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

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, el mismo caso.

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)

```
vector<unsigned int> res(n + 1)
int res[0] = 0
vector<unsigned int> coloreo(n, 0)
coloreo[0] = 1
colorear(1, g, h, coloreo, res)
return res
```

Algoritmo 2.2

colorear(*unsigned int* nodo, *Grafo* g, *Grafo* h,vector<unsigned int> coloreo,vector<unsigned int> solucion)

```
if (nodo > n)
    int temp = impacto(h, coloreo)
    if temp > solucion[0]
        solucion[0] = temp
        for i desde 0 hasta n
            solucion[i + 1] = coloreo[i]
else
    int maxColor = maximo color usado hasta el momento
    for c desde 1 hasta maxColor
        if es legal pintar el nodo nodo del color c
            vector<unsigned int> nuevoColoreo(coloreo)
            nuevoColoreo[nodo] = c
            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):

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

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

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, double porcentaje)
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
```

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 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

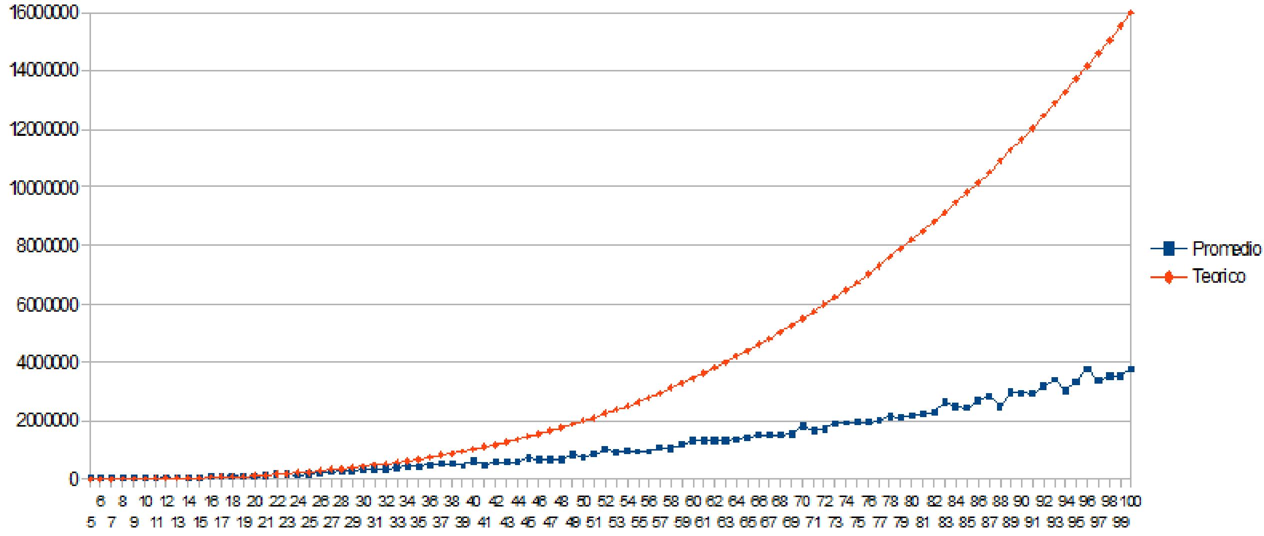

Figura 1: Costos

3.3.2. Grafos densos

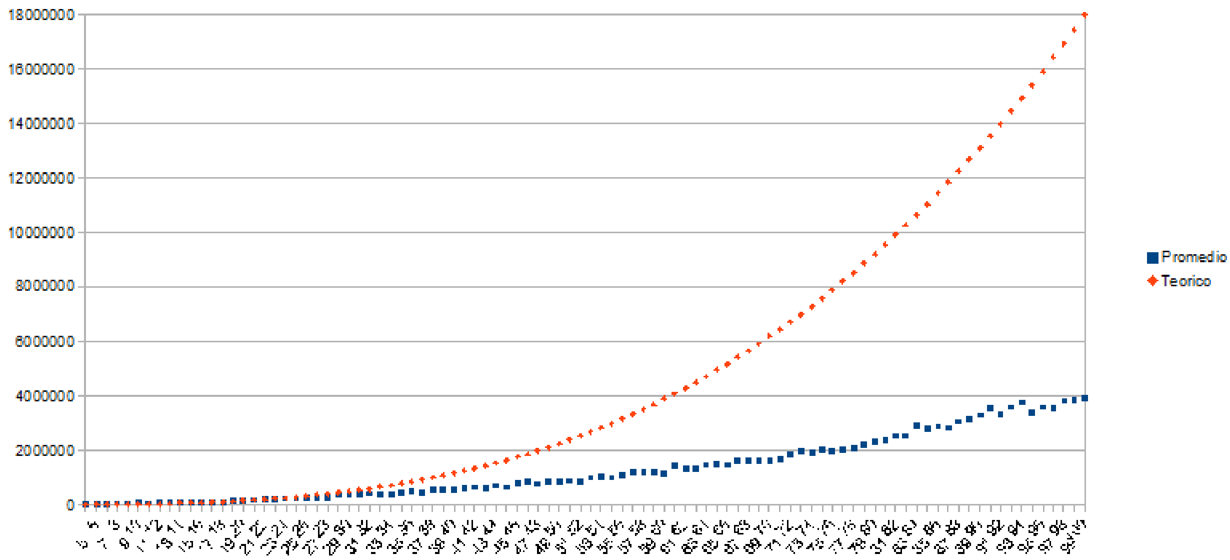

Figura 2: Costos

3.3.3. G y H complementos

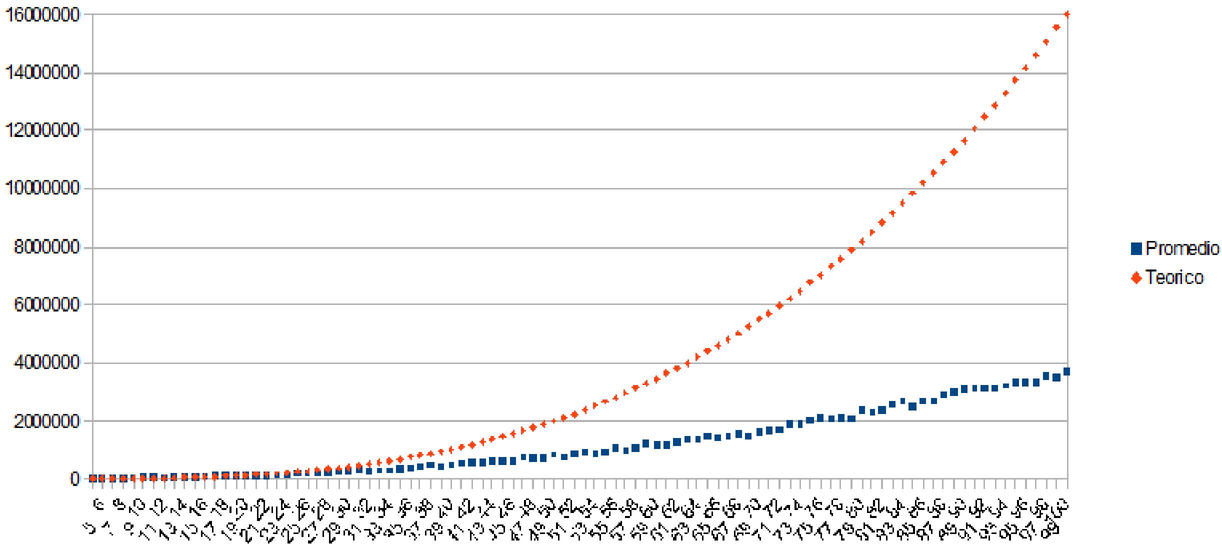

Figura 3: Costos

4. Heurística de Búsqueda Local

4.1. Algoritmo

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);

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

unsigned int nuevoImpacto = 0

for i desde 1 hasta n

    vector<unsigned int> vecinos = vecinosdelnodoienh

    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


impactoGoloso[0] = impactoParcial

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

return impactoGoloso
```

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

5. Metaheurística de Grasp

5.1. Algoritmo

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

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])

```

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

6. Experimentación General