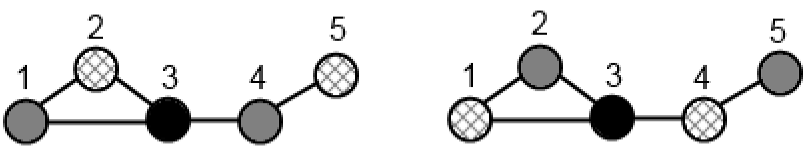
Por lo tanto, podemos aplicar una una poda considerando solamente los colores que sean legales con respecto a los de los nodos pintados con anterioridad:

**Algoritmo 2.2**

```
COLOREOS(nodo, coloreo)
if nodo ≤ LENGTH(coloreo)
      hacer algo con el coloreo
else
      for color in Colores
            if color legal para el nodo nodo
                  coloreo[nodo] = color
                  COLOREOS(nodo+1, coloreo)
```

Por otro lado, notamos que existen coloreos que son equivalentes (en el sentido en que uno puede obtenerse a partir del otro por medio de un renombramiento de sus colores), y por lo tanto, una vez analizado un caso, los demás solo aportan información redundante. Por ejemplo, supongamos que tenemos un grafo de 5 nodos 3-coloreable, y sean C = $\{$*gris* = 1, *rayado* = 2, *negro* = 3$\}$ los colores. Dos coloreos válidos son 1-2-3-1-2 y 2-1-3-2-1, como se muestra en la figura **??**. Sin embargo, fácilmente se ve que al intercambiar las etiquetas de los colores *1* y *2*, el primer coloreo puede transformase en el segundo, y el segundo puede transformase en el primero.

Figura 2: Ejemplo de dos coloreos legales que resultan equivalentes.




Recordemos que numeramos los nodos y los colores con los números de 1 a n, y dejemos de lado, por el momento, la condición de legalidad. Definamos un conjunto de coloreos F donde cada coloreo f es un vector

$$f = [f(1), ..., f(n)]$$

tal que, para cada uno, $f(v)$ cumple

- $m$ es el máximo en $f[1...v-1]$ (0, si $v-1 \leq 0$)
- $1 \leq f(v) \leq m+1$

Es decir, para pintar el nodo v, se usan $m+1$ maneras distintas, usando los colores ya usados, de 1,2...$m$ o un color nuevo, $m+1$.

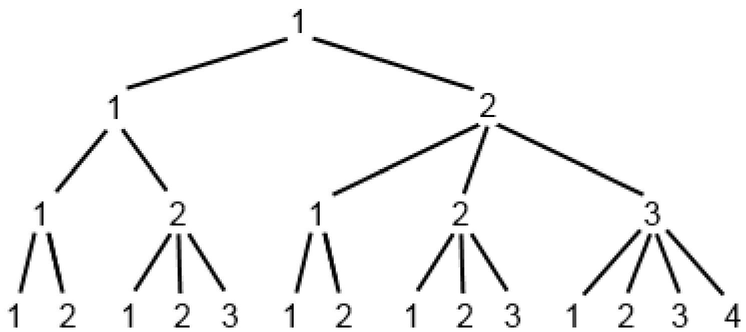



Figura 3: Ejemplo de un conjunto de coloreos según la caracterización dada. Cada camino desde la raíz a una hoja es un coloreo. Por ejemplo, 1-1-1-1, 1-2-1-2 y 1-2-3-4.

Por la forma en que estan construidos, resulta que dos coloreos f y f' cualesquiera de F no son equivalentes.

Además, cualquier otro coloreo que no esté en F es equivalente a uno que si lo está.

Entonces, teniendo esto en cuenta, para evitar analizar los casos de más definimos una forma más restrictiva para armar los coloreos. El pseudocódigo del algoritmo es el siguiente:

**Algoritmo 2.3**

```
COLOREAR(nodo, G, H, coloreo, solucion)
if nodo ≥ G.cantNodos // se pintaron todos los nodos de G
    impacto = impacto del coloreo en H
    if impacto > solucion.impacto
        solucion.impacto = impacto
        solucion.coloreo = coloreo
else
    maxColor = maximo color usado hasta el momento
    for c from 1 to maxColor
        if es legal pintar nodo de color c
            nuevoColoreo = coloreo
            nuevoColoreo[nodo] = c
            colorear(G, H, nuevoColoreo, solucion)
```

La función que resolverá el problema planteado es

**Algoritmo 2.4**

MAXIMOIMPACTOEXACTO($Grafo$ G, $Grafo$ H)
Sea *solucion* un vector de n+1 elementos
Sea *coloreo* un vector de n elementos
$solucion[0] = 0$
$coloreo[0] = 1$
$colorear(1, g, h, coloreo, solucion)$
return *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):

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

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

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

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

$$\begin{gathered}
T(n)=(n-n+1)*(n+m + T(n-1))= \\
= (n+m) + T(n-1)\leq
\end{gathered}$$

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

que es lo que queríamos ver.

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

maximoImpactoExacto cuesta entonces O(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

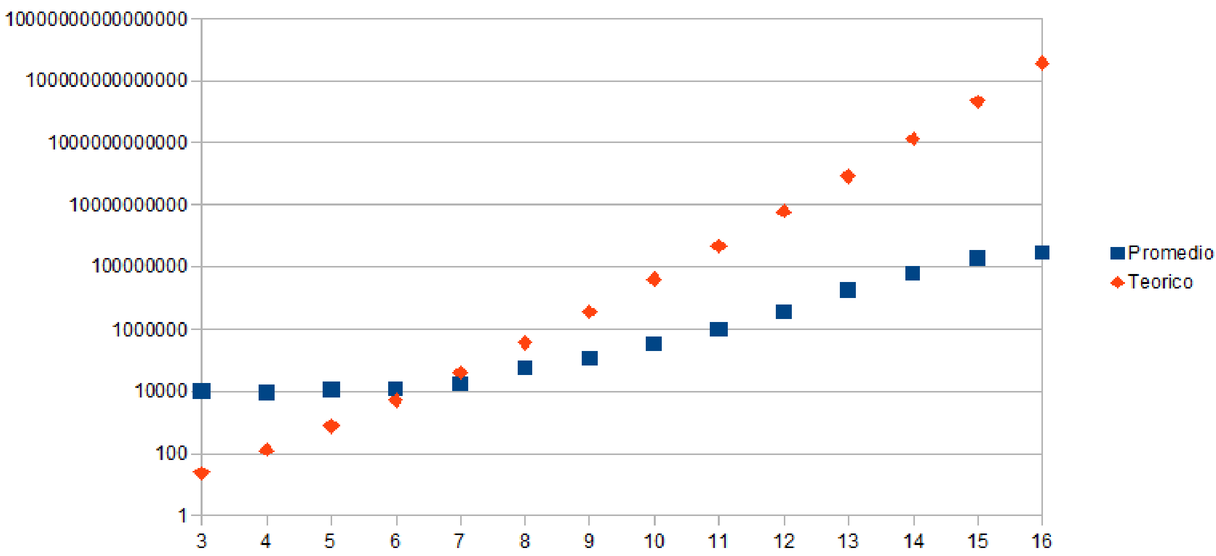



Figura 4: Costos

### 2.3.2. Grafos densos

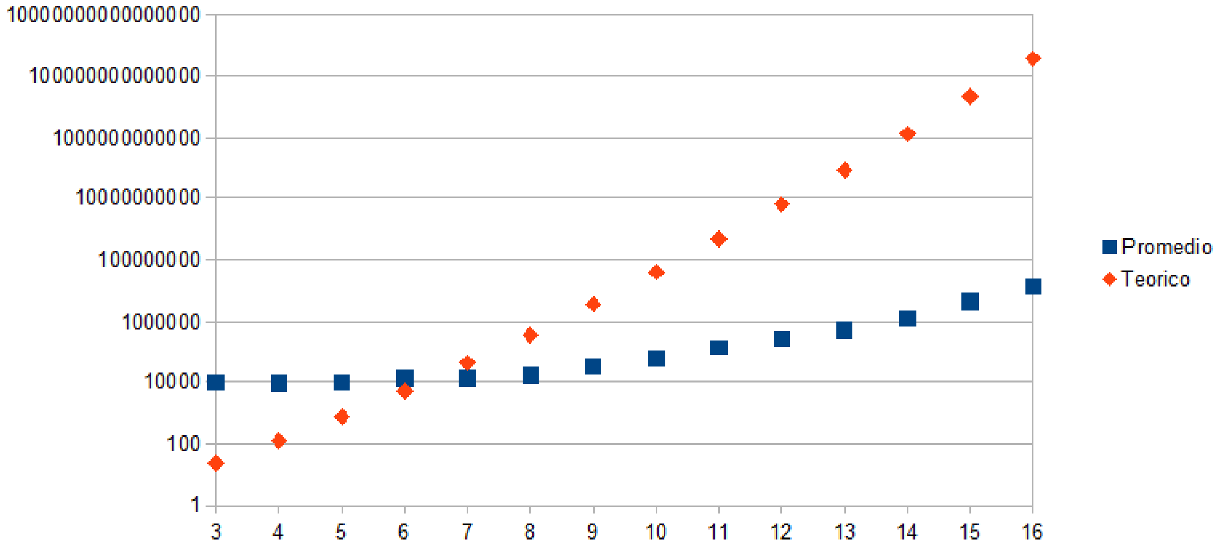



Figura 5: Costos

### 2.3.3. G y H complementos

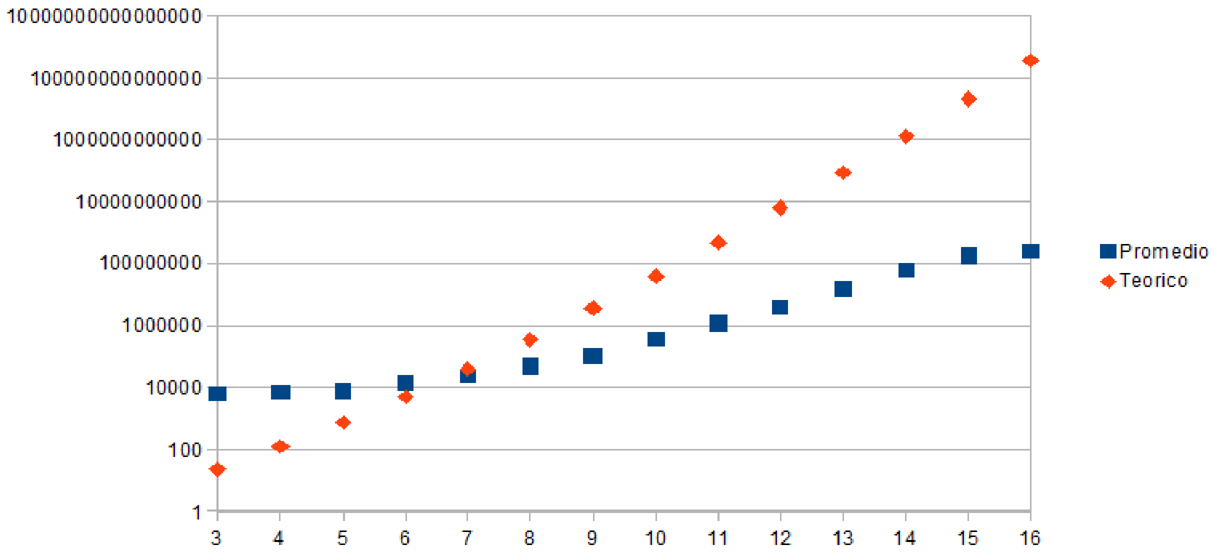



Figura 6: Costos

# 3. Heurística Constructiva Golosa

## 3.1. Algoritmo

El desarrollo de este algoritmo se encaró analizando ciertas características del problema a resolver. Dado que lo se busca es el máximo impacto posible, al comenzar el algoritmo se emplea una distribución de colores en los nodos que lo provea y que no es necesariamente un coloreo válido de G: aquella que surge de pintar todos los nodos con el mismo color (una distribución de colores que solamente es un coloreo para un grafo G que carezca de aristas en su totalidad y que para cualquier otro tipo de grafos no cumple con la definición de coloreo).

Teniendo en cuenta que partimos de un máximo, la idea es ir modificando el color de la menor cantidad de nodos posible hasta que se halle un coloreo válido para G, momento en el cual se deja de iterar y se devuelve el coloreo hasta ese momento obtenido. Qué nodo se priorizará modificar en cada paso se define en base a los siguientes criterios:

En primer lugar, deberá ser un nodo que no se haya modificado en algún paso anterior del algoritmo. En segundo lugar, que dicho nodo en caso de modificarse disminuya lo menos posible el impacto que se maneja hasta ese momento(lo que es lo mismo que decir que es el nodo que menos impacto aporta al coloreo). En caso de que dos o más nodos cumplan con estos requirimientos (lo que querría decir que modificar el color de cualquiera de ellos disminuye en la misma cantidad el impacto y que esa cantidad es la menor que se podría disminuir), se definieron ciertos criterios de desempate: se elegirá al nodo que mayor grado tenga en G ( ya que se busca lograr un coloreo legal de G modificando la menor cantidad de nodos posibles y al tener el nodo una mayor cantidad de vecinos en G modificar su color podría acercarnos más a un coloreo válido de G) y en caso de empatar en este criterio, se eligirá al nodo que menor grado tenga en H (porque se busca maximizar el impacto).

A la hora de implementarlo, sin embargo, le dimos una vuelta de tuerca más. Como precisábamos un algoritmo goloso con cierto grado de aleatoriedad, definimos una parámetro *porcentaje* número real entre 0 y 1 que se usa de la siguiente manera:

El proceso de elección del nodo mencionado anteriormente se implementó ordenando una lista de posibles nodos a modificar de manera tal que el nodo que mejor se ajuste a los requerimientos se encuentre en la primera posición. Una vez ordenada esta lista, se elegirá el nodo que se corresponda con el índice de dicha lista que se obtiene de obtener un número pseudoaleatoriamente y calcular el resto de dividirlo por el *porcentaje* del tamaño de la lista más 1 (es decir índice = rand% lista.size()**porcentaje* + 1). Es fácil observar que si *porcentaje*=0 entonces se devolverá el primer elemento de la lista, que es el nodo que mejor se ajusta a los criterios de elección.

La función que se encarga de la elección del nodo es $siguienteModificable$, que será llamada por *maximoImpactoGoloso* hasta que se encuentre un coloreo válido.

*maximoImpactoGoloso* es además la encargada de aplicar el cambio de color en el nodo devuelto por *siguienteModificable*.

El pseudocódigo es el que sigue:

**Algoritmo 3.1**

```
maximoImpactoGoloso(Grafo g, Grafo h, doubleporcentaje)
vector<unsigned int> res(n + 1)
int solucion[0] = 0
vector<unsigned int> coloreo(n, 1)
vector<bool> modificados(n, false)
while not G.coloreoLegal(coloreo))
    nodo = siguienteModificable(G,H,modificados,porcentaje)
    for c desde 1 hasta colores.size()
        if G.colorLegalDelNodo(nodo,coloreo,c)
        coloreo[nodo] = c
        exitFor
solucion[0] = impacto(h, coloreo)
for i desde 0 hasta n
solucion[i + 1] = coloreo[i]
return solucion
```

**Algoritmo 3.2**

```
siguienteModificable(Grafo g, Grafo h,vector<bool> modificados, double porcentaje)
vector<pair< unsigned int, unsigned int > > posibles
for n nodo in V(G)
    if not modificados[ nodo ]
    agregar(posibles,<G.impactoNodo(nodo,H,coloreo),nodo>)

sort(posibles)

unsigned int res = random(| posibles | * porcentaje)

return res
```

## 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 cantidad 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

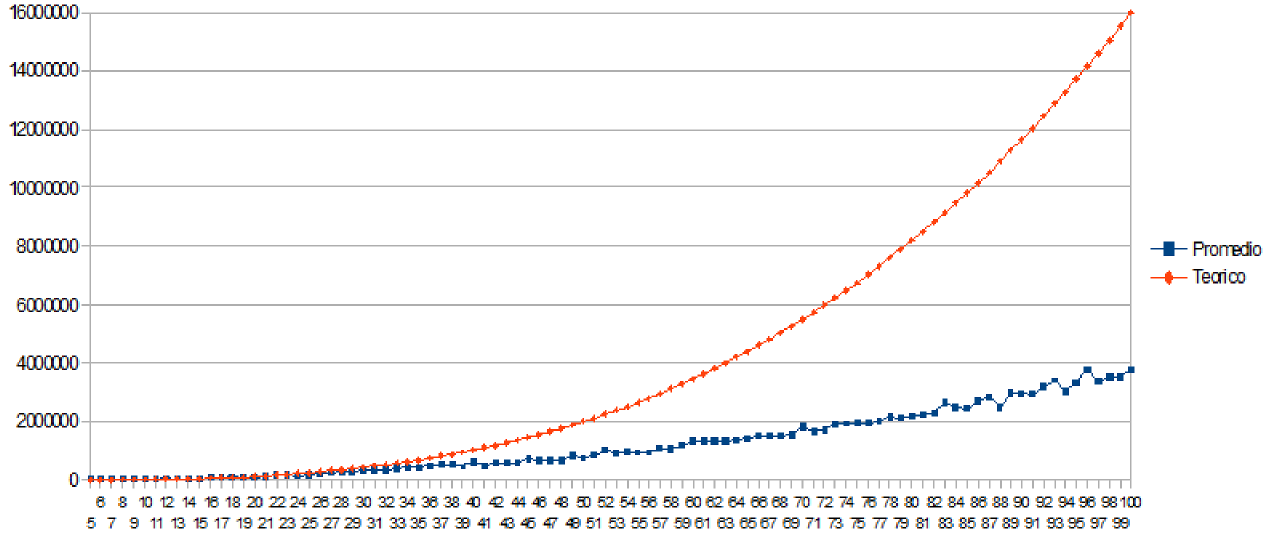



Figura 7: Costos

#### 3.3.2. Grafos densos

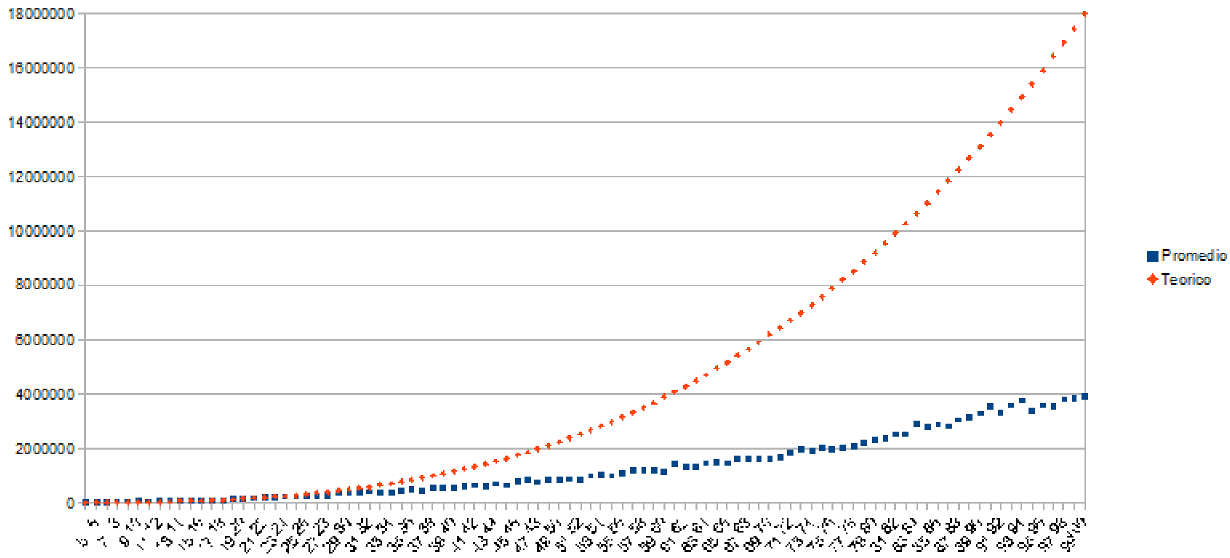



Figura 8: Costos

#### 3.3.3. G y H complementos

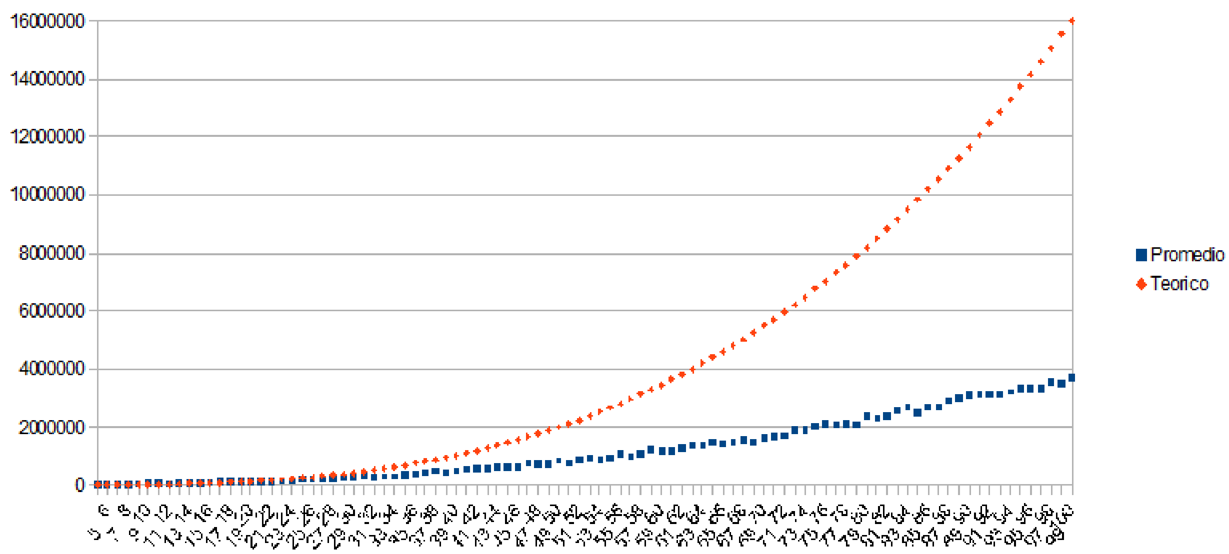



Figura 9: Costos

# 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.

Hay que tener en cuenta que al ser una búsqueda local, se termina *cayendo* en un mínimo local, con lo que probablemente no se obtenga la solución óptima del problema.

**Algoritmo 4.1**

```
maximoImpactoLocal(Grafo g, Grafo h, double porcentaje)

vector<unsigned int> impactoGoloso = maximoImpactoGoloso(g,h,porcentaje)
unsigned int impactoParcial = impactoGoloso[0];
vector<unsigned int> coloreo(n);

vector<unsigned int> solucionFinal(n+1);

for i desde 1 hasta n
    coloreo[i] = impactoGoloso[i]

unsigned int nuevoImpacto = 0

for i desde 1 hasta n

    vector<unsigned int> vecinos = vecinos del nodo i en h

    for j desde 1 hasta la cantidad de vecinos de i en h
        unsigned int color = coloreo[vecinos[j]]

        if pintar al nodo i de color es legal
            vector<unsigned int> nuevoColoreo = coloreo
            nuevoColoreo[i]= color
            nuevoImpacto = h.impacto(nuevoColoreo)

            if nuevoImpacto > impactoParcial
                coloreo[i] = color
                impactoParcial = nuevoImpacto


solucionFinal[0] = impactoParcial

for i desde 0 hasta n
    solucionFinal[i+1]=coloreo[i]

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

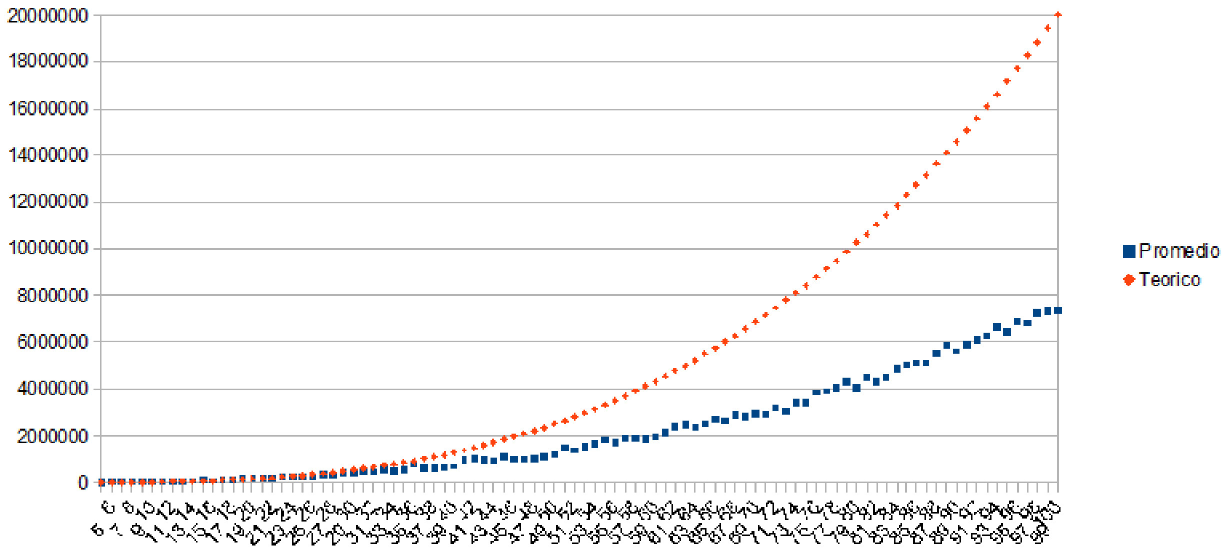



Figura 10: Costos

### 4.3.2. Grafos densos

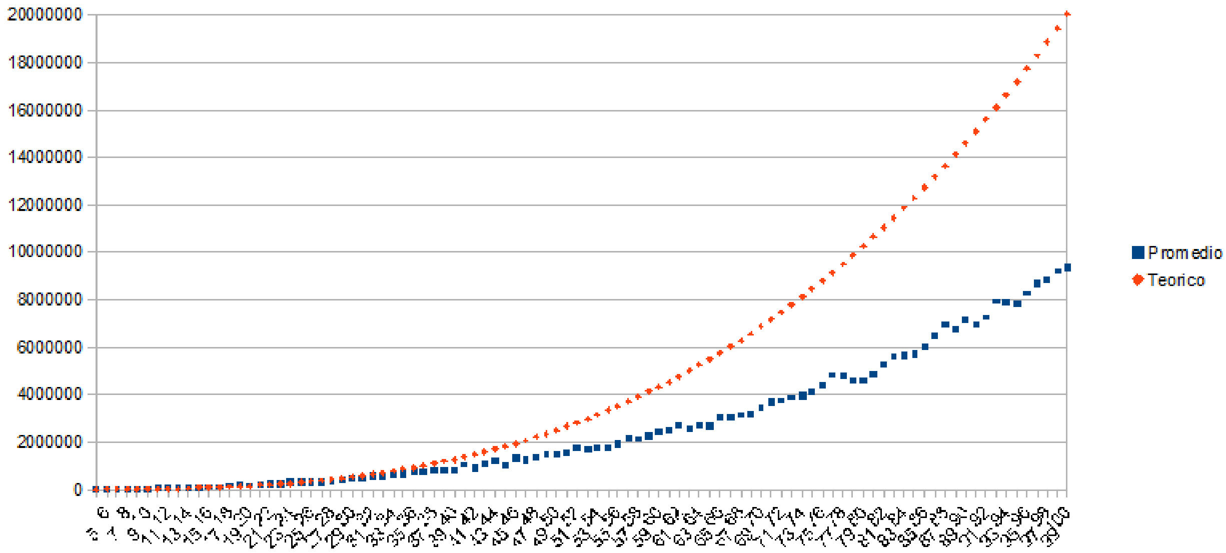



Figura 11: Costos

### 4.3.3. G y H complementos

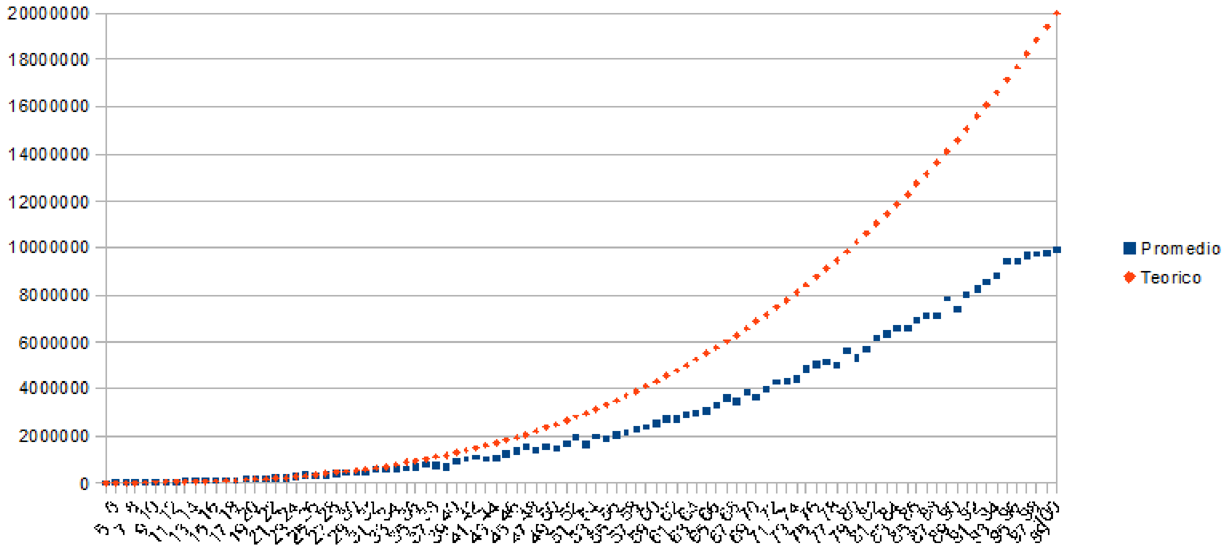



Figura 12: Costos

# 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)

```
vector<unsigned int> res(n + 1)
res[0] = 0
unsigned int sinMejora = 0

vector<unsigned int> coloreo(n,1) // Todos los elementos valen 1

for i desde 0 hasta maxIteraciones
    vector<vector<unsigned int>> rcl(maxRCL)
    for k desde 0 hasta maxRCL
        rcl[k] = maximoImpactoGoloso(g, h, porcentaje)

    unsigned int e = índice de uno de los elementos de rcl elegido al azar

    vector<unsigned int> solBusqLocal = maximoImpactoLocal(g,h,porcentaje,rcl[e])

    if solBusqLocal[0]>res[0]
        res[0] =solBusqLocal[0]

        for k desde 1 hasta n
            res[k]=solBusqLocal[k]

        sinMejora= 0
    else
            sinMejora++;
            if sinMejora == maxIterSinMejora
                salir del ciclo

    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

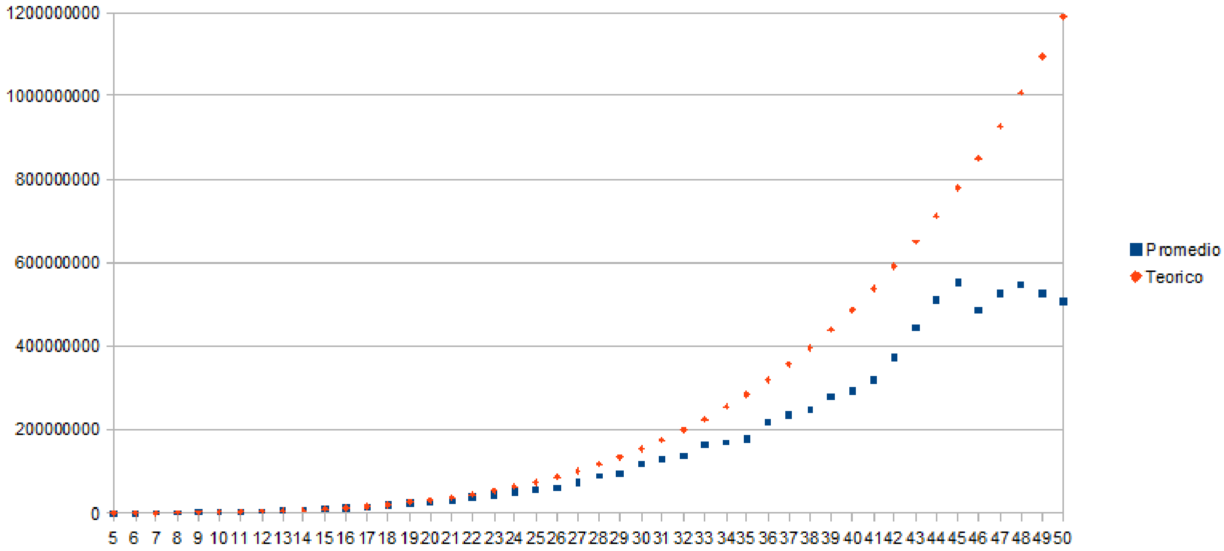



Figura 13: Costos

### 5.3.2. Grafos densos

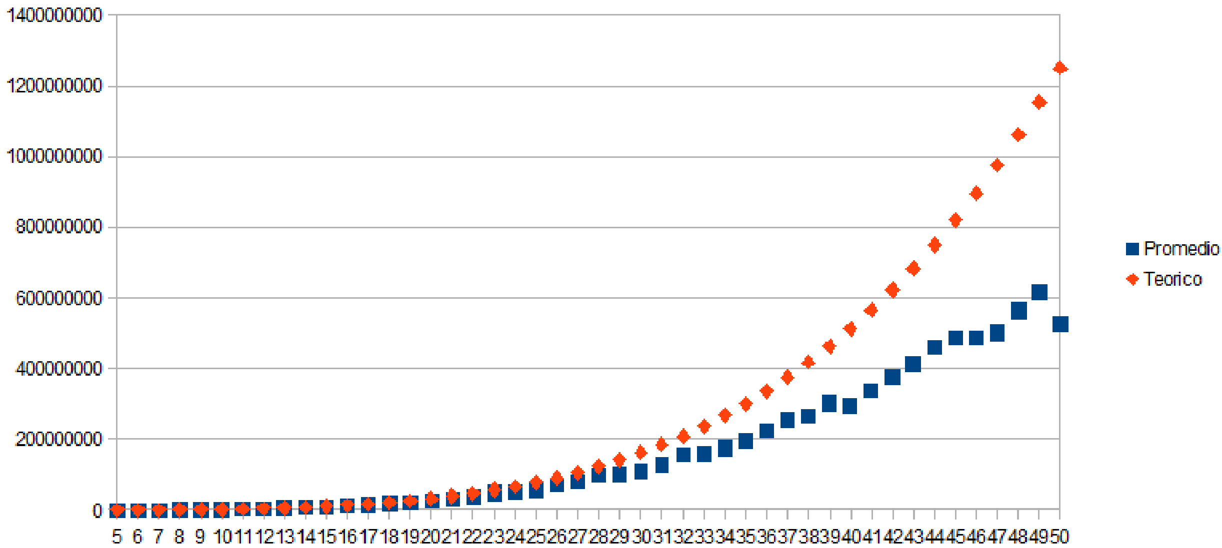



Figura 14: Costos

### 5.3.3. G y H complementos

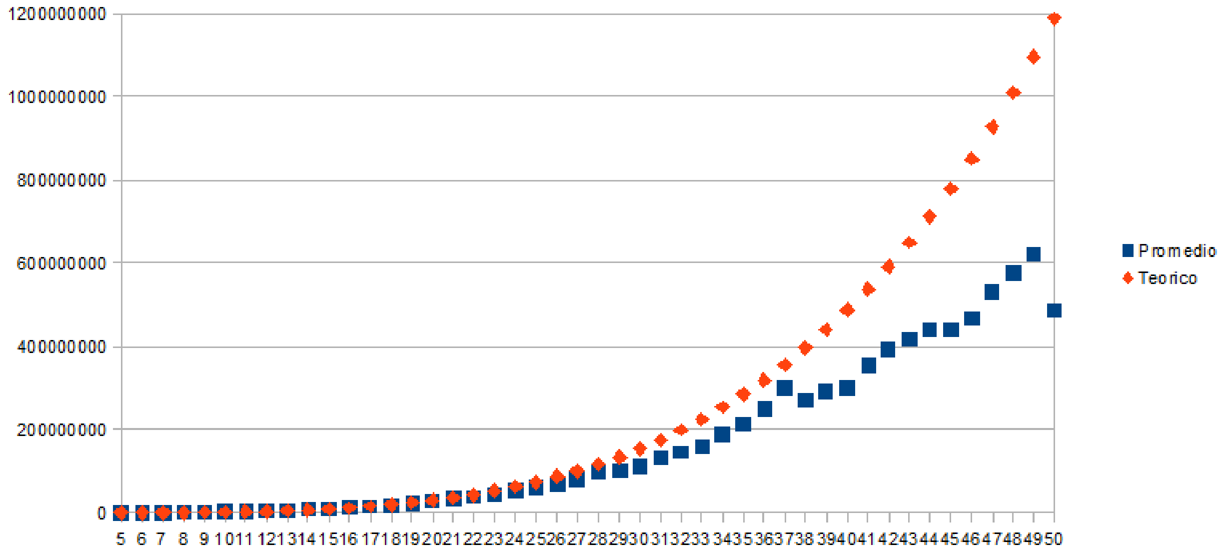



Figura 15: Costos

# 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.

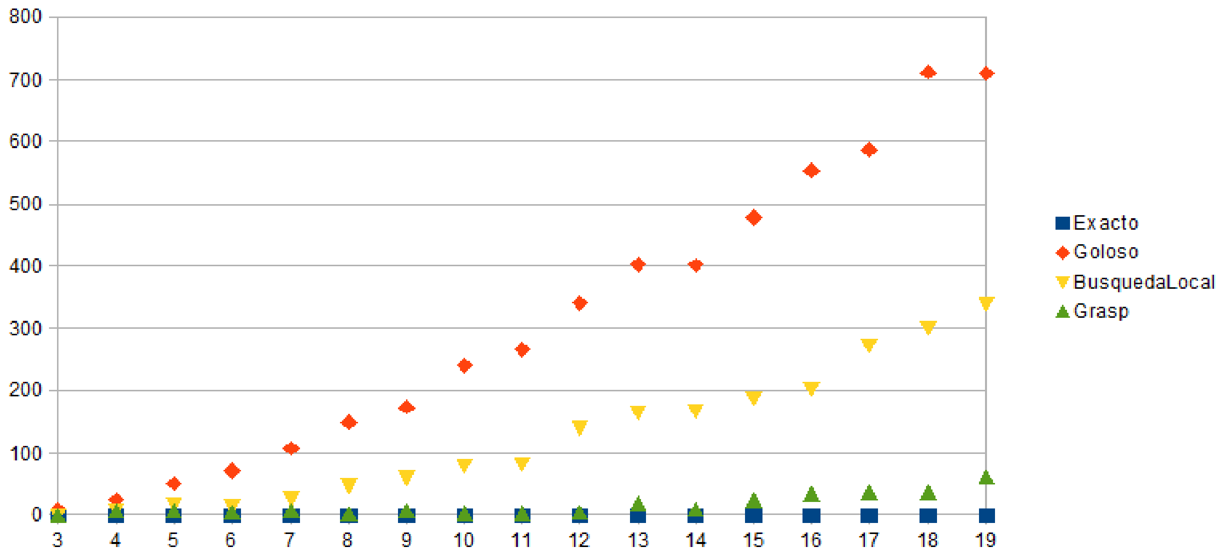



Figura 16: Grafos al azar

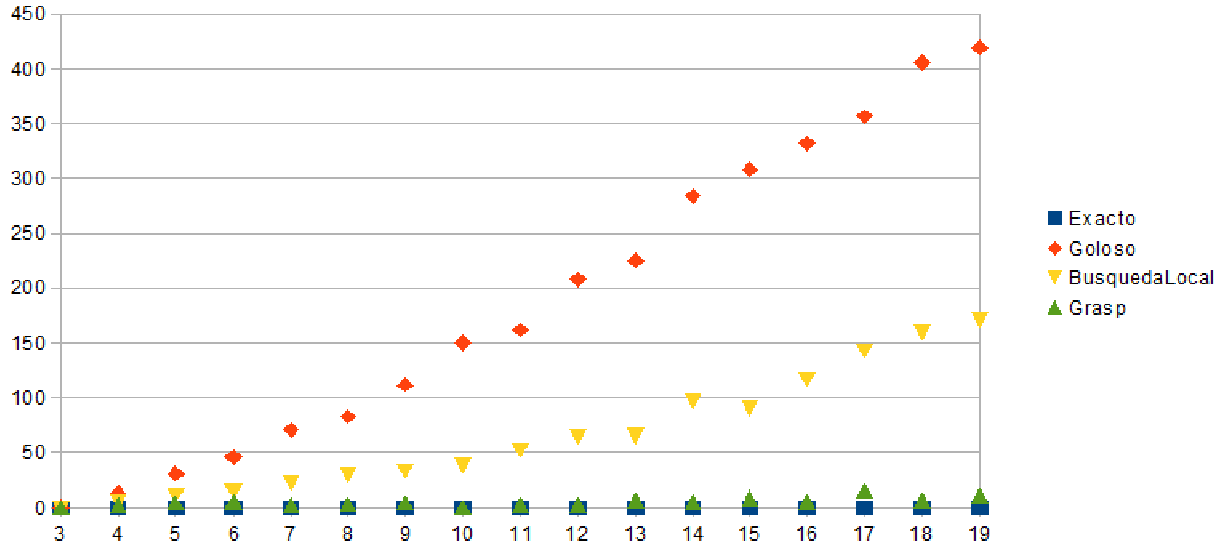



Figura 17: G y H densos

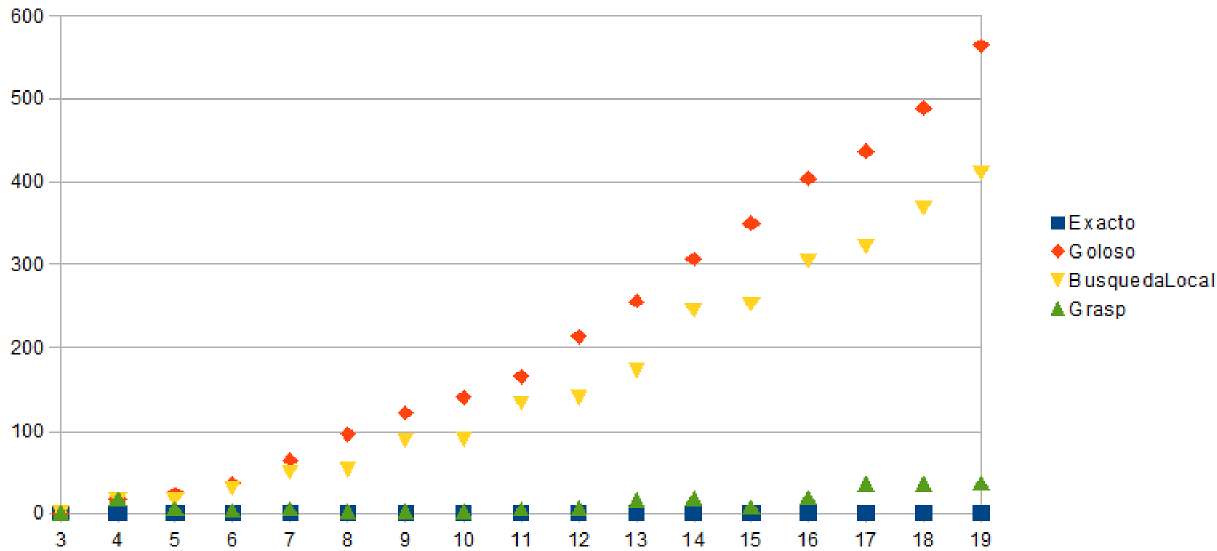



Figura 18: H es el complemento de G

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.

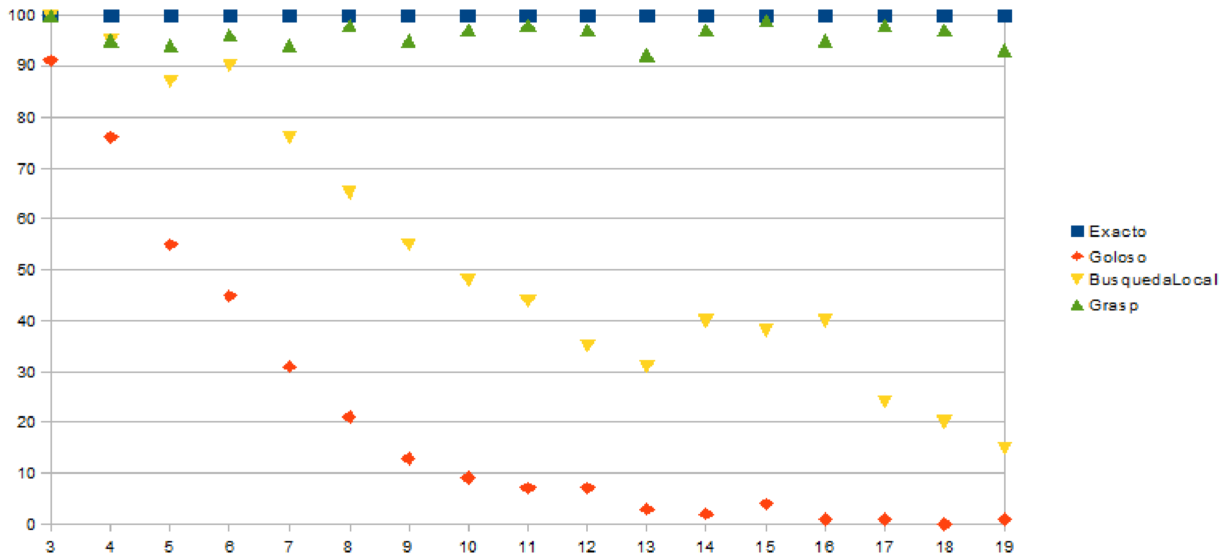



Figura 19: Grafos al azar

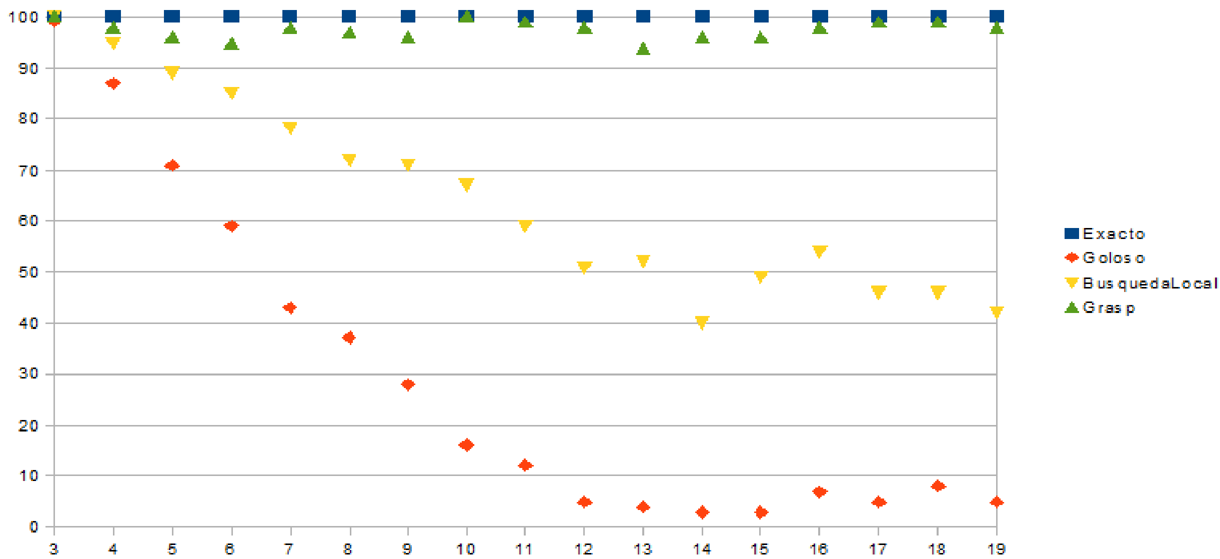



Figura 20: G y H densos

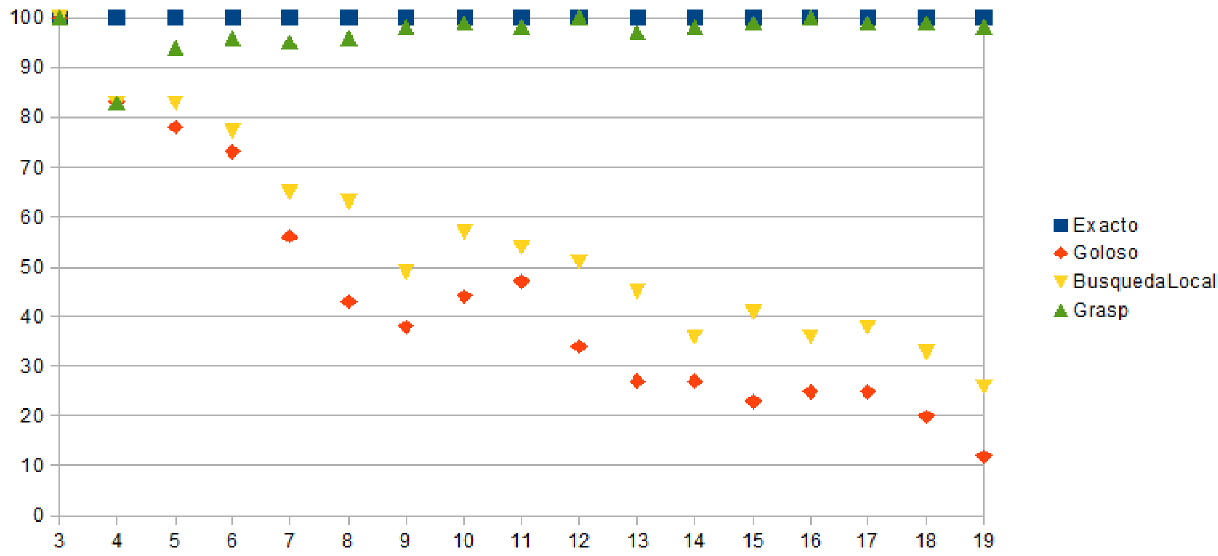



Figura 21: H es el complemento de G

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

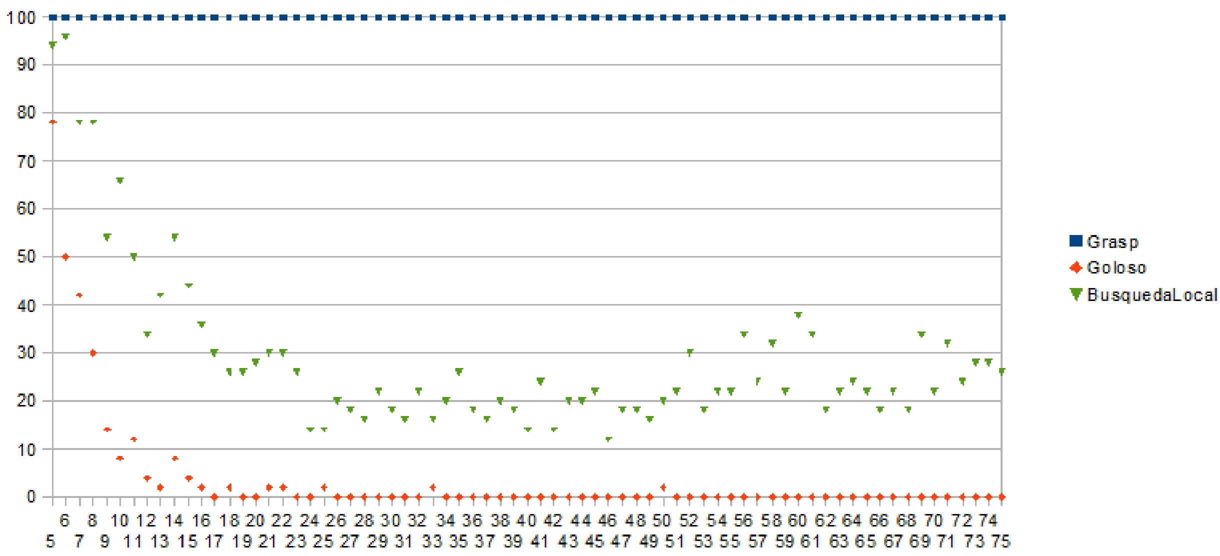



Figura 22: Comparación de efectividad contra Grasp

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

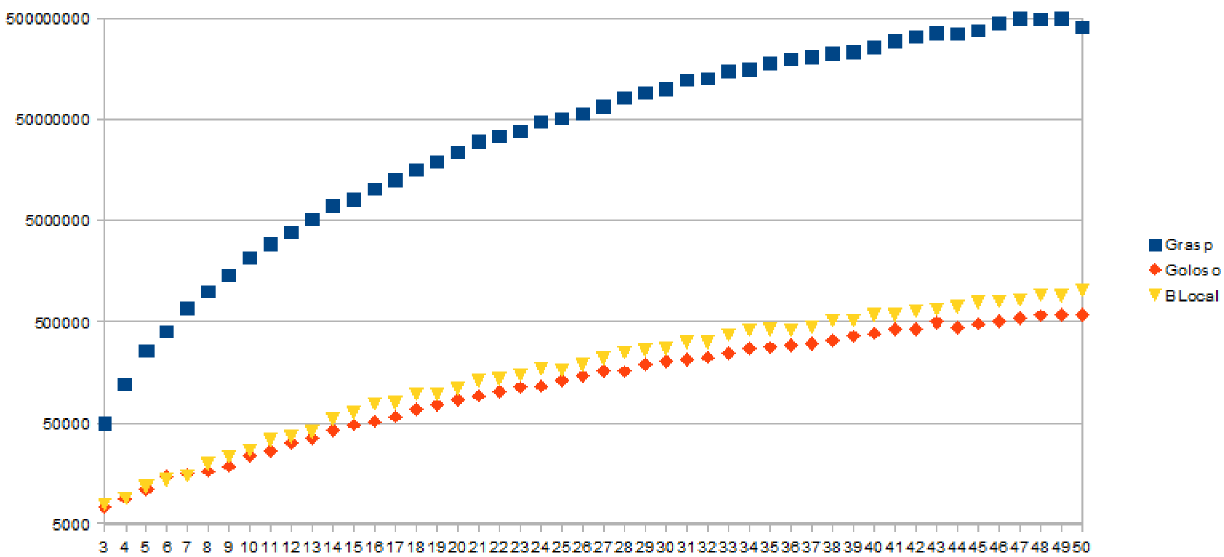



Figura 23: Costo temporal de las heurísticas, en escala logaritmica, contra Grasp

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.