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 general, observamos que 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.

1.1. Ejemplo 1

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 día. Además, cada materia necesita que en su aula se encuentren elementos de trabajo específicos, según el área a la que corresponda. A fin de poder equipar las aulas de la mejor manera reduciendo la cantidad de equipamiento necesario, se desea asignarlas a las materias de forma tal que se maximice la cantidad de materias del mismo area que se dicten en un 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,
- los colores asignados representan las aulas asignadas a las materias.

En este caso, un coloreo de máximo impacto 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.

1.2. Ejemplo 2

Un centro de salud tiene pacientes internados en distintas sedes. Los enfermeros del centro deben atender a los pacientes en horarios específicos, por lo si dos pacientes coinciden en estos horarios, el mismo enfermero no puede tratarlos. Debido a esto, puede ser necesario que los enfermeros se muevan entre las distintas sedes. A fin de minimizar este movimiento

de personal, se desea asignar los enfermeros a los pacientes tal que se maximice la cantidad de pacientes que cada uno debe tratar en una misma sede.

Aquí, en el modelado consideramos a los pacientes como nodos de los grafos G y H, tal que

- en G, los nodos están relacionados si los pacientes necesitan ser atendidos en el mismo momento;
- en H, los nodos están relacionados si los pacientes se encuentran en la misma sede.
- los colores asignados representan a los distintos enfermeros

Entonces, un CMI representa la asignación de enfermeros a los pacientes de manera que ningun enfermero sea asignado a dos pacientes si estos requieren atención en el mismo momento, mientras que se maximiza la cantidad de pacientes que cada enfermero ve en cada sede.

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)
1  if nodo ≤ LENGTH(coloreo)
2       hacer algo con el coloreo
3  else
4       for color in Colores
5            coloreo[nodo] = color
6            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.

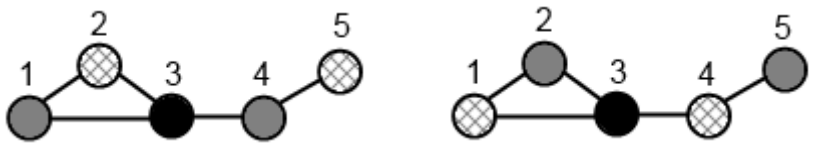
Por lo tanto, podemos aplicar una una poda considerando solamente los colores que sean legales con respecto a los de los nodos pintados con anterioridad:

Algoritmo 2.2

```
COLOREOS(nodo, coloreo)
1  if nodo ≤ LENGTH(coloreo)
2        hacer algo con el coloreo
3  else
4        for color in Colores
5              if color legal para el nodo nodo
6                    coloreo[nodo] = color
7                    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, $m+1$.

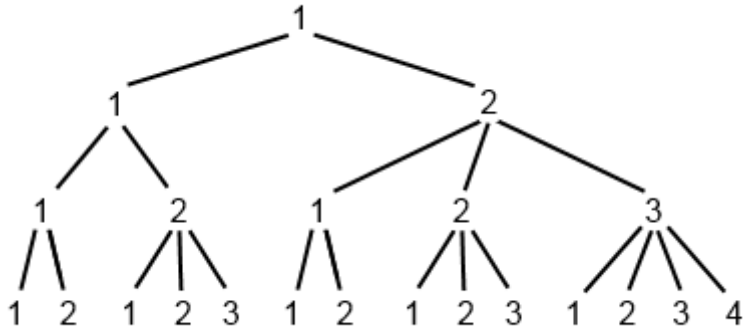


Figura 3: Ejemplo de un conjunto de coloreos según la caracterización dada. Cada camino desde la raíz a una hoja es un coloreo. Por ejemplo, 1-1-1-1, 1-2-1-2 y 1-2-3-4.

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)
 1  if nodo ≥ G.cantNodos // se pintaron todos los nodos de G
 2       impacto = impacto del coloreo en H
 3       if impacto > solucion.impacto
 4            solucion.impacto = impacto
 5            solucion.coloreo = coloreo
 6  else
 7       maxColor = maximo color usado hasta el momento
 8       for c from 1 to maxColor
 9            if es legal pintar nodo de color c
10                 nuevoColoreo = coloreo
11                 nuevoColoreo[nodo] = c
12                 colorear(G, H, nuevoColoreo, solucion)
```

La función que resolverá el problema planteado es

Algoritmo 2.4

```
maximoImpactoExacto(Grafo G, Grafo H)
1  Sea solucion un vector de n+1 elementos
2  Sea coloreo un vector de n elementos
3  solucion[0] = 0
4  coloreo[0] = 1
5  colorear(1, g, h, coloreo, solucion)
6  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):

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

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

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

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

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

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

que es lo que queríamos ver.

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

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

2.3. Experimentación y Resultados

Para realizar testeos y generar resultados lo más diversos posibles definimos 3 familias de grafos:

- **Grafos aleatorios:** Primero creamos un grafo de la cantidad de vértices deseada pero sin aristas. Luego recorremos todos los nodos por orden de etiqueta. Por cada nodo, determinamos si esta conectado a cada uno de los otros nodos. Usamos la función *rand()* de la Standar Library de $C++$ con una probabilidad del 30÷ si de que esté unido a un nodo.

  Vimos que la probabilidad si es mayor se generan casi siempre grafos conexos. Como buscamos que sean lo más diversos posibles, nos pareció un valor razonable donde se generan diversas componentes conexas y hasta incluso grafos conexos.

- **Grafos Estrella no uniformes (Star):** Se parte un nodo central luego se le añaden 4 nodos a los cuales los consideremos externos. Se van a ir agregando nodos hasta la cantidad deseada. Un nodo que se agrega puede estar conectado con un nodo *externo* o el nodo central. Si se agrega al nodo externo se considera como un nuevo nodo externo. Si se conecta con un nodo externo, éste deja de serlo y el nuevo nodo pasa a ser externo.

  Así, todos los nodos menos el central tienen grado uno o dos. Si es externo grado 1, caso contrario 2. El grado del nodo interno varia con cada creación de grafo. La elección a qué nodo se uno se hace aleatoriamente usando la misma función mencionada para los grafos aleatorios.

- **Grafos Web de 4 vértices (Red)[1]:** estos grafos son un caso particular de grafos Web_{st} que son t ciclos de s vértices cada uno conectados entre los ciclos un nodo con un solo nodo del ciclo aledaño.

[1] http://en.wikipedia.org/wiki/List_of_graphs

Aclaración Importante: Vamos a trabajar con estas familias de grafos con el resto de las experimentaciones de este trabajo práctico.

Se midieron los tiempos en corridas de 5 a 15 nodos con 50 repeticiones para cada cantidad de nodos.

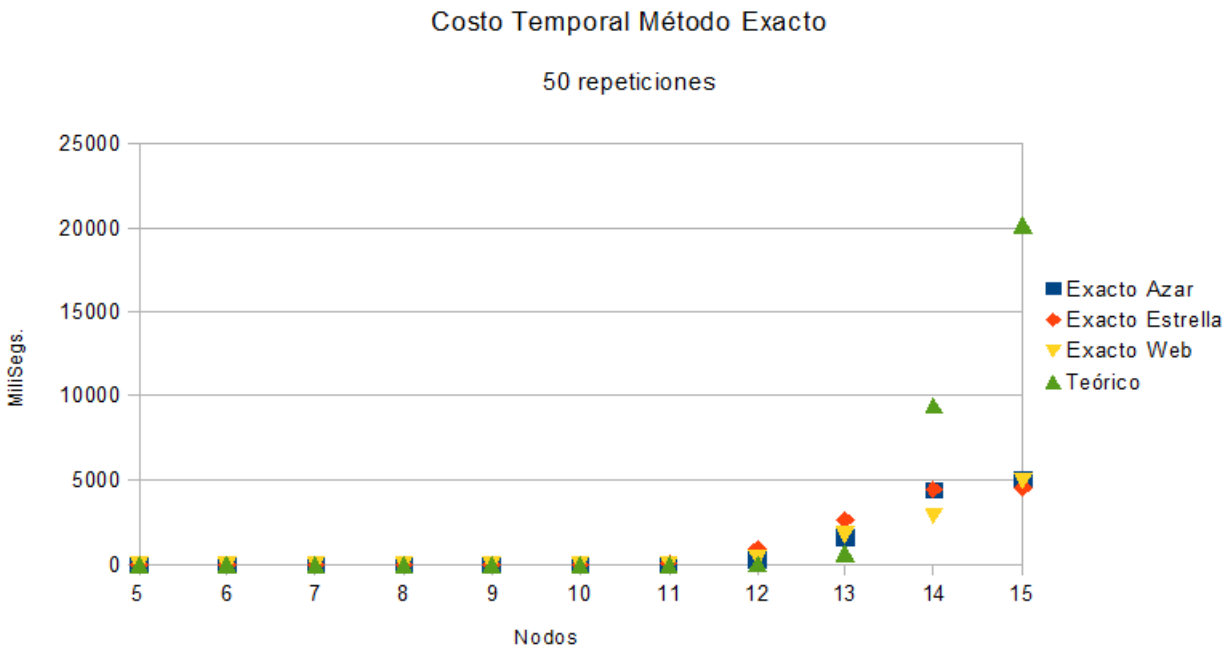


Figura 4: Costos

Debido al gran costo computacional de este método no se puede experimentar con una mayor cantidad de nodos sin que tome una cantidad de tiempo considerable. Elegir esa cantidad nos permite realizar varias repeticiones para obtener resultados más fiables.

Podemos observar que en este método no influye la diferencia entre las familias de grafos, tomando para cada uno el mismo costo temporal.

A partir del nodo 14 se ve claramente como la cota teórica sobrepasa a los resultados obtenidos. Éstos muestran una tendencia con una curvatura mucho menos pronunciada que la teórica.

Las mediciones estan en milisegundos, por lo que son despreciables los resultados con pocos nodos que toman pocos milisegundos pero se ve que a medida que aumenta la cantidad de nodos aumenta considerablemente el costo temporal.

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)
 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
```

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).

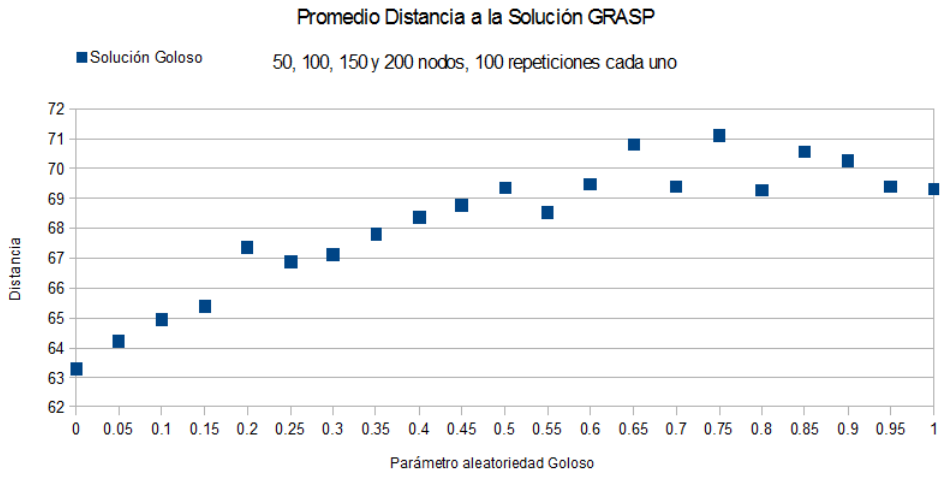
Por lo tanto, maximoImpactoGoloso cuesta O(n*(n+m)+ n^3).

3.3. Experimentación y Resultados

3.3.1. Optimización del parámetro

Como explicamos anteriormente, nuestra heurística recibe un parámetro que determina la aleatoriedad de la elección del siguente nodo a colorear con el objetivo de encontrar un coloreo del grafo G válido.

Comparamos los resultados obtenidos por el goloso contra la metaheurística GRASP, y calculamos la distancia de las soluciones. Es decir, la diferencia entre ambas. Probamos con 50, 100, 150 y 200 nodos con 100 repeticiones por cada cantidad de nodos. Testeamos con grafos aleatorios por su diversidad en tipos de grafos. Calculando un promedio obtuvimos:

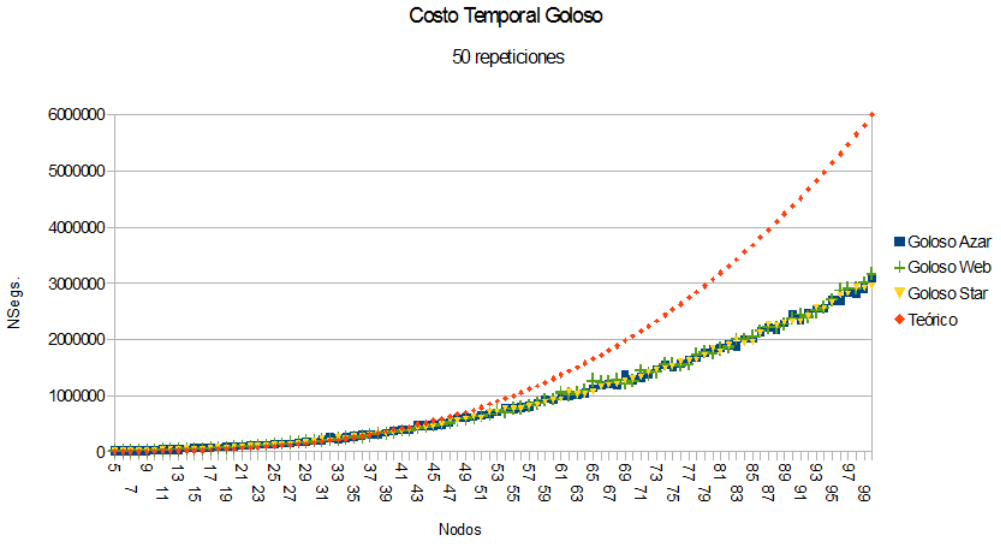


Podemos observar que cuanto mayor sea la cantidad de nodos a elegir aleatoriamente empeora el resultado. Esto se debe a que en la porción de lista que elegimos al azar, se encuentra ordenada por los criterios anteriormente desarrollados. Con lo que al principio se encuentran los mejores candidatos. Concluimos con que si se utiliza solamente esta heurística, es conveniente que el valor del parámetro sea 0.

3.3.2. Costo temporal

Una vez analizado el tema del parametro vamos a tomar el costo temporal de la heurística.

Analizaremos con grafos del tipo aleatorio, grafos de forma estrella no uniforme y grafos de red de 4 vértices.



Podemos observar que efectivamente se cumple con la cota teórica calculada anteriormente. Mantiene la tendencia asintótica de $O(n^3)$ pero más *suave*.

Si comparamos con las gráficas obtenidas con el método exacto es drástico la mejora en cuanto costos temporal pero como veremos más adelante lo que se gana en tiempo se pierde mucho en calidad de solución obtenida.

Se observa que con las tres familias de grafos con las que testeamos no hay diferencias apreciables en cuanto a costo temporal.

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)
 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

A continuación analizaremos el costo temporal de la búsqueda local. Al igual que antes, testeamos con grafos generados al azar, con grafos estrellas no uniformes y grafos web de 4 vértices.

Se midieron los tiempos en corridas de 5 a 100 nodos con 50 repeticiones para cada cantidad de nodos y se las comparó con la cota teórica calculada anteriormente.

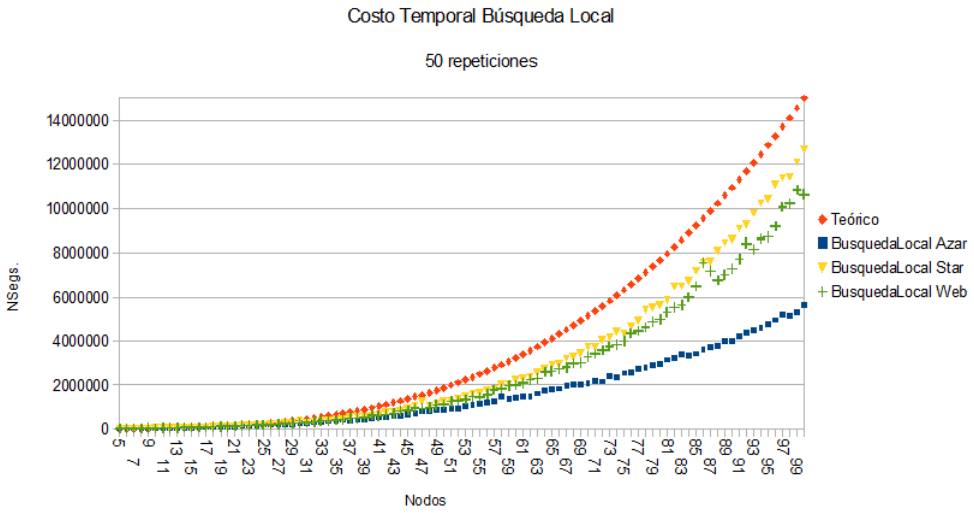


Figura 5: Costos

Se observa que se cumple con la cota del peor caso calculada.

A diferencia de la heurística golosa, se puede ver que con los tres tipos de grafos se obtuvieron diferentes resultados.

Con la que menor costos temporales se obtuvo fue con los grafos generados al azar. En cambio, vemos que con los grafos estrella el costo aumenta considerablemente. En menor medida ocurre con los grafos de red de 4 nodos.

Esto se debe a que estos dos últimos tipos de grafos en general son menos *densos* que los grafos generados aleatoriamente. Como nuestra heurística de búsqueda local busca

cambiar el color de un nodo por otro color legal (de algún vecino en el grafo H) en estos grafos menos densos hay mayor opciones por lo que se itera y prueba más veces.

Si bien las cotas teóricas de las heurísticas golosa y de búsqueda local son las mismas, las constante que las multiplica son distintas, siendo mayor la de esta última ya que requiere más cálculo computacional que proviene de explorar la vecindad de la solución golosa usada como base.

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

5.3.1. Optimización de parámetros

Primero buscamos optimizar los parámetros que utiliza nuestra implementación. Los cuales son el parámetro de la heurística golosa aleatoria que usa como base para ir armando las lista de candidatos, y los criterios de parada de una cantidad máxima de iteraciones y una cantidad máxima de iteraciones sin mejora en la solución que se va obteniendo.

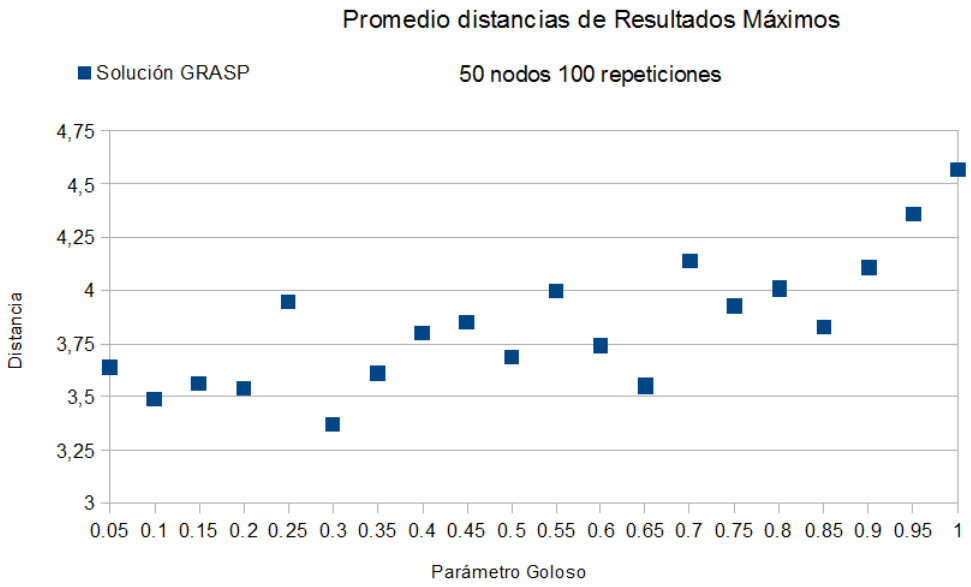
Para testear, medir y generar resultados usamos el concepto de distancia en este caso como la diferencia entre el valor máximo que se obtiene del valor de impacto para determinado grafo y el valor obtenido con la heuristica con ciertos valores de parámetros determinados.

- **Parámetro de la Heurística Constructiva Golosoa Aleatoria**

  Usamos como base la herística golosa aleatoria que desarrollamos anteriormente. Variamos el parámetro que recibe desde 0.05 hasta 1 con saltos de 0.05. Recordemos que cuando más chico es el valor, la heuristica golosa elige aleatoriamente entre menos candidatos posibles para ir coloreando, siendo si es valor 1, elige entre cualquier nodo posible.

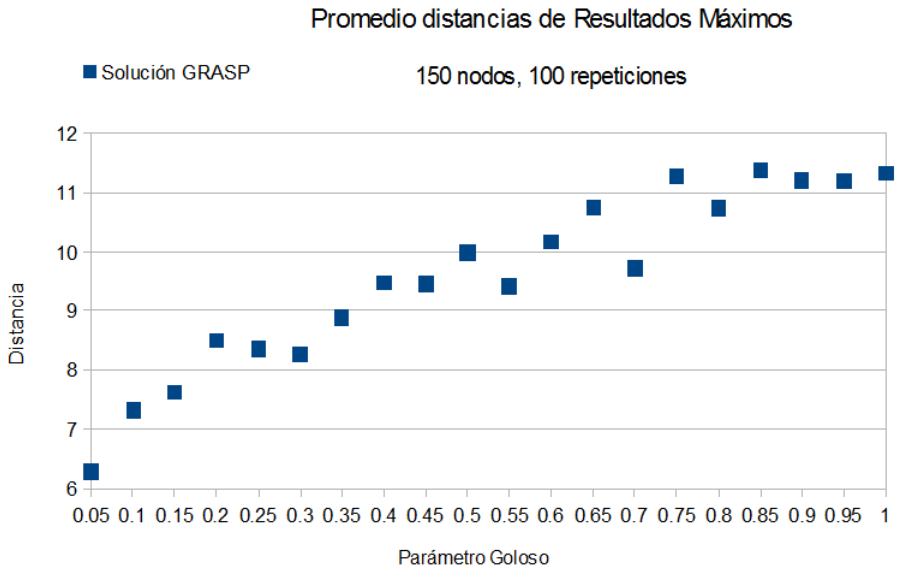
  Testeamos con 50, 100 y 150 nodos y 100 repeticiones para cada cantidad de los mismos. Los grafos elegidos para generar resultados son grafos generados al azar porque generar diversos tipos de grafos.

  Los resultados obtenidos con 50 nodos:



Con 100 nodos no se pudo apreciar diferencias con respecto a lo anterior.

Los resultados obtenidos con 150 nodos:



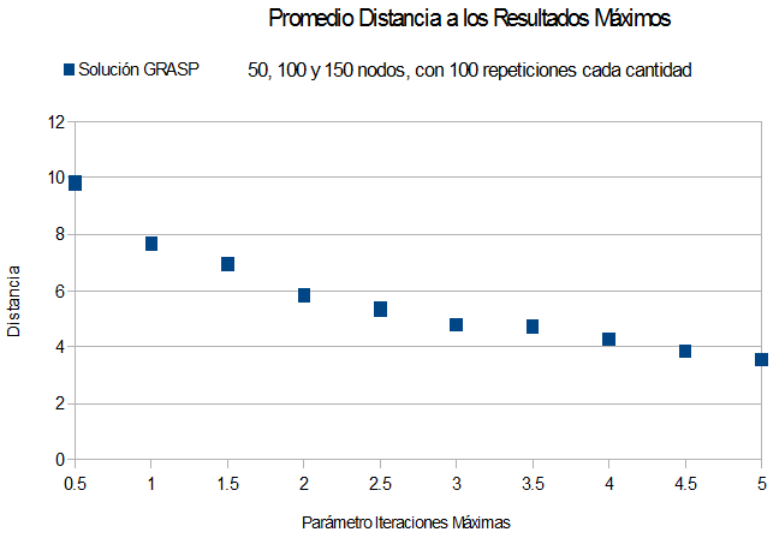
Tras obtener estos resultados, decidimos que cuando el grafo posea menos de 50 nodos el valor sea 0.3, en cambio, cuando tenga más nodos sea de 0.1. Esto significa que cuando más *chico* es el grafo, en el goloso aleatorio se elige al azar entre menor cantidad de nodos para colorear de cierta forma. Osea que cuando se construye nuestra lista de candidatos posibles, cuanto más *grande* es el grafo, se necesita menor porcentaje de aleatoriedad para crear posibles soluciones distintas entre si.

- **Parámetro Cantidad Máxima de Iteraciones**

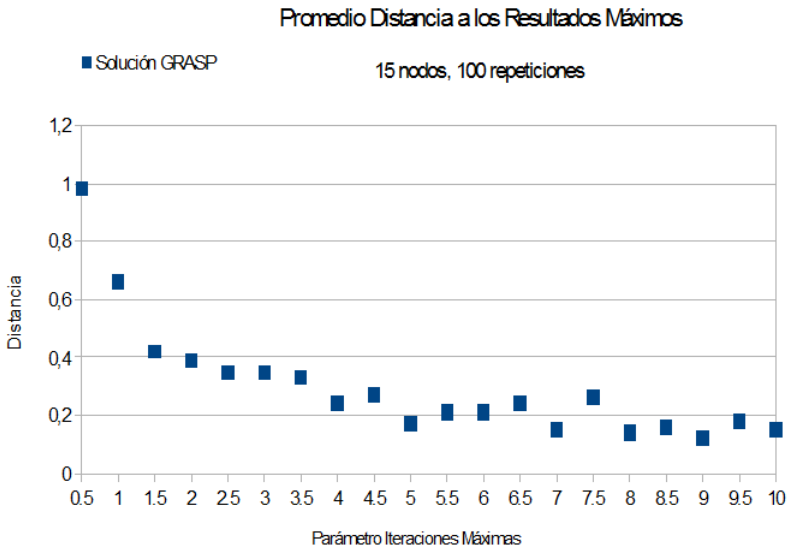
Como en nuestra implementación, este criterio de parada depende de la cantidad de nodos del grafo, decidimos variar para optimizarlo con los valores de 0.5 hasta 5, con pasos de 0.5. Es decir, es un coeficiente que multiplica la cantidad de nodos: siendo 2 se itera 2 veces la cantidad de nodos, y cuando es 5 se itera 5 veces la cantidad de nodos. Elegimos ese rango porque tenemos en cuanta que cuantas más iteraciones, más es el costo computacional de la heuristica.

Al igual que antes testeamos con 50, 100 y 150 nodos y 100 repeticiones para cada cantidad de los mismos. Los grafos elegidos para generar resultados son grafos generados al azar porque generar diversos tipos de grafos.

Lo que obtuvimos en promedio:



Se puede observar que a partir del valor 3 del parámetro, lo que se mejora de la soluciones que se van obteniendo es menos. Creemos que es el punto ideal entre costo computacional y mejora de las soluciones parciales. Sin embargo, cuando son grafos pequeños, la cantidad de iteraciones es poca por lo que probamos con menor cantidad de nodos y un rango para el parámetro mayor. Obtuvimos que lo óptimo sería el valor 5.

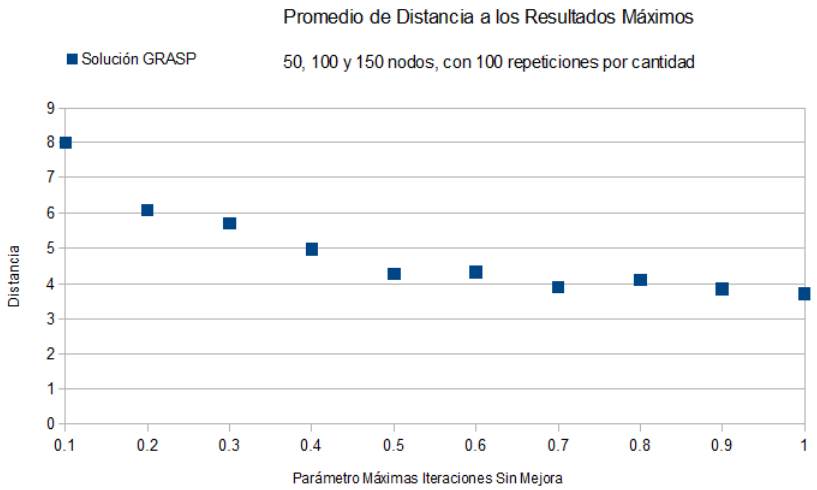


Aumentar el valor del parámetro de 3 a 5 no implicaria un gran costo computacional debido a que es menor la cantidad de nodos. Entonces queda que para grafos de menos de 50 nodos se utiliza la herística con cantidad máxima de iteraciones 5 veces la cantidad de nodos. Con grafos *más grandes* se utiliza 3 veces la cantidad de nodos.

- **Parámetro Cantidad de Iteraciones Máximas Sin Mejora**

Con este criterio de parada, determinamos si pasa cierta cantidad de iteraciones sin que se haya mejorado la solucón parcial obtenida, se finaliza. Este parámetro determina el porcentaje de las iteraciones máximas que tienen que pasar sin que se mejore para terminar. Por eso, los rangos posibles que definimos son de 0.1 a 0.9.

Al igual que con los otros dos parámetros, probamos con grafos aleatorios de 50,100 y 150 nodos con 100 repeticiones para cada cantidad.

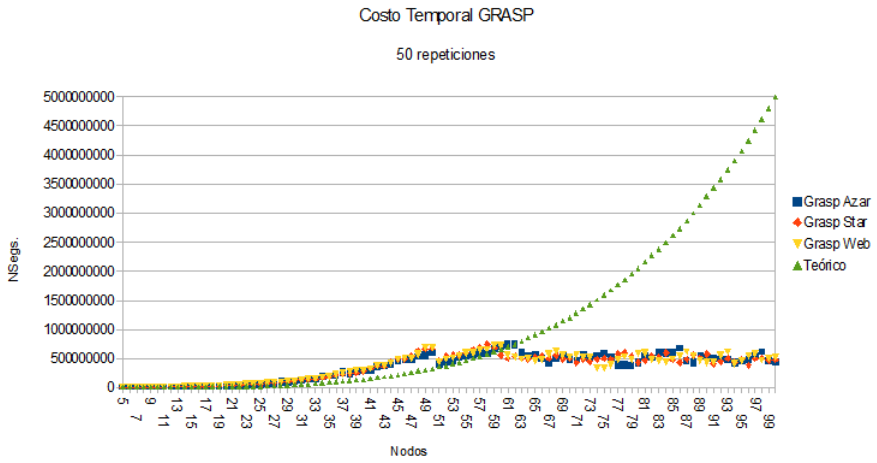


Podemos observar como a partir del valor 0.5, se estabiliza la mejora por lo que consideramos a ese valor como el óptimo. Es decir, que si no se obtiene una mejora en la solución parcial obtenida en esa cantidad de iteraciones se termina, osea, la mitad de la cantidad máxima de iteraciones de la heurística. Ejemplo: si son 500 iteraciones máximas, si se cumple que en 250 iteraciones consecutivas no hay mejora, se finaliza.

5.3.2. Costo temporal

Vamos a comparar el costo obtenido por nuestra heuristica versus el costo calculado teóricamente en la sección de complejidad.

Vamos a testear con 3 tipos de grafos: al azar, grafos estrellas no uniformes, grafos redes de 4 vértices.



Se ve claramente como a partir del nodo 50 hay un cambio brusco de los costos prácticos obtenidos. Refleja nuestro criterio de que para grafos con menos de 50 nodos se itera como máximo 5 veces la cantidad de nodos, caso contrario 3 veces. Lo que produce una disminución en los costos temporales.

Podemos observar que a partir de cierto punto se estabiliza el costo. Se debe a que se esta cumpliendo el segundo criterio de parada, el de si ocurre cierta cantidad de iteraciones donde no se mejora la solución se termina.

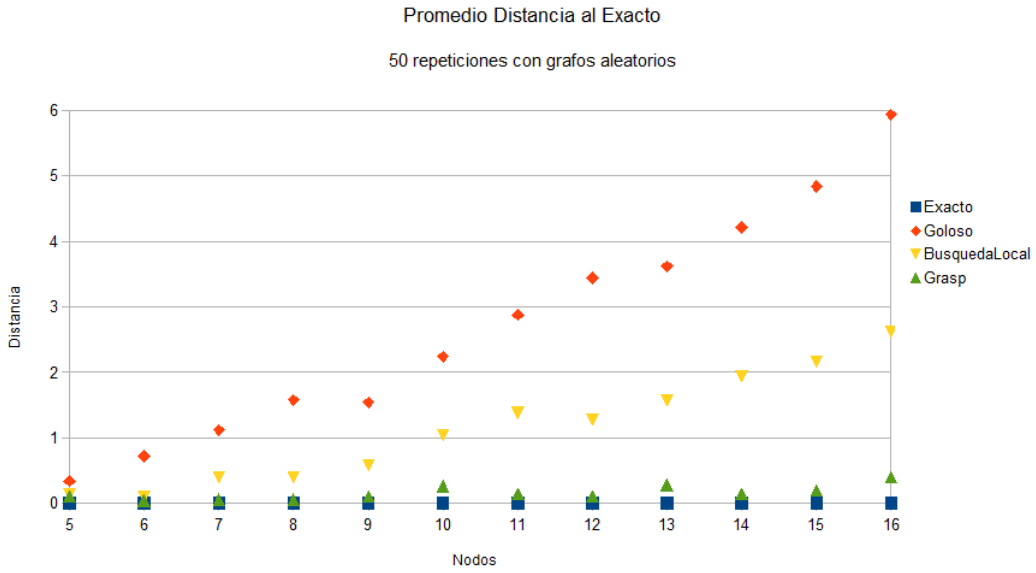
Con respecto a las distintas familias de grafos vemos que el comportamiento de nuestra heurística es prácticamente el mismo. No se observan grandes diferencias. Con lo cual concluimos con estas familias de grafos no influye en el costo temporal.

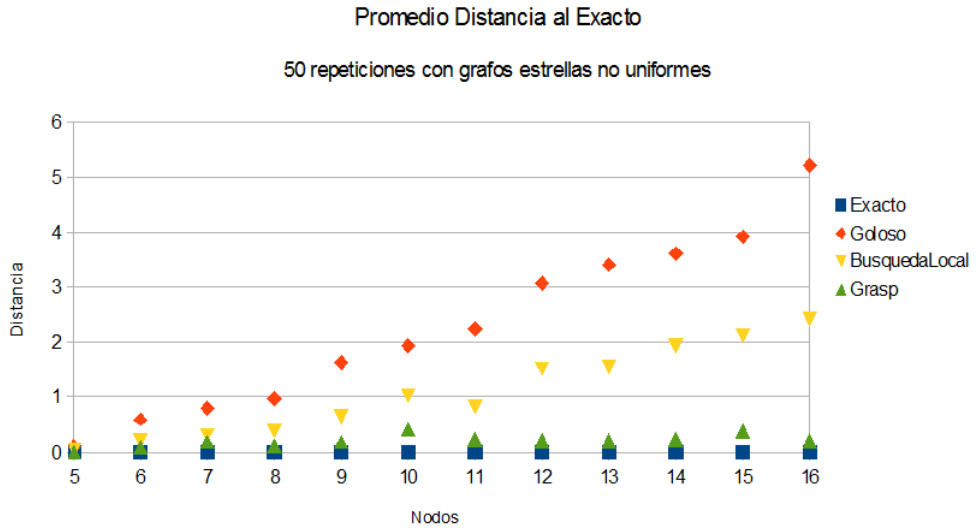
6. Experimentación General

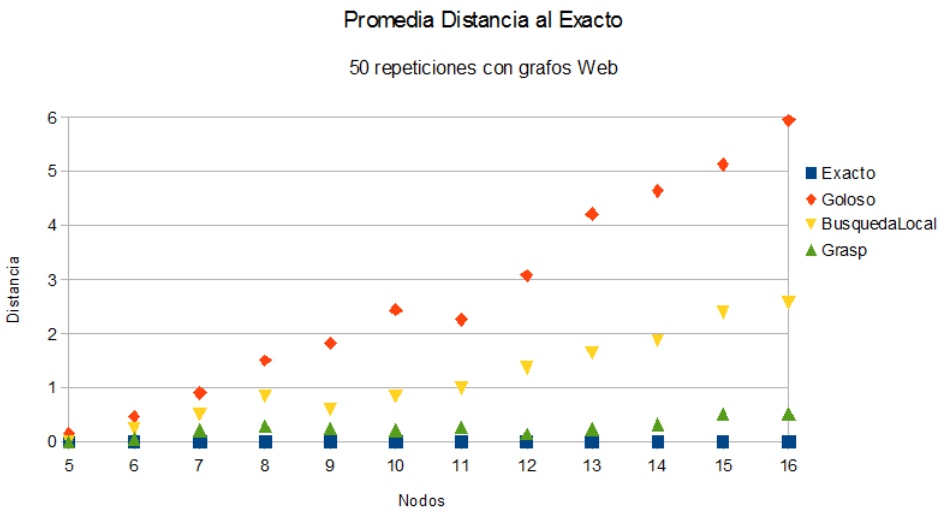
6.1. Comparando con el Exacto

Vamos a comparar todas nuestras implementaciones y a calcular las distancias con respecto al algoritmo exacto. Debido al costo del exacto se realizaron tests de 5 nodos a 16 con 50 repeticiones para cada cantidad. Definimos distancia como la diferencia entre el resultado exacto y el de la heurística correspondiente. Esto nos da una idea de *que tan mala es la heurística*.

Comparamos primero con grafos generados aleatoriamente, luego con grafos estrellas no uniformes y grafos *web*.







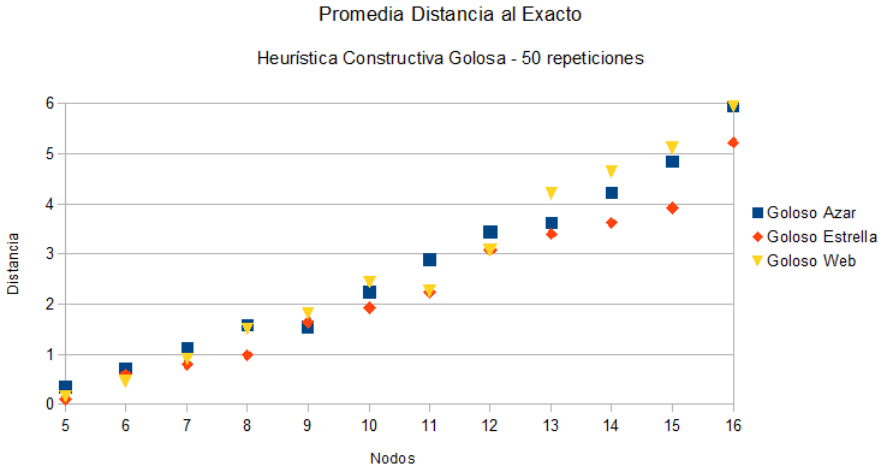
Se ve claro como a medida de que se incrementan los nodos en los grafos la calidad de las soluciones obtenidas empeoran.

Con respecto a la heuristica golosa es la peor de las heuristicas, sin embargo como se vera más adelante es la de menor costo temporal. El empeoramiento (distancia) es lineal a la solución exacta en las 3 familias de grafos.

Con la búsqueda local pasa algo similar salvo que por los datos obtenidos podriamos decir que es prácticamente la mitad de peor que el goloso.

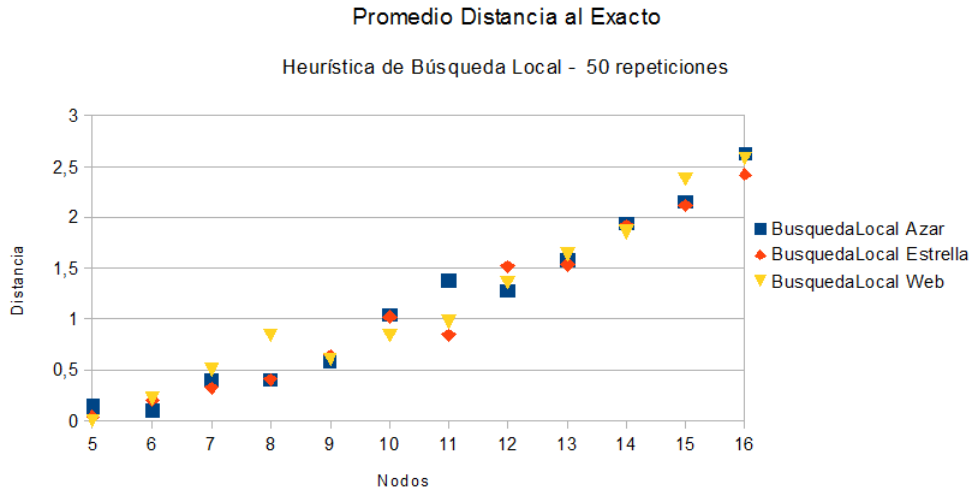
En cambio, con la metaheurística GRASP se obtiene distancias casi nulas con respecto a la solución exacta. Siendo en promedio menor a una para este rango de nodos de grafos que experimentamos (en las 3 familias).

Ahora vamos a mostrar los datos anteriores pero comparando cada heuristica consigo misma fijándonos la distancia con la solución exacta.

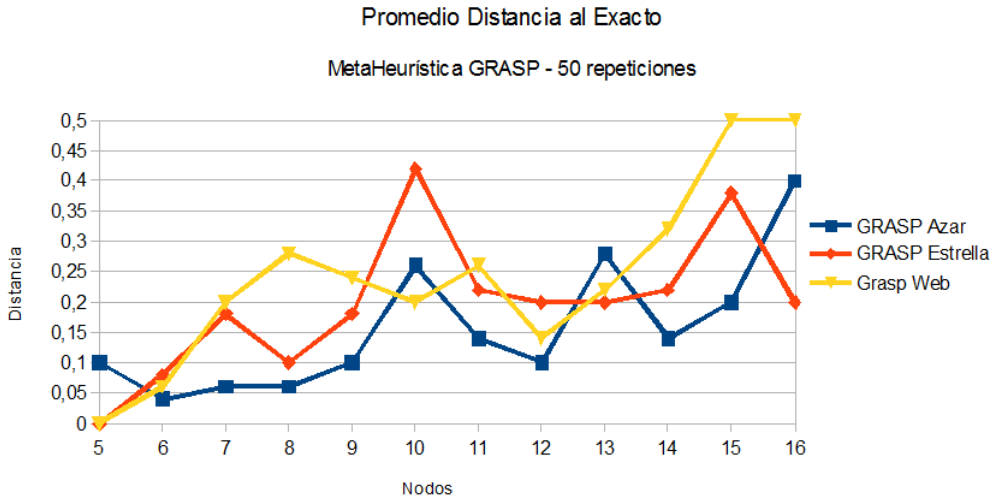


Prácticamente el goloso se comporta igual con los tres tipos de grafos, es decir, que la calidad de las soluciones obtenidas no depende de estas familias de grafos. Podemos determinar que a pesar de la similitud, esta heurística es levemente peor con los grafos Web de

4 vértices.



Vemos un comportamiento menos estable que con el goloso con respecto a las distintas familias de grafos. Sin embargo la tendencia es clara, lineal. No se puede determinar una familia para el cual esta heurística es peor.



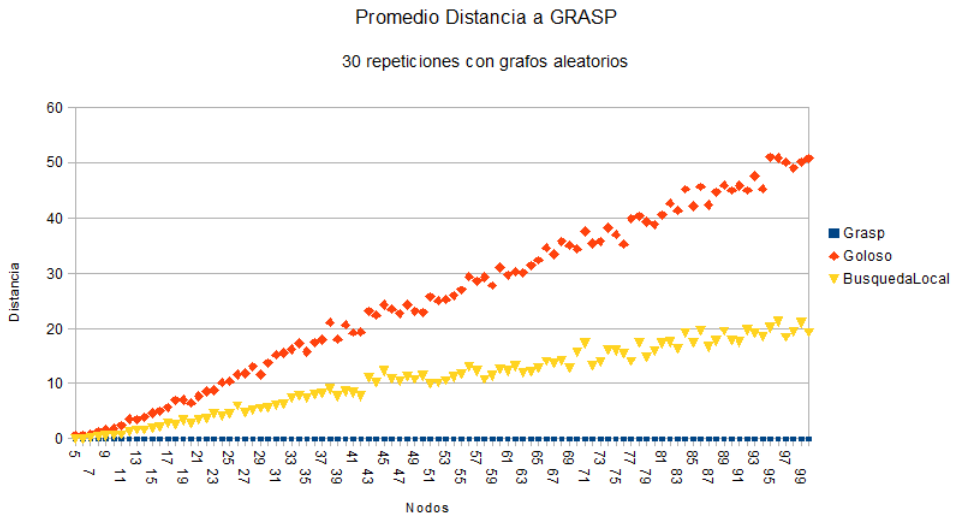
Esta metaheurística es la menos estable de las 3. Varia mucho la distancia, presentando varios picos. Creemos que se debe a los criterios de parada definidos aunque hay que tener en cuenta que el promedio máximo es 0.5 menor a 1. Se podria decir que esta metaheuristica se comporta peor con las familias de grafo Web de 4 vertices porque con los grafos *más grandes* es la que produce mayor distancia de la solución exacta.

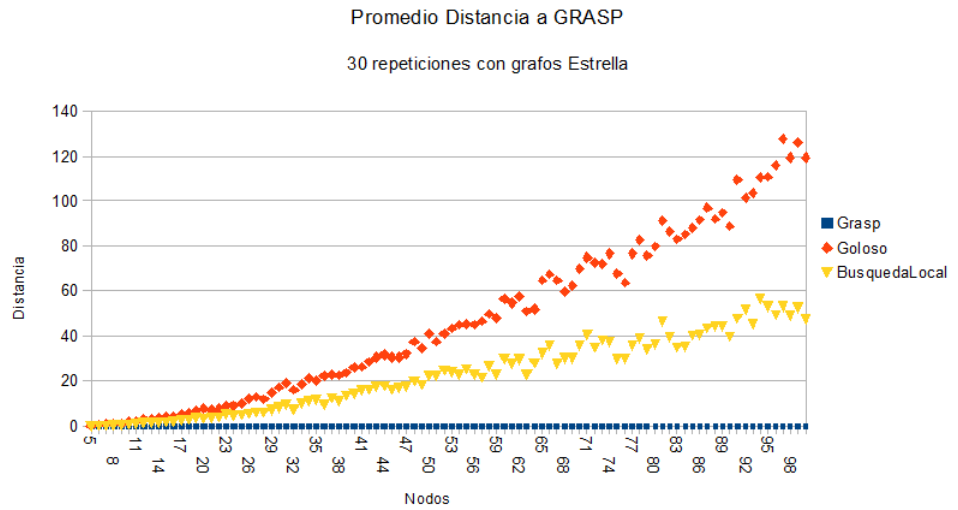
6.2. Comparando con Grasp

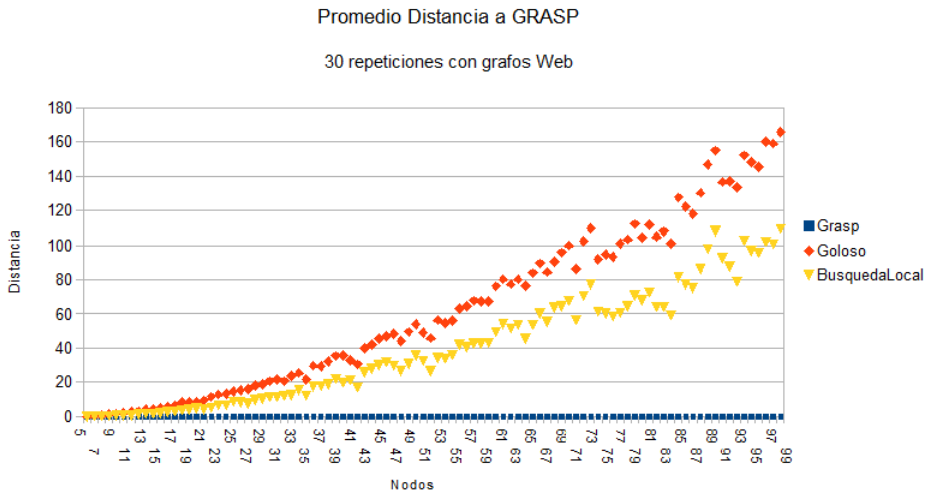
Dejando de lado el algoritmo exacto se puede experimentar con una mayor cantidad de nodos en un tiempo razonable.

Comparamos entre si las heuristicas golosa, búsqueda local y GRASP.

6.2.1. Distancia

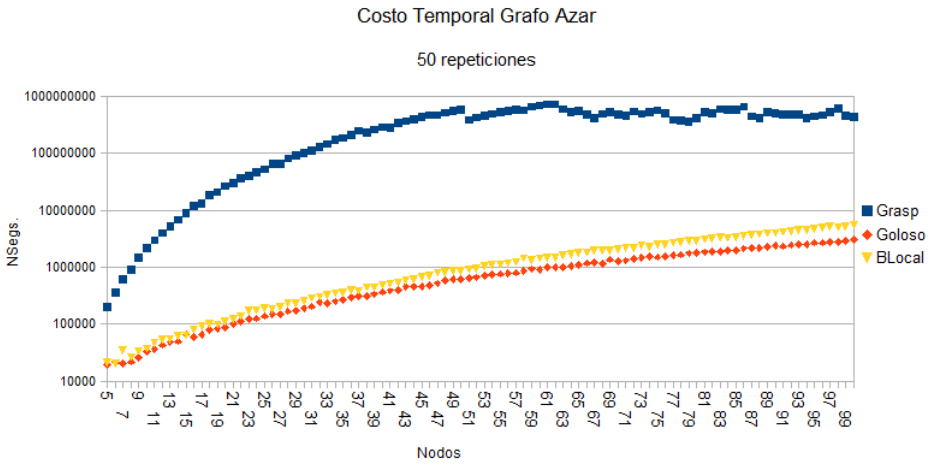


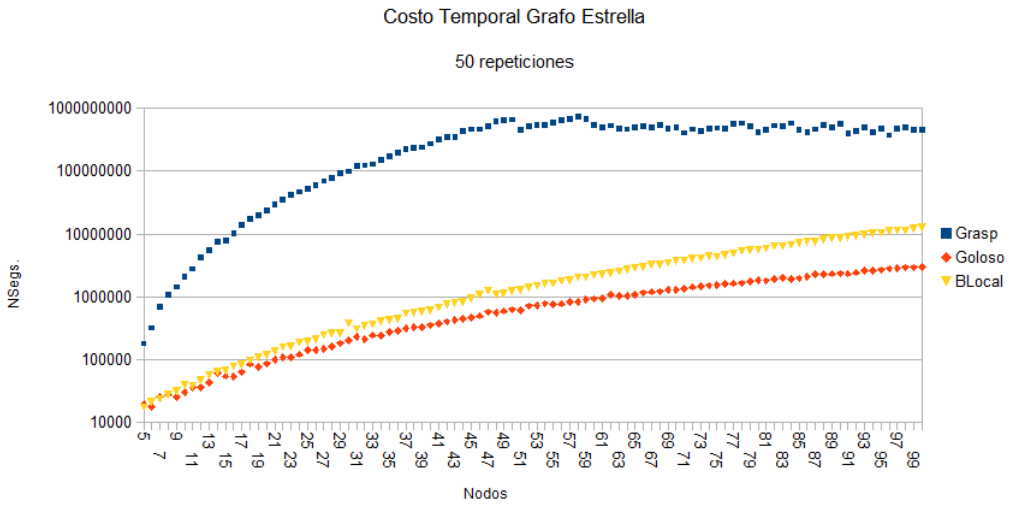


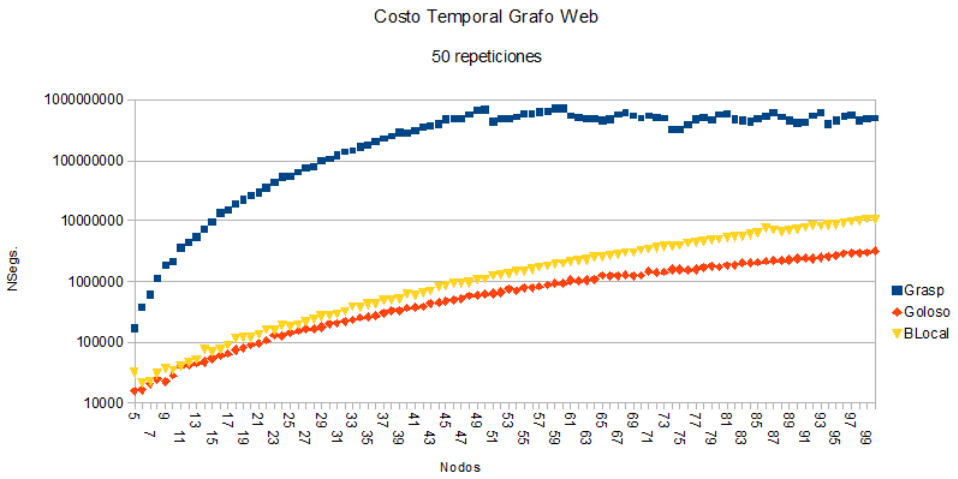


6.2.2. Costo

Algo...







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.