DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico III

Algoritmos y Estructuras de Datos III
Primer Cuatrimestre de 2017

Integrante	LU	Correo electrónico
Ituarte, Joaquin	457/13	joaquinituarte@gmail.com
Oller, Luca	667/13	ollerrr@live.com
Otero, Fernando	424/11	fergabot@gmail.com
Arroyo, Luis	913/13	luis.arroyo.90@gmail.com

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) 4576-3300
http://www.exactas.uba.ar

Contents

1 Introducción

Dado un grafo simple G = (V, E), un subconjunto de vértices de G es una clique si y solo si éste induce un subgrafo completo de G. Es decir K $\subset$ V tal que K $\neq \emptyset$, es una clique de G si y solo si para todo par de vértices u, v $\in$ K, u $\neq$ v, existe la arista u-v en E. Se define la frontera de una clique K como el conjunto de aristas de G que tienen un extremo en K y otro en V $-$ K.

En el siguiente trabajo práctico se resolverá el problema de clique de frontera máxima, que consiste en hallar una clique K de G cuya frontera tenga cardinalidad máxima.

El problema presentado es no polinomial, porque para encontrar la clique K de frontera máxima, necesitamos poder ver todas las cliques que tiene nuestro grafo y a cada una de ellas calcular la cantidad de aristas que están afuera, y esta cuestión de encontrar todas las cliques de un grafo es similar a el problema de encontrar la clique máxima en un grafo que este es un problema no polinomial y por lo tanto nuestro problema también lo es. Así que se resolverá el problema de manera exacta asumiendo que la complejidad sea mala, pero nos va a servir para tener un punto de comparación y luego se realizarán dos heurísticas y una metaheurística. Las heurísticas serán de búsqueda local y golosa, mientras que la metaheurística será taboo search.

El código será realizado integramente en C++, los gráficos serán generados a partir de un archivo de texto y el graficador provisto por la cátedra. Es importante destacar que a medida que se implementaron las funciones, fueron testeadas para verificar su correcto funcionamiento.

2 Parte 1: Aplicaciones en la vida real de CMF.

En términos generales, si modelamos un grafo siendo los nodos representaciones de elementos y las aristas representaciones de alguna propiedad o característica que se cumple para dos elementos, el problema de CMF sirve para encontrar un grupo de estos elementos de forma tal que todos cumplan alguna propiedad que los relacione a todos entre sí, y que además haya una cantidad máxima de relaciones entre dos elementos , uno fuera y otro dentro de este , que cumplan con la propiedad o característica.

2.1 Ejemplo 1:

Por ejemplo, sean los nodos representaciones de países y sean las aristas la representación de que dos países pueden discutir e intercambiar opiniones sin generar conflictos. Los países quieren formar un comité de países en el cual al discutir problemas internacionales no haya discusiones que generen conflictos y retrasen la toma de decisiones para generar una solución. Encontrar la CMF en el grafo que representa a esta situación significa encontrar un comité en el que no ocurran conflictos. Al maximizar a los pares de países (uno dentro y otro fuera del comité) que pueden dialogar, se encontraría la mejor opción para que haya la mayor cantidad de diálogo para encontrar una solución de manera pacífica a cualquier problema internacional que pudiese llegar a suceder.

2.2 Ejemplo 2:

Otro ejemplo sería el modelar como un grafo a un conjunto de personas aspirantes a formar un nuevo grupo para liderar cierto sector de una empresa, un nuevo grupo político o similares, y las aristas entre los nodos representan el apoyo mutuo que existe entre las dos personas. Se quiere formar un grupo de forma tal que en el grupo no haya dos personas que no se apoyen mutuamente para evitar peleas internas, y como la fuerza de este nuevo grupo se mide por la cantidad de personas que apoyan a cada miembro del grupo, se quiere maximizar la cantidad de pares de personas, una fuera y otra dentro del grupo, que se apoyen entre sí. Buscando la CMF se encuentra la clique que representa al mejor grupo de personas que se puede formar para cumplir estas condiciones.

3 Parte 2: Algoritmo exacto.

3.1 Explicación del algoritmo.

Para la resolución del problema de CMF de forma exacta, decidimos implementar un algoritmo en el cual buscamos todas las cliques y entre ellas buscamos la que mayor frontera tenga.

Para obtener esto, buscamos todas las k-cliques en orden de tamaño, es decir, comenzamos buscando las 1-cliques y luego vamos incrementando en 1 el tamaño de las cliques que buscamos en el grafo. Por cada conjunto de k-cliques obtenemos la clique que mayor frontera tenga y luego comparamos a ésta con nuestra mejor clique obtenida hasta el momento. Si es mejor, la k-clique de frontera máxima hallada es guardada como la mejor solución encontrada hasta el momento. Al terminar de buscar todas las posibles cliques, tendremos guardada la CMF y ésta será la solución al problema.

A continuación expondremos el pseudocódigo del algoritmo implementado y serán explicados su funcionamiento y la complejidad que posee cada función.

3.1.1 Pseudocódigo: Función principal.

Este es el método principal del algoritmo. Podemos dividir esta función en distintas partes:

- Primero generamos una solución inicial que va a ser la 1-clique de mayor frontera. Esto se hace simplemente buscando el nodo de mayor grado. Como implementamos nuestro grafo en una matriz de adyacencias, encontrar el grado de un nodo es $O(n)$ por ver todos los nodos si son vecinos o no, y como se realizan para todos los nodos para ver cual es el de mayor grado, la complejidad de esto es $O(n^2)$.
- Luego entramos en un ciclo con un valor i que va desde 2 hasta n, en el que en cada iteración busca la i-clique de mayor frontera y la compara con la clique guardada y entre ellas se guarda la que mejor solución sea. Buscar la i-clique de mayor frontera en cada iteración tiene complejidad $O(2^n * n^2)$, y como se realizan $n - 1$ iteraciones, la complejidad es de $O(2^n * n^3)$.
- Al finalizar el algoritmo, retornamos la solución hallada, la cual al ser obtenida comparando todas las cliques, va a ser la CMF.

La complejidad obtenida para este algoritmo, la cual es la complejidad para resolver el problema de manera exacta, es $O(n^2 + 2^n * n^3) = O(2^n * n^3)$

Algorithm 1 AlgoritmoExacto

```
function AlgoritmoExacto(AdyMatrix AdyMat ) → Solucion
    int n ← size(AdyMat)
    Solucion max ← new Solucion
    max.FronteraSize ← MaximoGrado(AdyMat)                    ▷ O(n^2)
    max.CliqueSize ← 1
    max.clique ← nodoMayorGrado(AdyMat)                       ▷ O(n^2)

    for each i ← desde 2 hasta n, i tamaño de la clique que se está buscando do
        Solucion sol ← MejorSolucionParaKClique(AdyMat,i)     ▷ O(2^n * n^2)
        if (max.FronteraSize < sol.FronteraSize) then
            max ← sol
        end if
    end for                                                   ▷ O(2^n * n^3)
    return max
end function
```

3.1.2 Pseudocódigo: Función para hallar la mejor k-clique.

Esta función la hemos dividido en dos partes: la primera genera todas las k-cliques del grafo, y la segunda busca entre estas cliques a la de mayor frontera. Ambas partes del algoritmo tienen complejidad $O(2^n * n^2)$, por lo que esta función tiene esta misma complejidad.

Algorithm 2 MejorSolucionParaKClique

```
function MejorSolucionParaKClique(AdyMatrix AdyMat, int cliqueSize ) → Solucion
    list<list<nodo > > listaDeCliques ← NuevaListaVacia()
    list<nodo> cliqueActual ← NuevoVectorVacio()
    listaDeCliques ← ObtenerKCliques(AdyMat, cliqueSize, listaDeCliques, cliqueActual, 0)
                                                                          ▷ O(2^n * n^2)
    Solucion sol ← MaximaFronteraEntreKClique(AdyMat,listaDeCliques, cliqueSize)   ▷ O(2^n * n^2)
    return sol
end function
```

3.1.3 Pseudocódigo: Función para obtener todas las k-cliques.

Con esta función obtenemos a todas las cliques de tamaño *cliqueSize*. Lo que realiza este algoritmo es generar dinámicamente todos los subconjuntos de k elementos y luego corrobora que sea o no una clique. Al un nodo poder estar o no en un subconjunto, tenemos dos decisiones distintas por cada nodo, y al haber n nodos, se pueden tener realizar a lo sumo 2^n decisiones distintas, cada una devolviendo un subconjunto de nodos distinto.

Cada llamado recursivo en esta función representa una sucesión de decisiones tomadas, en la cual si esta decisión genera una clique del tamaño deseado, guardamos la clique para devolver luego una lista de todas las cliques deseadas. Para ver si podemos tomar la decisión de agregar a un nodo al conjunto que representa a una clique, vemos que si al agregar al nodo nuevo, el conjunto que se forma genera una clique. Ésto lo realizamos con una complejidad $O(size(cliqueActual)^2)$, que es el costo de ver que para cada nodo de la clique, éste es vecino de los demás nodos de la misma. Como el tamaño deseado de la clique puede llegar a ser n, la complejidad en peor caso de esta función es de $O(2^n * n^2)$.

Algorithm 3 ObtenerKCliques

```
function ObtenerKCliques(AdyMatrix AdyMat, int cliqueSize, list<list<nodo> > cliques,
list<nodo> cliqueActual, int indiceActual)
    if cliqueSize es 0 then
        add(cliques, cliqueActual)
    end if
    for each nodo_j = j-ésimo nodo; indiceActual ≤ j do
        if {nodo_j} ∪ cliqueActual es una clique then                  ▷ O(n^2)
            add(cliqueActual, nodo_j)                                  ▷ O(1)
            ObtenerKCliques(AdyMat, cliqueSize-1, cliques, cliqueActual, j)   ▷ O(1)
            remove(cliqueActual, nodo_j)                               ▷ O(1)
        end if
    end for
end function
```

3.1.4 Pseudocódigo: Encontrar la CMF dentro de un conjunto de k-cliques.

En esta función tomamos un conjunto de k-cliques y buscamos cual de ellas tiene la mayor frontera. El conjunto de cliques de un tamaño k es un subconjunto de partes $\mathcal{P}$(nodos del grafo), y como éste tiene una cantidad de 2^n elementos, nuestro orden puede ser acotado por $O(2^n)$. Luego obtenemos la suma de los grados todos los nodos de la clique. Como a cada nodo le calculamos su grado en $O(n)$, calcular los grados de toda la clique es $O(n * k)$ y como k pertenece a $O(n)$, la complejidad es $O(n^2)$. A este valor le restamos la cantidad de ejes que posee la clique, que al ser un subgrafo completo de k nodos posee $k * (k-1)/2$ ejes, y el resultado va a ser la cardinalidad de la frontera. Todo este algoritmo se realiza entonces con complejidad $O(2^n * n^2)$.

Algorithm 4 MaximaFronteraEntreKClique

```
function MaximaFronteraEntreKClique(AdyMatrix AdyMat, list< list< nodo > > listaDe-
Cliques, int cliqueSize ) → Solucion
    Solucion sol
    int max ← 0
    list < nodo > ← Vacio()
    int gradosInternos ← cliqueSize - 1
    for each clique ∈ listaDeCliques do
        int fronteraSize ← 0
        for each nodo ∈ clique do
            fronteraSize += grado(nodo)
        end for
        fronteraSize-= gradosInternos*cliqueSize
        if fronteraSize > max) then
            max ← externos
            maxclique ← clique
        end if
    end for
    sol.FronteraSize ← max
    sol.cliqueSize ← k
    sol.clique ← maxclique
    return sol
end function
```

3.2 Correctitud

El algoritmo es correcto porque $\forall k/\ 1 < k \leq n$ donde n es la cantidad de vértices del grafo ,el algoritmo lista todas las cliques de tamaño k para el n dado, luego calcula la frontera máxima de estas k cliques y por ultimo se queda con la k clique dentro de todas las que existen, que tenga la frontera mas grande.

3.3 Experimentación

Primero vamos a testear el algoritmo con algunos grafos típicos, para ver el tiempo que tarda el algoritmo en relacion a estos casos.

Por ultimo, vamos a realizar una experimentación para ver que la complejidad es la que suponemos que debe ser. Vamos a realizar el test por medio de correlacionar los tiempos de nuestro algoritmo con una función perteneciente a la clase $O(2^n)$ para esto, vamos a tomar los tiempos según la metodología descripta mas abajo y la función con la cual vamos a correlacionar

3.3.1 Metodología de la experimentación

Para medir el tiempo utilizamos la librería time de C++ midiendo el tiempo en microsegundos,de entrada tomamos grafos completos de tamaño empezando con 5 y fuimos aumentando cada 5 elementos hasta los 135 elementos. El tamaño de entrada es 135 como máximo para la cantidad de vértices porque

para números mayores ya el tiempo pasa a ser insostenible como para poder ser procesado. Para cada instancia, repetimos 10 veces el experimento y tomamos el promedio de tiempos de las 10 repeticiones de esta forma tenemos una estimación mucho más precisa del tiempo real en el que tarda el algoritmo, no tomamos ni el tiempo máximo ni el mínimo debido a los posibles miss o hits de la memoria caché.

3.3.2 Experimento 1:

Para este experimento vamos a tomar 4 familias de grafos distintas en las que suponemos que podemos llegar tener algun resultado interesante, nos vamos a basar en familias:

- **Completo:** Es un grafo K_n de tamaño n.
- **Camino:** Es un camino simple de longitud n.
- **Bipartito:** Es un grafo bipartito de la forma $K_{n/2,n/2}$.
- **Estrella:** Es el grafo en donde todos los vértices se conectan a un mismo vértice.

Optamos por estos 4 grafos porque cada uno, a pesar de tener el mismo tamaño en la cantidad de vértices, tiene una capacidad muy distinta en la cantidad de aristas, por ejemplo el grafo que es un camino tiene n-1 aristas, o el completo tienen la cantidad máximas de aristas que puede tener un grafo.

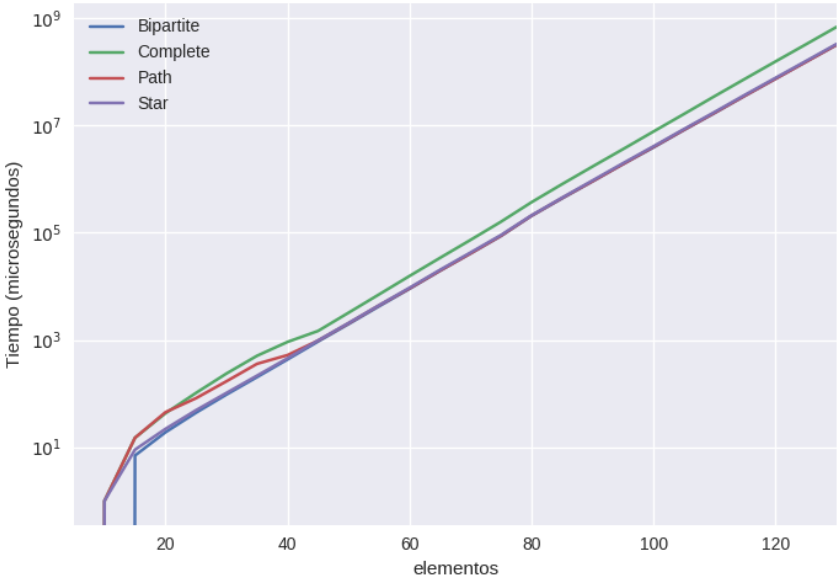

Figure 1: Tiempos de los distintos casos testeados

Según la imagen vemos que los grafos que tienden a tener menor cantidad de aristas como el grafo de camino, el bipartito y el estrella, tienden a tener menor tiempo de ejecución que el grafo completo, esto se debe a que, al tener menor cantidad de aristas, estos grafos poseen menos cliques a las cuales les tengo que calcular la frontera.

3.3.3 Experimento 2:

Vamos a corroborar que la complejidad es exponencial. Para esto tomamos todas las curvas y las comparamos contra una curva exponencial $(2^n * n^3)$ para calcular el coeficiente de Pearson. La comparación nos arroja un coeficiente de Pearson de 1, corroborando que la complejidad del problema es la descripta. El gráfico a continuación corresponde al Pearson del grafo completo Kn.

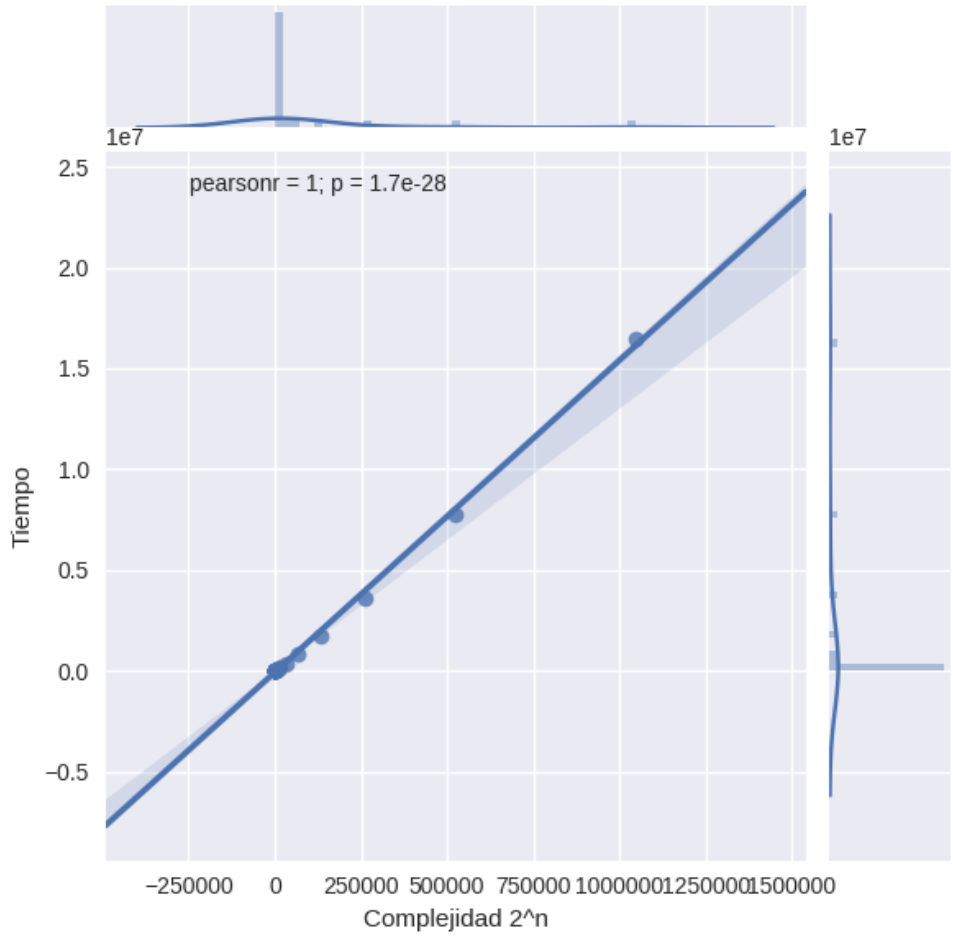

Figure 2: Correlación de nuestros datos vs $f(n) = 2^n * n^3$

4 Parte 3: Heurística constructiva golosa

4.1 Resolución del problema

Debido a la elevada complejidad de resolver el problema en forma exacta, se implementó una heurística golosa. Dado un grafo, el algoritmo implementado toma cada 1-clique y aumenta esa clique de forma golosa, probando agregar cada nodo del grafo según el grado de dicho nodo, de forma decreciente. Los nodos se agregan si al hacerlo se forma una clique y la cardinalidad de la frontera aumenta. El paso de agregar un nodo a la clique K_i sobre la cual se esté iterando se puede escribir con la fórmula:

$$f(K_i, e) = \begin{cases} K_i \cup \{e\} & si \quad esClique(K_i + e) \wedge \delta(K_i \cup \{e\}) > \delta(K_i) \\ \\ K_i & sino \end{cases} \tag{1}$$

Guardándonos la clique de mayor frontera hallada hasta el momento, una vez que terminemos el algoritmo retornaremos la clique con mayor frontera que se encontró en el procedimiento.

4.2 Pseudocódigo

La función esClique toma la matriz de adyacencia, una clique y un entero e y devuelve true si hay una arista entre e y cada nodo de la clique.

La función grado(A,e) devuelve el grado del nodo e segun la matriz de adyacencia A.

Algorithm 5 hGolosa

```
function hGolosa(AdyMatrix A, int n ) → Solucion
    Orderar los nodos según su grado                                  ▷ O(n^2)
    maxF ← 0
    vector<int> maxClique
    for each e ∈ V(A) do                                              ▷ O(n)
        vector<int> clique;
        add(clique, e);
        F ← grado(A,e);                                               ▷ O(n)
        for each v ∈ V(A) de mayor grado a menor grado do
            if esClique(A, clique, v) then                            ▷ O(n)
                if F < F + grado(A,v) - 2* clique.size() then         ▷ si tiene una frontera más grande
                    F = F + grado(A, v) - 2* clique.size();
                    add(clique, v);
                end if
            end if
        end for
        if maxF<F then
            maxF=F
            maxClique= copy(clique)
        end if
    end for
    Solucion s;
    s.F = maxF;
    s.k = maxClique.size();
    s.v = maxClique;
    return s
end function
```

Para averiguar si la frontera aumenta al agregar el nodo e, utilizamos la siguiente formula:

$$\delta(K_i \cup \{e\}) = \delta(K_i) + gr(e) - 2 * i$$

La nueva frontera es igual a la frontera de K_i + el grado de e menos las aristas compartidas por K_i y e (que aparecen tanto en el grado de e como en la frontera de K_i).

4.3 Complejidad

Notando a "n" como la cantidad de nodos del grafo, a la complejidad del algoritmo la podemos analizar por partes:

- **Ordenar nodos:** Utilizando una técnica de ordenamiento simple ordenamos los nodos en complejidad temporal $O(n^2)$.
- **For interno:** En este ciclo podemos usar hasta n nodos, y a cada nodo analizamos si puede ser agregado o no a la clique. Esto tiene tiempo $O(n)$ por cada nodo, por lo que se obtiene una complejidad $O(n^2)$.
- **For externo:** En este ciclo analizamos cada 1-clique del grafo, es decir, n cliques. Por cada clique realizamos el ciclo interno, por lo que se tienen n cliques de tiempo cuadrático cada una, obteniéndose un tiempo del orden $O(n^3)$. Al terminar el ciclo interno, asumiendo que cada iteración obtiene una clique con mayor frontera, copiar la clique es copiar hasta a lo sumo n nodos, por lo que se tiene una complejidad lineal por cada 1-clique. De esta forma obtenemos finalmente la complejidad $O(n^3 + n^2) = O(n^3)$.
- **Retornar la solución:** Retornar la solución es retornar una estructura que posee dos enteros (tamaño de la frontera y de la clique), y la clique, cuyo tamaño puede ser de hasta n nodos. Por lo tanto la complejidad de esto resulta ser $O(n)$.

Luego, sumando las complejidades obtenemos:

$$O(n^2) + O(n^3) + O(n) = O(n^2) + O(n^3) = O(n^3)$$

4.4 Soluciones no optimas

Como explicamos previamente el algoritmo ordena todos los nodos según el grado, en orden decreciente. No solo eso, sino que prueba comenzar con todos los nodos. Esto significa que para que el algoritmo falle, para todo nodo con el que empieza debe encontrar un nodo para ampliar la clique solución, que no pertenezca a la solución exacta. Esto puede darse si para cada nodo su vecino de mayor grado no forma parte de la CMF, ya que el algoritmo avanzaría sobre una 2-clique que no forma parte de la solución exacta y por ende ninguna clique que forme a partir de ella lo será.

Con esta idea, creamos una familia de instancias que se generan partiendo de un grafo K_n. Todos los nodos de éste grafo van a tener grado $n-1$ y la CMF estará incluida en éste. Para que nuestro algoritmo no encuentre esta solución, conectamos cada nodo $n_i \in K_n$ a un nuevo nodo n_j (uno distinto por cada n_i). Ahora el grado de cada n_i es n y el de cada n_j es 1. Nuestro objetivo va a ser que el goloso al elegir cualquier nodo, termine eligiendo a un nodo n_j formando una 2-clique y luego no pueda agregar otro nodo. Para ésto tenemos que hacer que los nodos n_j tengan un grado $n+1$, lo que resolvemos con agregarle a cada n_j, n nuevos nodos (los llamaremos n_k). De esta forma cuando se aplique el algoritmo, suceden tres situaciones:

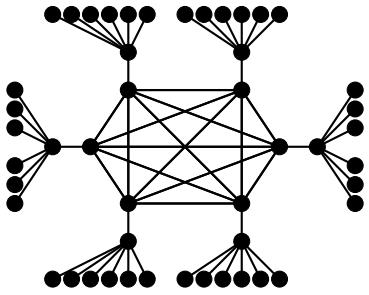

Figure 3: Ejemplo a partir de un K_6.

- **El algoritmo comienza con un nodo n_k:** Al comenzar con uno de estos nodos, el único vecino que tiene es un nodo n_j, por lo que lo utiliza para formar una 2-clique y no puede generar otra clique más grande. La cardinalidad de la frontera que se genera es de n, que es el grado del nodo n_j restada la arista usada para conectar con n_k.
- **El algoritmo comienza con un nodo n_i:** Éste nodo tiene como vecinos a un n_j de grado $n+1$ y a todos los demás nodos de K_n de grado n. Luego el algoritmo utiliza al nodo n_j y forma una 2-clique. Como los vecinos de n_j no son vecinos de n_i y a la inversa sucede lo mismo, no se puede generar otra clique más grande. Luego la cardinalidad de la frontera que se forma son los $n-1$ ejes que conectan a los nodos de K_n con n_i y los n ejes que conectan a los nodos n_k con n_j, obteniéndose una cardinalidad de $2 * n - 1$.
- **El algoritmo comienza con un nodo n_j:** Como los vecinos de este nodo son los n_k de grado 1 y un nodo n_i de grado n, el algoritmo utiliza al nodo n_i. Luego se forma la misma situación que en el caso anterior, formando una frontera de cardinalidad $2 * n - 1$.

En este grafo generado, se pueden formar 3 tipos de soluciones. Una es la 2-clique que utiliza los nodos n_k y n_j, que como explicamos tiene frontera de cardinalidad n. El segundo tipo de solución es la 2-clique que forman los nodos n_j y n_k, que como también se explicó, tiene frontera de cardinalidad $2 * n - 1$. El último tipo es la CMF del grafo k_n. La CMF de este grafo va a ser la clique que cumple que $n_a * n_b + n_a$ sea máxima, siendo n_a la cantidad de nodos de la clique y n_b la cantidad de nodos de k_n que no están en la clique, representando la multiplicación la cardinalidad de la frontera de la clique considerando solo a k_n y el término $+n_a$ son los ejes que conectan a los n_a nodos con los nodos n_j. Éste valor va a ser máximo para cuando la clique tenga la mitad de los nodos de k_n, obteniéndose $\lceil \frac{n}{2} \rceil * \lfloor \frac{n}{2} \rfloor + \lceil \frac{n}{2} \rceil$.

Luego esta familia de grafos devuelve una solución mala en el algoritmo goloso cuando $2 * n - 1 < \lceil \frac{n}{2} \rceil * \lfloor \frac{n}{2} \rfloor + \lceil \frac{n}{2} \rceil$. Cuando n es par: $2 * n - 1 < \frac{n}{2} * \frac{n}{2} + \frac{n}{2} \leftrightarrow 0 < (n - (3 + \sqrt{5}))(n - (3 - \sqrt{5}))$. Y cuando n es impar: $2 * n - 1 < \frac{n+1}{2} * \frac{n-1}{2} + \frac{n+1}{2} \leftrightarrow 0 < (n-1)(n-5)$. Esto nos indica que la familia devuelve malas instancias cuando $6 \leq n$ para ambos n par e impar.

4.5 Experimentación

4.5.1 Experimento 1:

Para tener una idea de que tan bueno es aplicar una heurística golosa al problema, creamos un algoritmo que genera 500 grafos aleatorios para cada valor de n. En cada uno de estos grafos se calcula la cantidad de aristas de forma aleatoria como n ≤ m ≤ n*(n-1)/2. El resultado fue que para n < 13 la heurística siempre encontró el resultado exacto para los grafos que se generaron, y da buenos resultados en n < 25 (diferencia entre el goloso y el exacto menor a 2).

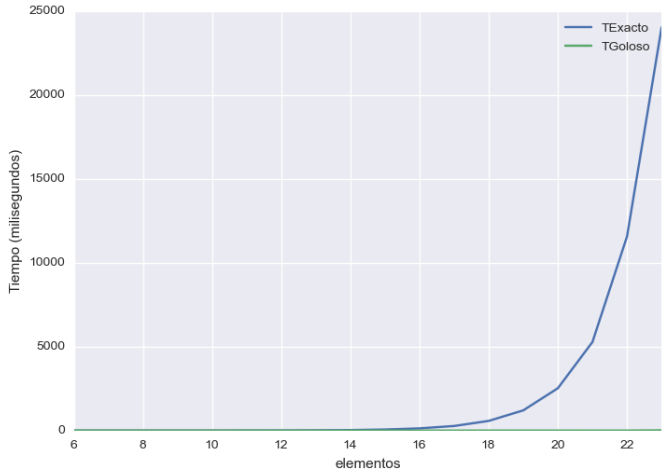

Figure 4: Comparación del tiempo de cómputo goloso vs exacto

El gráfico nos muestra que que el tiempo para el algoritmo exacto siempre es mucho mayor, y por mucho, que el goloso. Además, para los casos en los que pudimos comparar el tiempo del algoritmo goloso vs el exacto, el goloso se encuentra completamente " aplastado " porque la velocidad es muchísimo mas rápida que casi todo el exacto en el resto de las iteraciones, no lo graficamos en escala logarítmica porque nos encontramos con el mismo resultado que este pero con la diferencia que la curva de elementos desaparece, esto se debe que cuando medimos el tiempo con la libreria time de c++ el goloso se resolvía mucho antes de un milisegundo en cambio el exacto se volvía prácticamente interminable. Como no podemos indagar nada sobre el tiempo del algoritmo goloso ya que se resuelve instantáneamente, , vamos a usar pearson para corroborar correctitud del algoritmo goloso contra la complejidad teórica la cual es $O(n^3)$.

Para el cálculo de la correlatividad entre el tiempo del algoritmo goloso vs la función $O(n^3)$ corrimos el algoritmo independientemente promediando el tiempo por cada repetición con saltos cada 5 elementos hasta el tamaño de 500 vértices.

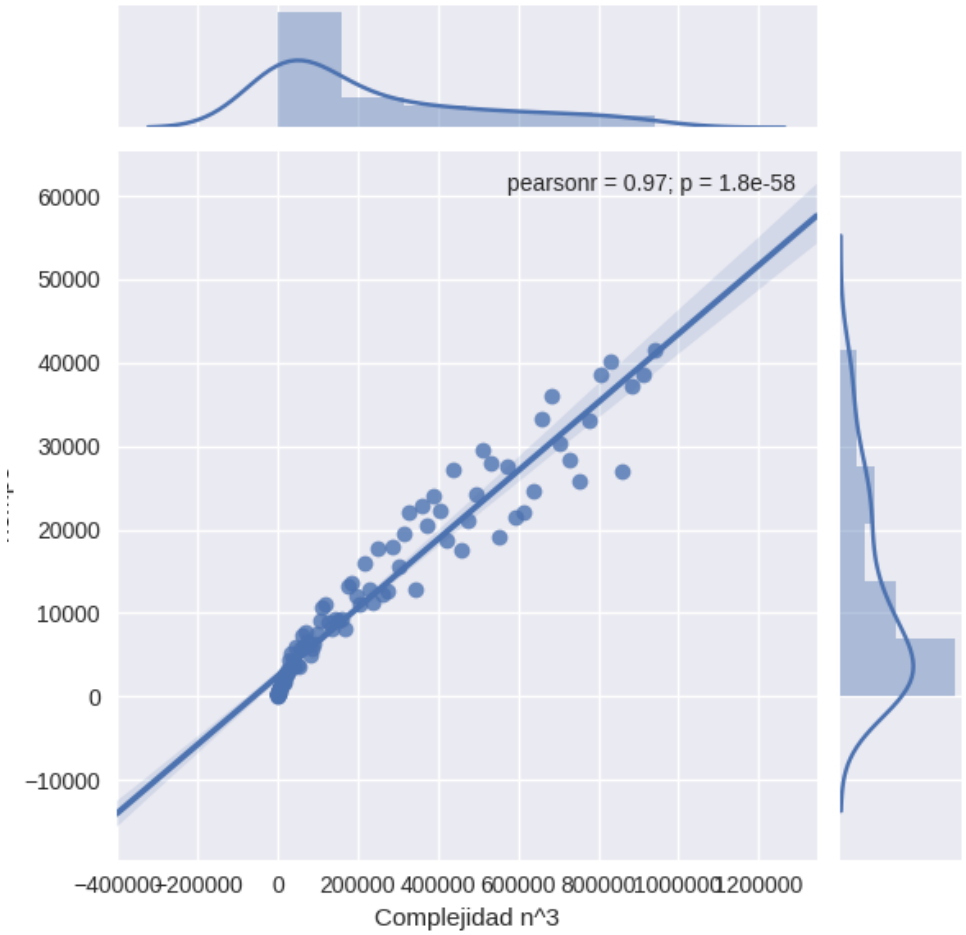

Figure 5:

Luego de probar con otras funciones polinomiales potencia distinta a 3, todas las cuales no pertenecen a $O(n^3)$ nos han dado un coeficiente menor a 0,97 por eso podemos corroborar que la complejidad es correcta y que, como se ve en nuestro gráfico, el coeficiente no es 1 porque los tiempos de computos para los distintos n tienen una varianza bastante grande, pero tomando el todo del experimento, es una muy buena aproximación.

4.5.2 Experimento 2:

Como siguiente experimento, comparamos la calidad de las soluciones dentro del rango que nos permite evaluar el algoritmo exacto debido a su complejidad. En base a 500 grafos generados aleatoriamente por cada n, tomamos dos valores, el promedio de la diferencia entre el goloso y el exacto y la máxima obtenida de la cardinalidad de las fronteras.

El resultado de esto fue que para los grafos aleatorios el algoritmo goloso es muy eficiente respecto del exacto. La solución dista a lo sumo en 1 arista en la frontera de error contra el exacto en promedio, y a lo sumo en todo el experimento lo más lejos que erró respecto al exacto fue de 3 aristas. Estas diferencias entre la solución del goloso y el exacto aparecen con $n > 13$ hasta $n < 25$, donde el algoritmo exacto tarda más de un día en dar una solución. Deducimos de estos datos que nuestro algoritmo goloso es eficiente.

4.5.3 Experimento 3:

Un contraejemplo para nuestro algoritmo es el descripto previamente en la sección de solución no óptima. Con este contraejemplo vamos a medir el promedio de la diferencia que hay entre la solucion del goloso y la del exacto para todo n y el máximo alcanzado en todas las pruebas para cada n. Éstos grafos van a ir aumentando sus nodos de manera cuadrática por la forma en la que construimos el grafo a partir de un grafo completo, debido a que este grafo es muy particular y fue creado específicamente para que el algoritmo falle lo más posible.

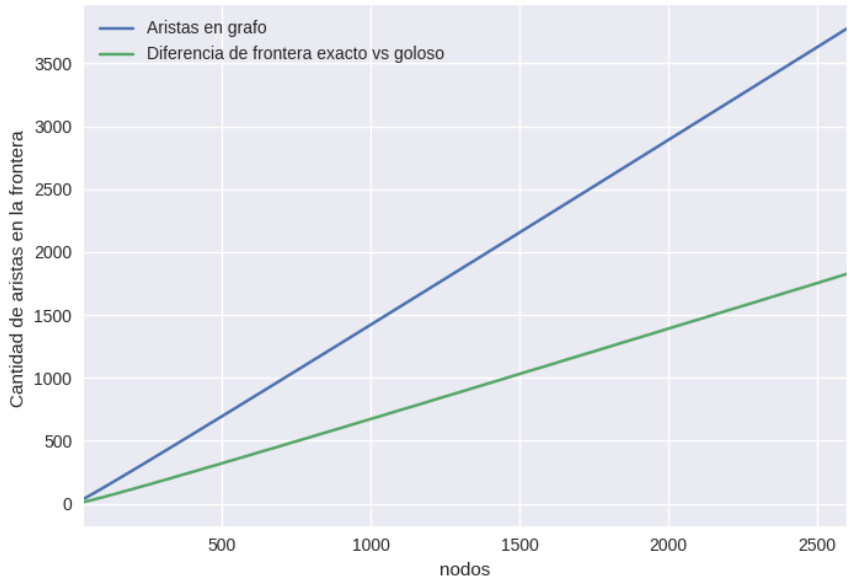

Figure 6: Caso no óptimo para el algoritmo goloso.

Para este tipo de grafo, el error que tiene nuestro algoritmo goloso es siempre el mismo para cada tamaño n por ser construido en forma determinista. Además se puede observar en los resultados el incremento lineal respecto a la cantidad de nodos que posee el error de la solución obtenida.

5 Parte 4: Heurística de Búsqueda Local

5.1 Introducción

Se nos pide implementar una heurística de búsqueda local para el problema planteado en el ejercicio 1. Además queremos analizar la calidad de solución y el tiempo de cómputo en comparación con el algoritmo exacto.

Una heurística local consiste en plantear una vecindad de una solución y encontrar la mejor solución dentro de ella. Una vecindad es un conjunto de soluciones con la particularidad de que varían muy poco con respecto a la solución inicial.

Puede haber casos en los que la heurística local no encuentre una mejor solución, esto se debe a que la solución inicial es un máximo local.

Por lo tanto, los puntos importantes de una heurística local son encontrar una solución inicial, plantear una vecindad y encontrar la mejor solución dentro de esa vecindad.

5.2 Resolución del problema

El algoritmo recibe una solución del problema generado por la heurística golosa, e intenta mejorarla. La vecindad elegida es el intercambio de un único nodo de la solución recibida por un nuevo nodo del grafo, siempre que siga siendo una clique y aumente la frontera. Si se encuentra este nodo a intercambiar, luego se revisa si se pueden agregar mas nodos, generando una clique mas grande de mayor frontera. Esto podría no hacerse y dejar la vecindad sólamente como el intercambio de 2 nodos, pero siendo que al hacerlo uno puede conseguir una solución incluso mejor sin aumentar la complejidad del algoritmo (ver sección Complejidad), y no arruina ningún calculo en el algoritmo Tabú desarrollado en el punto siguiente, no tiene sentido no hacerlo.

Vecindad:
La vecindad propuesta es la de reemplazar algún nodo de la clique solución, por uno o mas nodos. El nodo a reemplazar es elegido cuando encontramos un nodo que no pertenece a la clique y que, al reemplazarlo, forma una clique de mayor frontera. Luego verificamos si se pueden agregar nodos, y si se puede los agregamos a la clique. Los nodos a agregar se testean ordenados por grado, en orden decreciente. Se prueba intercambiar todos los nodos con todos y nos quedamos con el reemplazo que devuelva la frontera mas grande.

5.3 Pseudocódigo

En este caso utilizamos, además de la función *esClique* utilizada en el algoritmo goloso que recibe 3 parámetros, la función *esClique* que recibe 4 parámetros: esClique(MatrizAdy ady, Clique clique, Nodo e, Nodo ignorar) recibe la matriz de adyacencia, la clique, el elemento a evaluar si esta conectado a todos los elementos de la clique y un ultimo elemento que si pertenece a la clique, sera ignorado. La diferencia entre las funciones es que la primera, que utiliza 3 parámetros (MatrizAdy, Clique, Nodo), se fija si la clique con el nodo sigue formando una clique. En cambio la segunda, que recibe 4 parámetros, se fija si al intercambiar en la clique a ambos nodos sigue formando una clique.

Tanto en esClique de 3 como de 4 parámetros, dará falso si el nodo "nuevo" pertenece a la clique, por lo que no debemos preocuparnos de agregar un elemento repetido.

Algorithm 6 hLocal

```
function hLocal(AdyMatrix A, Solucion sol, int n ) → Solucion
    Orderar los nodos según su grado                                          ▷ O(n^2)
    maxF = sol.FronteraSize
    list<nodo> maxClique ← sol.clique
    do
        prevF ← maxF;
        for each e ∈ sol.clique do                                            ▷ O(sol.cliqueSize)
            for each v ∈ V(A) de mayor grado a menor grado do                  ▷ O(n)
                if esClique(A, sol.clique, v, e) then        ▷ me fijo si es clique   ▷ O(n)
                    gr_e ← grado(A,e);
                    gr_v ← grado(A,v);
                    if sol.FronteraSize < (sol.FronteraSize - gr_e) + gr_v then   ▷ y si tiene una frontera
mas grande
                        auxF ← sol.FronteraSize - gr_e + gr_v;
                        cliqueAux ← sol.clique;
                        swapNodos(cliqueAux, e, v)
                        for each p ∈ V(A) de mayor grado a menor grado do      ▷ O(n)
                            if esClique(A, cliqueAux, p) then                 ▷ O(n)
                                if auxF < auxF + grado(A,p) - 2* clique.size() then
                                    auxF = auxF + grado(A, p) - 2* cliqueAux.size();
                                    add(cliqueAux, p);
                                end if
                            end if
                        end for
                        if auxF > maxF then
                            maxF ← auxF
                            maxClique ← cliqueAux
                        end if
                    end if
                end if
            end for
        end for
    while prevF < maxF
    Solucion res;
    res.FronteraSize ← maxF;
    res.cliqueSize ← maxClique.size();
    res.clique ← maxClique;
     return res;
end function
```

5.4 Complejidad

La complejidad obtenida por el algoritmo, sin contar el ciclo mientras mejore, es la siguiente:

- **Ordenar los nodos:** Ordenamos los nodos con un algoritmo de ordenamiento sensillo, de complejidad $O(n^2)$.
- **En el ciclo principal que itera por cada e en la solución de entrada:** Tenemos un ciclo interno en el que por cada nodo v del grafo, chequeamos si swapear e con v mejora la solución con tiempo lineal a la cantidad total de nodos para verificar que es una clique y averiguar el grado de v para calcular la nueva frontera.

  Luego de realizar el swapeo, todavía dentro de los ciclos, intentamos agregar de forma golosa nodos a la nueva clique. Como se pueden agregar $O(n)$ nodos a la clique, y por cada uno verificamos si forma clique y mejora la solución (como se explicó en el párrafo anterior, con costo lineal), el costo obtenido de realizar esto es de $O(n^2)$.

  Finalmente, la complejidad total del ciclo es de:
  $O(IterCicloPrincip * IterCicloInter * (CheckSwap + IncrementarSolucionSwap))$.

  Como $sol.cliqueSize$ puede llegar a ser el grafo completo, la cantidad de iteraciones del ciclo principal es $O(n)$. Además, las iteraciones del ciclo interno son $O(n)$ y luego obtenemos la complejidad $O(n * n * (n + n^2)) = O(n^4)$.

Finalmente nos falta ver cuántas iteraciones se hacen por mejorar la solución. Como no podemos demostrar un mínimo y un máximo de frontera que encuentran los algoritmos, acotamos (aunque en la práctica de forma grosera) estos valores por 0 y por F, siendo F la cardinalidad de la frontera más grande que se pueda formar para cualquier grafo de n nodos. Como tampoco podemos demostrar cuanto mejora de frontera cuando lo hace el algoritmo, acotamos nuevamente con un incremento de 1 elemento a la nueva cardinalidad de la frontera, pudiendo así incrementarse hasta F veces por ser éste el máximo.

Nuevamente podemos volver a acotar el valor F en función de n. Utilizando el grafo completo, ya que cualquier otro grafo del mismo tamaño es subgrafo de éste y por ende la máxima frontera del grafo completo va a ser igual o mayor a la de cualquier subgrafo, podemos acotar este valor para K_n. Sea C la CMF con $n_1 \leq n$ nodos, la frontera va a ser todos los ejes entre los nodos de C y los $n_2 = n - n_1$ nodos fuera de C, es decir, $n_1 * n_2$. Como n_1 y n_2 son ambos del orden $O(n)$, la cardinalidad de la frontera de C va a ser del orden $O(n^2)$.

Finalmente, la complejidad del algoritmo es de $O(n^2 * (n^2 + n^4)) = O(n^6)$.

5.5 Experimentación

Para los siguientes experimentos, los tiempos fueron promediando cada instancia ejecutándose 100 veces. En el caso en el que se utilizaron grafos aleatorios, por cada repetición se utilizó un grafo distinto y se realizan 500 diferentes por cada tamaño de entrada.

5.5.1 Experimento 1: Instancias aleatorias

Hemos puesto a correr grafos aleatorios en donde aplicamos nuestra heurística golosa y la de búsqueda local, y nos hemos encontrado con resultados similares al caso anterior, en otras palabras por más que hagamos búsqueda local luego del algoritmo goloso, el tiempo necesario para calcular el algoritmo exacto es abismal comparado contra estos otros dos.

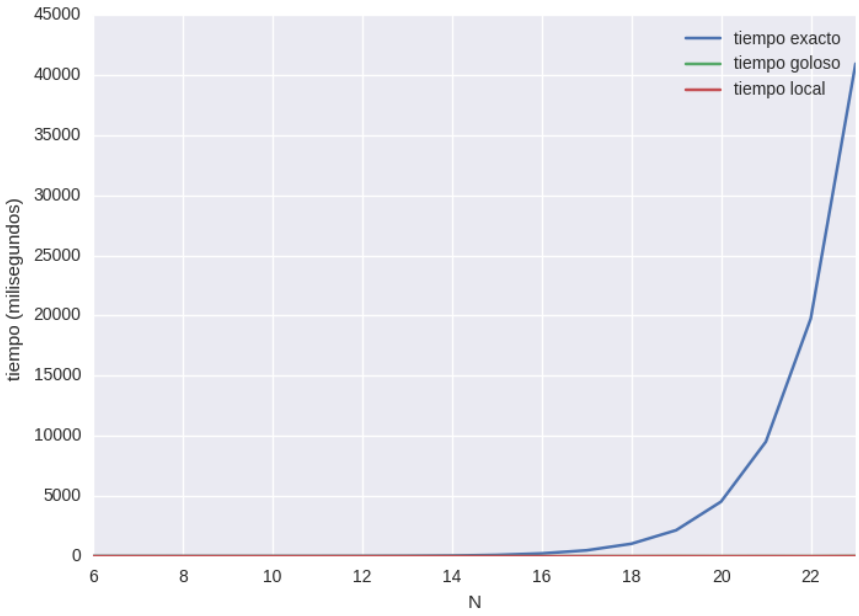

Figure 7: Tiempos local, goloso y exactos.

En el siguiente gráfico mostramos los tiempos de ejecución de los algoritmos local y goloso sin el exacto y en una mayor cantidad de tamaños de instancias. Se puede apreciar que la diferencia de tiempos es leve y tienen comportamientos similares. Esto se debe a que el algoritmo de búsqueda local realiza pocas búsquedas que mejoran el algoritmo, por lo que no se realizan demasiadas operaciones luego de la primer solución obtenida con el goloso.

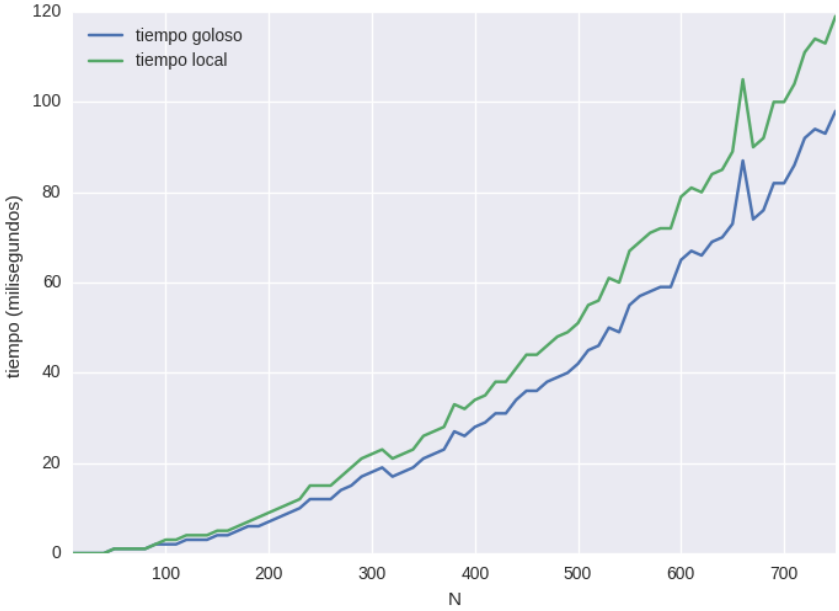

Figure 8: Tiempos local y goloso.

Para los mismos grafos a los que se les midió el tiempo, también se les analizaron los resultados obtenidos. En el siguiente gráfico mostramos en promedio cuanto incrementa el tamaño de la frontera la búsqueda local respecto a la solución obtenida con el algoritmo goloso (curva promGL) y también cuanto fue el máximo que mejoró para algún grafo de determinado tamaño (curva maxGL).

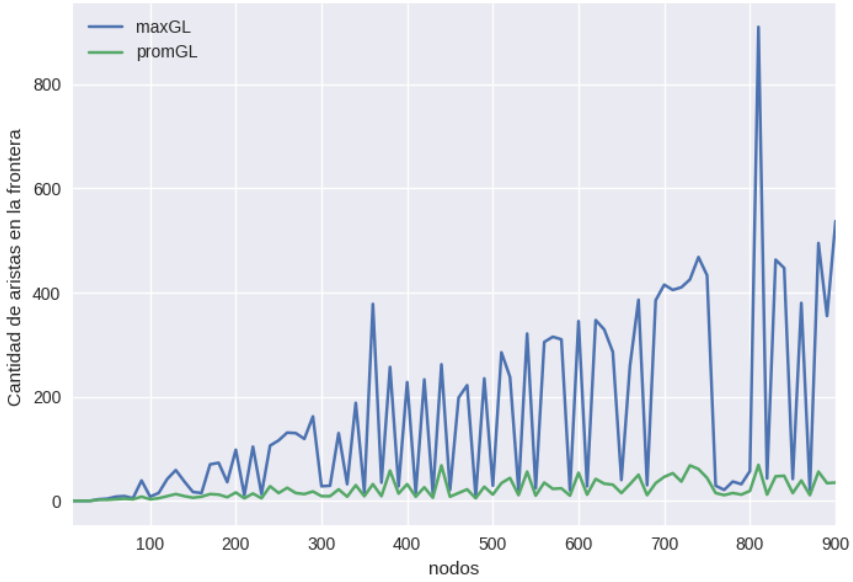

Figure 9: Mejoras de la solución usando búsqueda local.

Se puede observar que en promedio la búsqueda local mejora a la solución que devuelve el goloso, y esta mejora como se observa en los valores máximos, puede ser muy alta a comparación del promedio.

5.5.2 Experimento 2: Casos no óptimos.

Como la vecindad es reemplazar un solo nodo (y agregar todos los que se pueda luego), no podremos visitar las soluciones donde se requiera quitar mas de un nodos a la vez. Los casos malos explicados en el algoritmo goloso, también van a ser casos malos aquí. Esto se debe a que el goloso encuentra una clique con el nodo n_j que conecta con el subgrafo completo. Utilizando solo un swap de nodos es imposible mejorar la solución, debido a que como la frontera máxima que encuentra es de a lo sumo $2n - 1$, debe swapear el nodo n_j con algún nodo del subgrafo completo, pero ésto genera una frontera de tamaño $2 * (n - 2) + 2 = 2n - 2$, $(n - 2)$ por cada uno de los dos nodos de k_n mas un grado a cada uno por el eje hacia el nodo n_j de cada uno), lo cual es menor y por ende no cambia la solución.

A estos grafos les medimos el tiempo de ejecución y las calidades obtenidas, promediando 500 repeticiones de cada grafo. Nos encontramos con un panorama muy similar al ejercicio anterior, donde las soluciones no mejoraron (se superponen las curvas en el gráfico) y los tiempos de ejecución, a pesar de que el algoritmo goloso es más rápido, son similares.

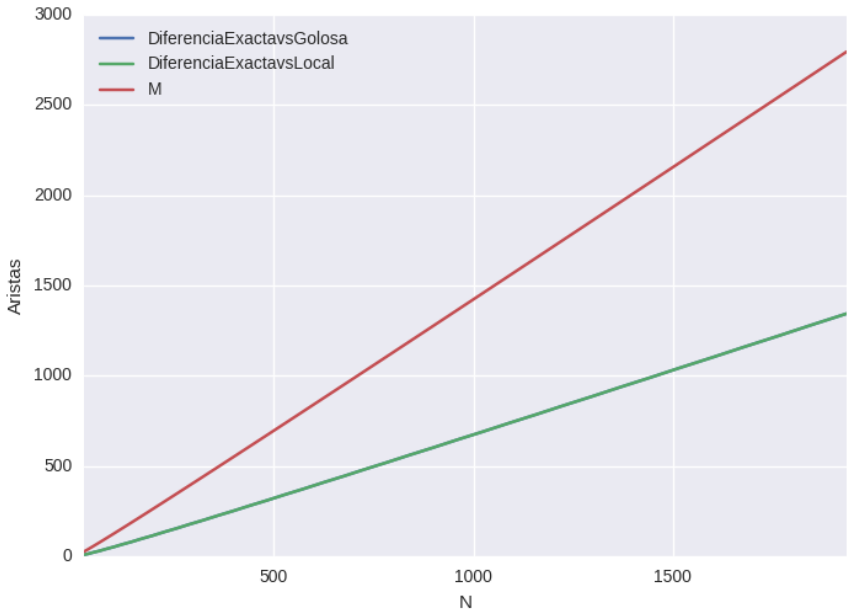

Figure 10: Diferencia del tamaño de la frontera respeto de la solución exacta.

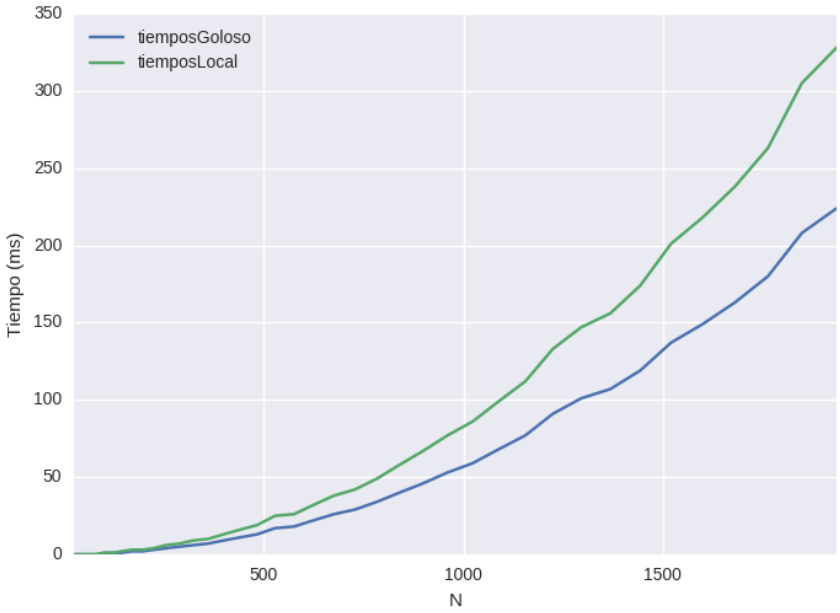

Figure 11: Tiempos de ejecución de algoritmos goloso y búsqueda local.

6 Parte 5: Metaheurística Tabú

6.1 Introducción

Buscando mejorar la solución dada por la búsqueda local, creamos un algoritmo de metaheurística Tabú. El algoritmo de búsqueda tabú consiste en mantener una lista de soluciones (lista tabú) que no deben ser visitadas. Cada vez que el algoritmo no tiene soluciones dentro de la vecindad que está analizando, agrega esa solución a la lista tabú, y así no volver a repetir el análisis sobre esta solución en caso de volver a encontrarla. La idea de la busqueda tabú es poder salir de los mínimos locales para visitar vecindades que no son vistas en la búsqueda local.

6.2 Resolución del problema

El algoritmo Tabú consiste en tomar una solución, resultado de la heurística golosa, mejorarla todo lo posible por medio del algoritmo de búsqueda local, y finalmente tomar una decisión que permita salir de mínimos locales.

Como la diferencia entre la solución devuelta por la búsqueda local y la solución exacta puede ser muy grande, decidimos salir de los máximos locales llamando nuevamente al goloso, de tal forma que devuelva una solución distinta a la devuelta previamente. Luego de hacer varias experimentaciones con los parámetros de la búsqueda tabú, concluimos que el largo de la lista tabú mas eficiente ronda los n/3 elementos. Para el método de parada, iteramos mientras el algoritmo encuentre mejoras, o no mejore por una cierta cantidad. Muchas veces basta con esperar que el algoritmo no mejore por n* 3/4 iteraciones, pero no siempre fue el caso. Por lo tanto creemos que es mas eficiente detener el algoritmo cuando no mejora por n iteraciones.

6.3 Pseudocódigo

Algorithm 7 hTabu

```
function hTabu(AdyMatrix matrizOriginal, int n ) → Solucion
    Solucion solucion
    int sinMejorar ← 0
    Solución mejorS
    do
        solución ← hGolosa2(matrizOriginal, n, tabuList)        ▷ O(hGolosa) = O(n^3)
        nodo2tabuList(solucion.clique, tabuList)                 ▷ O(1)
        solucion ← hLocal(matrizOriginal, solucion, n)
        if mejorS.FronteraSize < solucion.FronteraSize then
            mejorS ← solucion
            sinMejorar ← 0
        else
            sinMejorar++
        end if
    while sinMejorar < n
     return mejorS
end function
```

nodo2tabuList agrega el primer nodo de la solución a la lista tabú, o reemplaza por el nodo correspondiente si la lista esta llena.

La función hGolosa2(AdyMatrix Matriz, int n, vector<int> tabuList) es igual a hGolosa explicado en el punto 3, usando la lista tabú para decidir con que nodos evaluara crear la solución. De esta forma la clique solución nunca tendrá los nodos prohibidos por la lista tabu, pero si los tiene en cuenta para calcular la frontera de la clique.

La lista tabú tiene tamaño n/3, resultado de los experimentos que explicamos en dicha sección, y tiene como finalidad evitar que los nodos guardados en la misma sean utilizados por (tamaño de la lista tabú) iteraciones del algoritmo.

6.4 Complejidad

Vamos a calcular la complejidad del algoritmo. Como el algoritmo es realizar un ciclo y en cada iteración es aplicar lo descripto previamente, la complejidad va a ser del orden $O(cantidadIteraciones * CostoIteracion)$.

- **Cantidad de iteraciones:** El algoritmo itera n veces mientras no encuentra una solución mejor. Como esta la cantidad de repeticiones se reinicia si se encuentra una mejora, tenemos a lo sumo $n * cantidadMejoras$ de iteraciones. Como se vio en la complejidad del algoritmo local, la cantidad de mejoras que se puede realizar es del orden $O(n^2)$. Luego, la cantidad de iteraciones del ciclo va a ser de $O(n * n^2) = O(n^3)$,
- **Costo de la iteración:** El costo de la iteración se basa en tres funciones:
  - **Algoritmo goloso:** Tiene complejidad $O(n^3)$.
  - **Algoritmo local:** Tiene complejidad $O(n^6)$.
  - **Función nodo2tabuList:** Tiene complejidad $O(1)$.

Luego, la complejidad del algoritmo tabú es de $O(n^3 * (n^3 + n^6)) = O(n^9)$.

6.5 Soluciones no óptimas

Sabemos que el algoritmo de búsqueda tabú puede salir del mínimo local que se genera si el grafo de entrada es el descripto en la sección de soluciones no óptimas del goloso, sin embargo dependemos de como se presente el grafo.

Para ver que eso puede ocurrir, analicemos las distintas soluciones que puede devolver la búsqueda local para ese tipo de grafos, usando la notación n_i = nodo de K_n, n_j = vecino de n_i y n_k pero no está en K_n y n_k = vecino de n_j pero no de n_i. Recordemos que gr(n_i) = n, gr(n_j) = n+1 y gr(n_k) = 1.

- $K_s = \{n_j, n_i\}$: Este caso se resuelve porque el algoritmo prohíbe n_j. Al rehacer el goloso y comenzar con el mismo n_i como K_1, no hay ningún nodo de grado n+1 para agregar a n_i, así que creará K_2 con otro nodo de K_n, y luego encontrara la solución óptima al ir agarrando nodos del subgrafo completo, pues ya no puede agregar ningun n_j dado que con ninguno formara una clique.
- $K_s = \{n_j, n_k\}$: Nuevamente se resuelve porque se prohíbe n_j y al ejecutar el algoritmo goloso, cuando empiece por el n_i adyacente al n_j prohibido, encontrara la solución óptima .
- $K_s = \{n_i, n_j\}$ y $K_s = \{n_k, n_j\}$: Dada la estabilidad de nuestro algoritmo, estos casos ocurren cuando n_i o n_k son números menores a n_j (según su orden de entrada), ya que encuentra esta solución antes que otras permutaciones. En esta situación el nodo prohibido (n_i o n_k) no es el ideal, debido a que al volver a ejecutar el goloso, vuelve a encontrar la 2-clique con el nodo n_j. El peor caso es cuando se agrega n_i a la lista tabú, ya que se pierde un nodo de la K_n principal hasta que ocurran $\|lista\ tabu\|$ iteraciones. Si además el orden de los nodos en el grafo es $\forall_{i,j}$ $ord(n_i) < ord(n_j)$ el tabu no mejorará la solución porque siempre entrara en este caso (notar que esto no depende del tamaño de la lista tabú, como se vera en los experimentos del grafico 18)

En los grafos que testeamos como peor caso tenemos ordenados los nodos como explicamos en el tercer caso y por lo tanto el tabú no mejora la solución de la heurística local.

6.6 Experimentación

6.6.1 Parámetros de la Búsqueda Tabú

En esta sección mostraremos los experimentos que nos llevaron a decidir los parámetros de la búsqueda tabú.

En primer lugar tomamos la decisión de realizar iteraciones para salir de los mínimos locales con el goloso, ya que hacerlo con una vecindad nos resulto muy poco confiable. Nuestra idea entonces es prohibirle al goloso que repita una de las k soluciones previas, siendo k uno de los parámetros a calcular (el tamaño de la lista tabú).

La primera forma propuesta para evitar que el goloso repita soluciones, fue agregar todos los nodos de la solución a la lista tabú.

Como este método quita muchos nodos y potencialmente evitaría muchas nuevas soluciones, la segunda propuesta fue que la lista tabú solamente guarde un nodo por solución. El nodo elegido para los primeros tests fue el primero de la solución devuelta por la búsqueda local.

Con estos 2 algoritmos creamos un tester, que para cada valor de n evaluado crea 10000 grafos aleatorios. Para cada grafo aleatorio, prueba los valores de k de 1 hasta n y guarda el primer valor de k que retorna la frontera mas grande. También calculamos la máxima cantidad de iteraciones que fueron necesarias para conseguir una mejora.

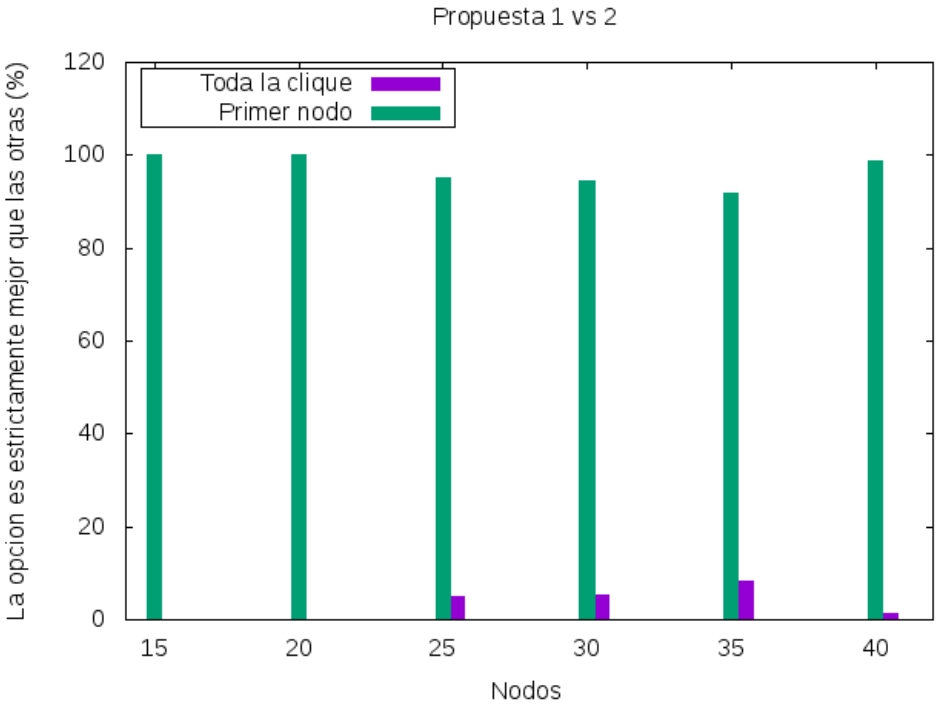

Figure 12: Prohibir grafo completo vs un nodo

Como se puede ver en el gráfico, la cantidad de veces que el segundo algoritmo (prohibir el primer nodo) da una solucion con una frotera estrictamente mayor que guardar todos los nodos de la solución es muy superior a la inversa. Ya en este test k esta siempre acotado por $n/3$ y la cantidad máxima de repeticiones necesarias parece acotado por $n * \frac{3}{4}$. Lo graficamos luego para que sea mas claro y prolijo.

El siguiente paso fue considerar otros nodos a quitar. Para eso comparamos con el mismo test descrito anteriormente 4 opciones: quitar el primer nodo, el de mínimo grado, el de máximo grado y uno aleatorio.

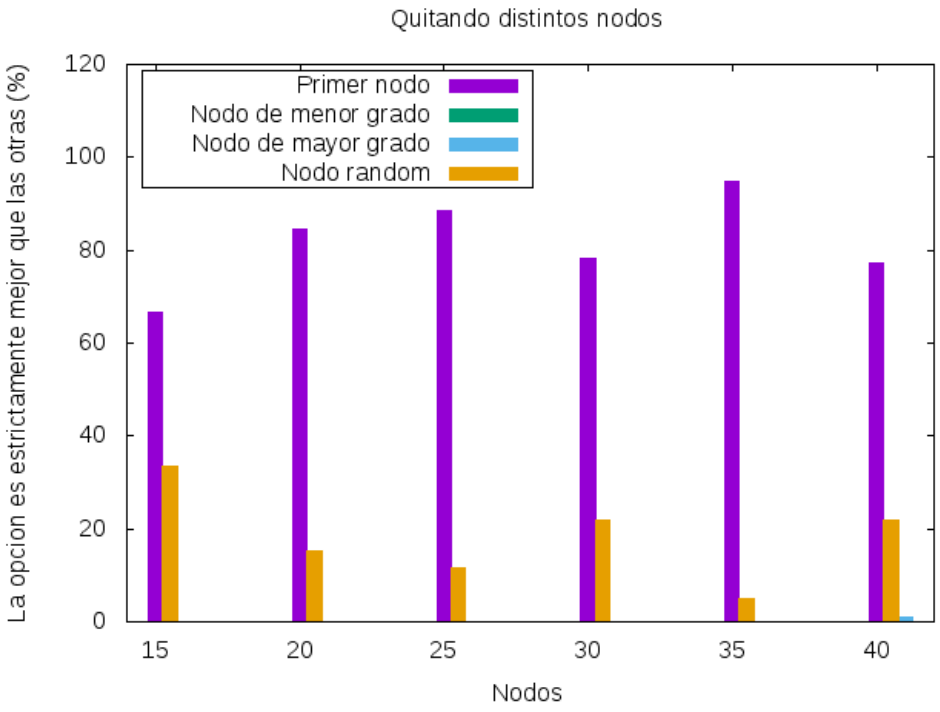

Figure 13: Análisis sobre que nodo es mejor prohibir

Podemos apreciar que quitar el primer nodo es mucho mas eficiente que el resto de las opciones. Es importante aclarar que comparamos la cantidad de veces que la opción dio estrictamente mejor que el resto de opciones.

Finalmente, eligiendo sacar el primer nodo de la clique devuelta por la búsqueda local, comparamos las distintas soluciones variando el tamaño de la lista tabú, que llamaremos k. Para cada n, del menor k necesario para encontrar la mayor frontera, graficamos el que mas se usa en los distintos grafos aleatorios. Luego graficamos el mínimo y máximo k entre los primeros k que dan la solución de mayor frontera para cada grafo. Esto lo hacemos con el mismo test explicado antes.

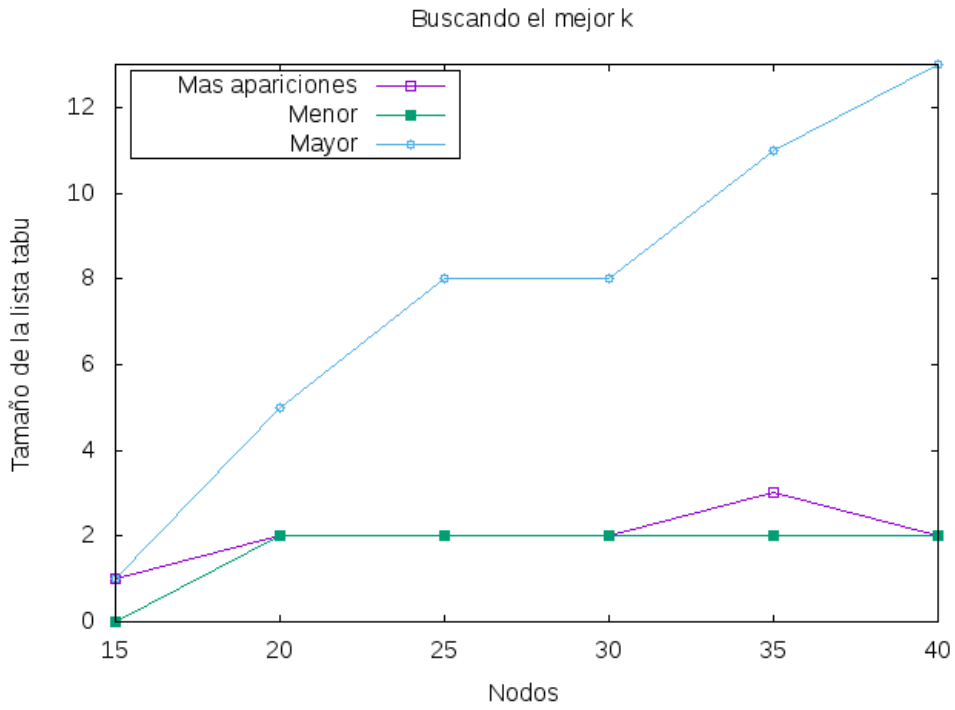

Figure 14: Comparación entre los menores tamaños de lista tabú para devolver la mejor frontera

Éste gráfico muestra el máximo valor de k acotado por $n/3$, y la mayoria de los valores del primer k que da la solución final alrededor de $k = 2$. Queda entonces experimentar para ver si al utilizar un k mas grande que el mínimo necesario para obtener la mejor solución del tabú, hacemos que este mejore o empeore. Para eso analizamos k=n/3, k=2 que es el k que la mayoria de los grafos necesitan para dar la mejor solucion posible y k=3 para tener mas variedad (dado que tambien aparece en el grafico).

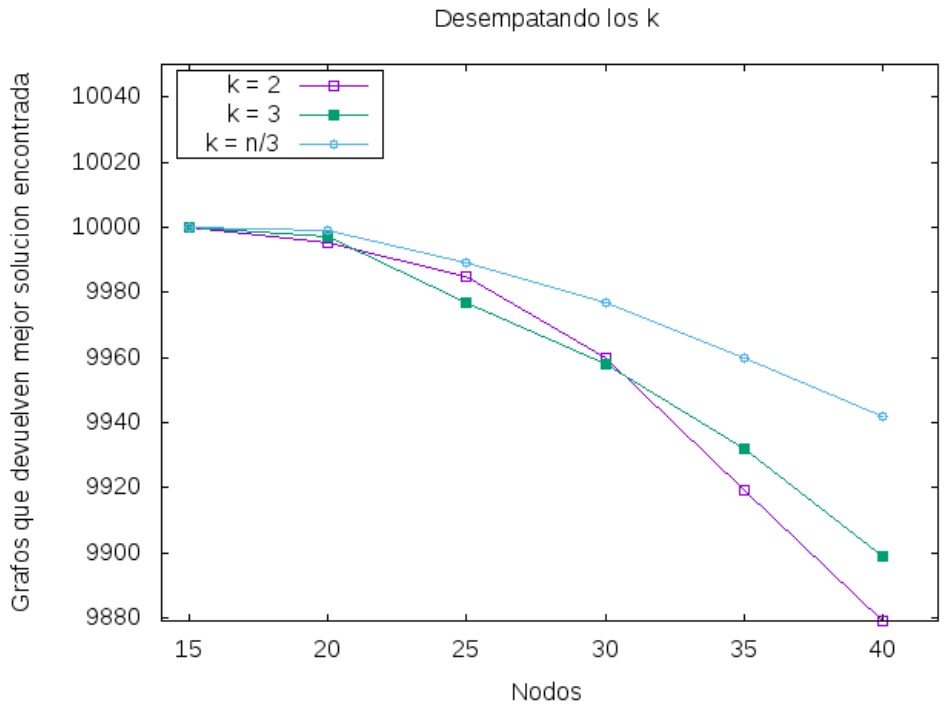

Figure 15: Análisis de los mejores tamaños evaluados para la lista tabú

Con esto definimos $k = n/3$ por ser el valor que mejores resultados nos da. Ahora graficaremos los valores de las repeticiones necesitadas para conseguir estos resultados:

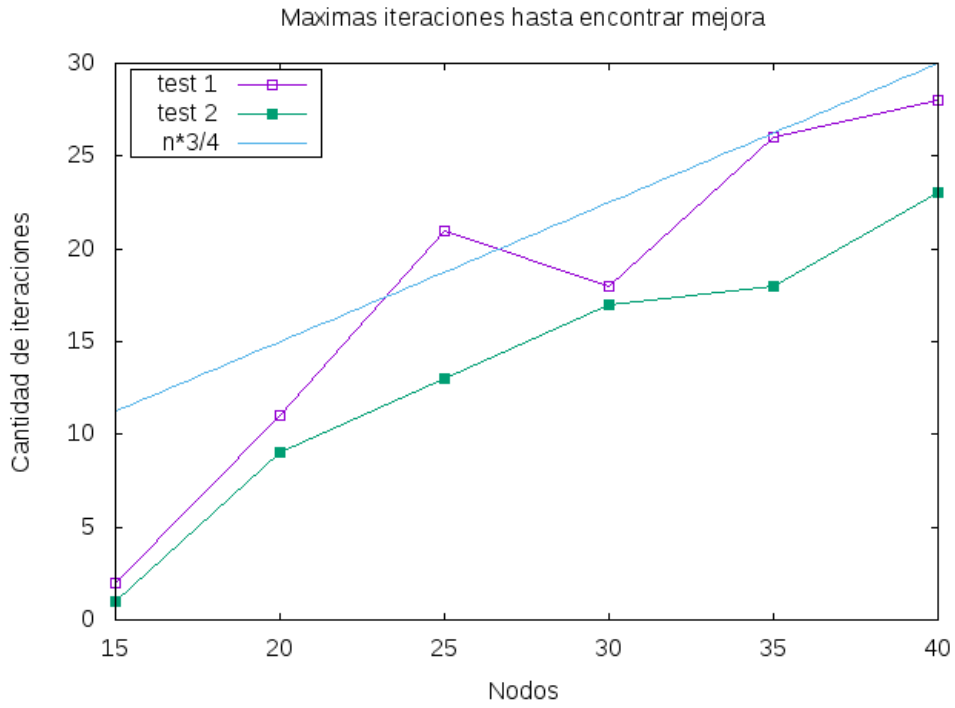

Figure 16: Repeticiones máximas para conseguir una frontera mejor

Finalmente podemos ver que los mejores parámetros para el algoritmo son los descritos previamente en la sección de pseudocódigo. Si bien la mayoría de los valores de repeticiones necesarias para encontrar la mejor solución esta acotada por $n * \frac{3}{4}$, al no ser siempre el caso elegimos dejar la cantidad de iteraciones sin que el algoritmo encuentre una mejor frontera en n.

Resta verificar que lo analizado se aplica al peor caso encontrado para estos algoritmos. Lo primero que analizamos es que las opciones evaluadas no mejoraban a la solución del goloso. Teniendo eso en cuenta, y dado que el comportamiento del algoritmo con ese grafo en particular fue probado previamente, decidimos evaluar sobre grafos isomorfos al mismo. Esto es porque como vimos en la sección de la solución no óptima del algoritmo tabú, éste depende del orden que tengan los nodos.

Utilizando el mismo tester usado en los gráficos anteriores, pero variando la cantidad de grafos a 50 (grafos isomorfos generados aleatoriamente), realizamos el mismo análisis.

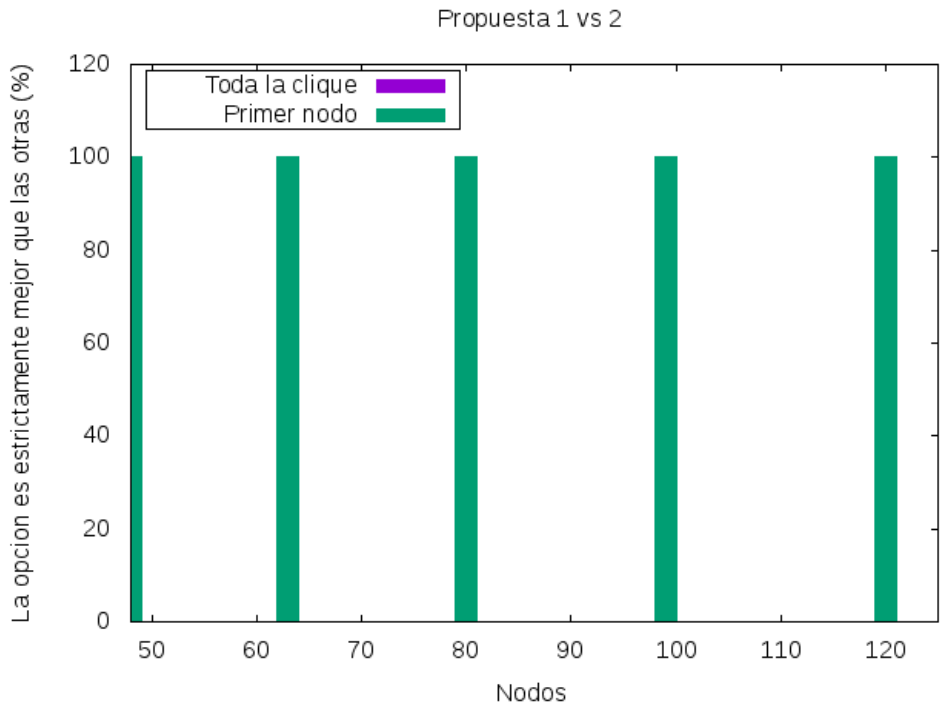

Figure 17: Prohibir grafo completo vs un nodo

En el primer gráfico volvemos a ver que la opción de guardar todos los nodos de una solución es ineficiente, y guardar el primer nodo de la misma da mucho mejor resultado.

En cuanto al segundo gráfico, no lo mostramos pues ningún método dio estrictamente mejor a otro. Solamente cabe notar que elegir el nodo de menor grado nunca dio la frontera mas grande obtenida para un grafo. Esto se corresponde con la estructura del grafo, siendo que los nodos problemáticos son aquellos que tienen mayor grado.

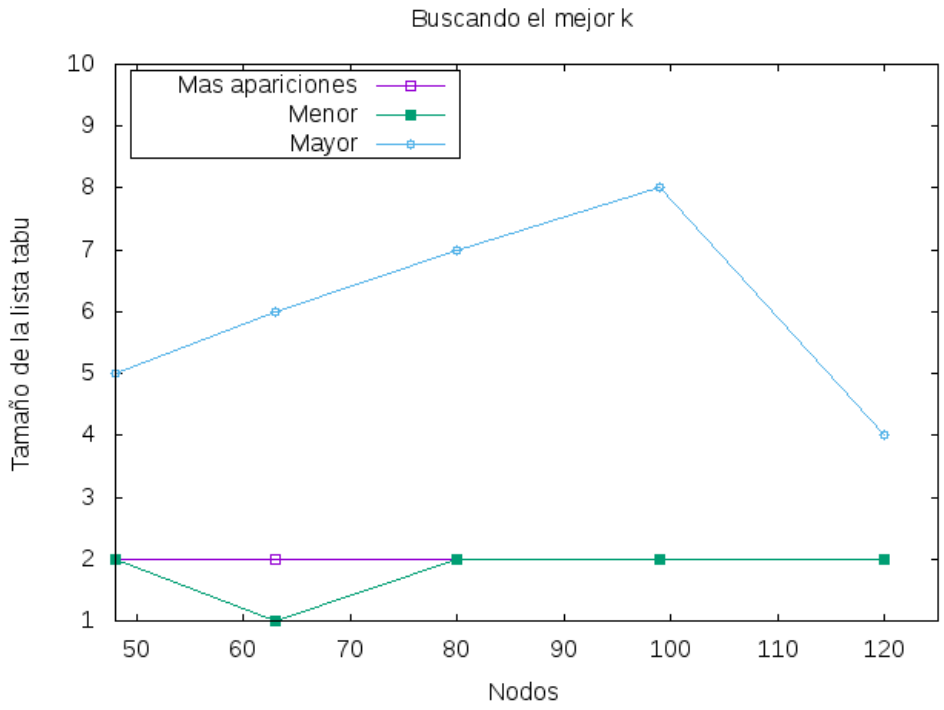

Figure 18: Comparación entre los menores tamaños de lista tabú para devolver la mejor frontera

Aquí el mínimo k y el k con la mayor cantidad de apariciones es casi el mismo al analizado previamente. La única diferencia es que el mayor k desde el que no se encuentran mejores soluciones es proporcionalmente mucho mas chico que $n/3$

Finalmente verificamos que tener un k superior al necesario da mejores resultados

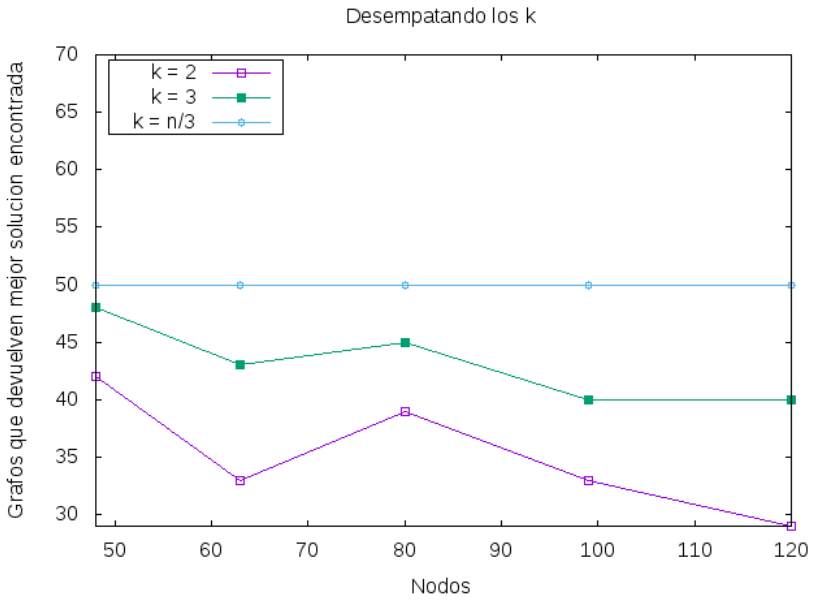

Figure 19: Análisis de los mejores tamaños evaluados para la lista tabú

No incluimos ningún gráfico sobre la máxima cantidad de iteraciones pues en todos los casos las iteraciones necesarias fueron menos de n/2, así que no aportan informacion nueva al análisis.

6.6.2 Experimento 1: casos aleatorios

Realizamos un análisis de tiempo y calidad de las soluciones comparando todos los algoritmos para grafos aleatorios de tamaño hasta 22, debido a la alta complejidad el algoritmo exacto, y hasta 140 nodos para analizar los algoritmos sin el exacto. Cada resultado para los tamaños de entrada son promedios 500 grafos aleatorios distintos, cada uno evaluado con todos los algoritmos.

Los tiempos obtenidos muestran la misma conclusión que los ejercicios anteriores: el algoritmo exacto tiene una performance un lenta para aplicarlo en la práctica. Del segundo gráfico se puede extraer que la búsqueda tabú es ampliamente más lenta que la búsqueda local y la golosa, el cual es el costo para tratar de mejorar las soluciones. Sin embargo, como se explicó previamente, las soluciones obtenidas fueron las mismas que el algoritmo goloso, no mejorando en absoluto al usar búsqueda tabú.

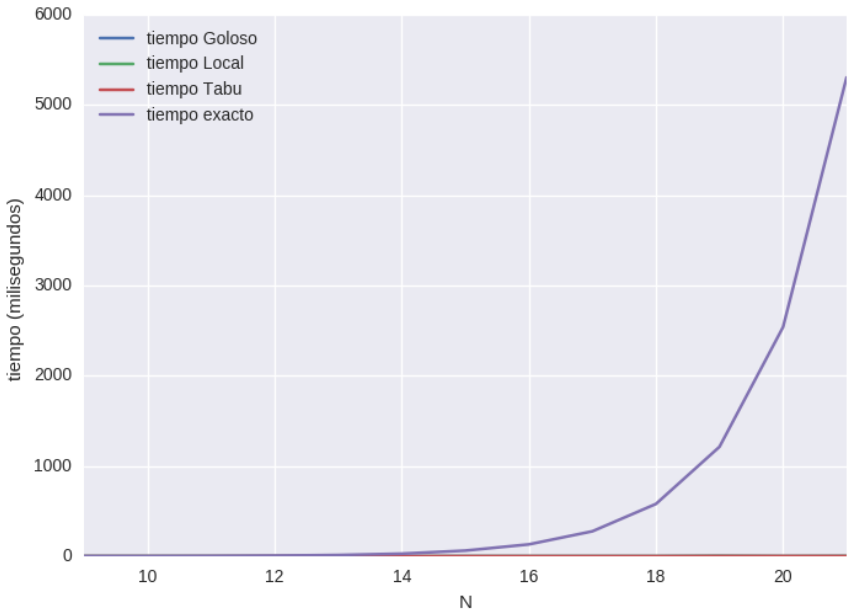

Figure 20: Tiempos de los algoritmos.

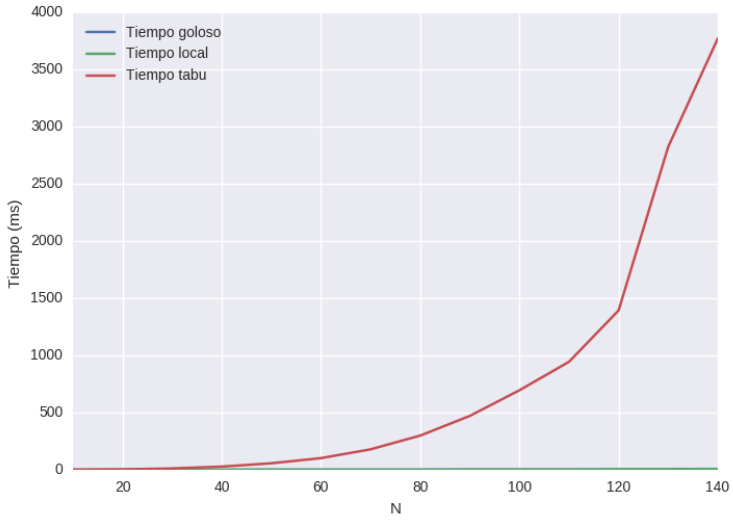

Figure 21: Tiempos de las heuristicas

Los resultados de la búsqueda tabú para instancias de tamaño menor a 22 las comparamos con los resultados exactos y devuelven soluciones exactas o con una diferencia de 1 eje. Luego para valores hasta 140, los comparamos con las soluciones dadas por el algoritmo goloso y el local. Se puede observar como en promedio la búsqueda tabú mejora ampliamente la solución obtenida y las notables diferencias entre las soluciones obtenidas entre los algoritmos..

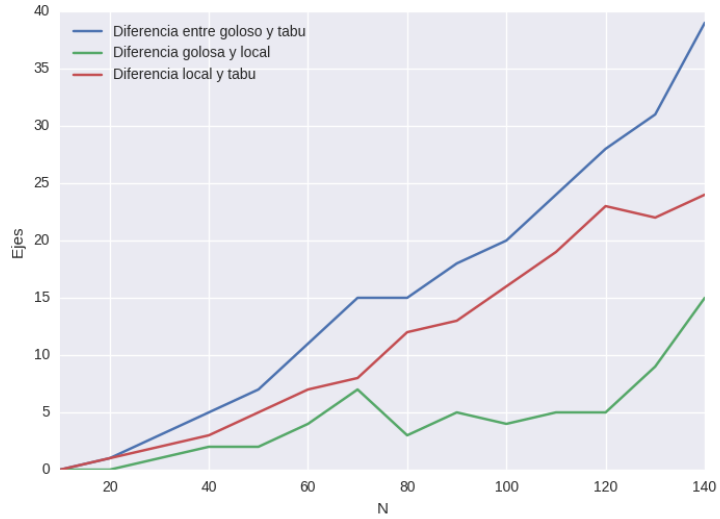

Figure 22: Diferencia entre soluciones obtenidas.

6.6.3 Experimento 2: Caso patológico

Para este experimento utilizamos la familia de grafos patológicos de los ejercicios anteriores. Cada instancia fue analizada promediando su performance repitiendo 100 veces cada una.

El gráfico muestra la amplia diferencia de tiempos que tiene la búsqueda tabú respecto de la búsqueda local al intentar encontrar mejores soluciones. Sin embargo en estas familias el objetivo no se logra. Las soluciones obtenidas por la búsqueda tabú fueron los mismos que la heurística golosa. En esta familia de grafos, utilizar este algoritmo implica obtener los mismos resultados que con el goloso, muy alejados de la solución exacta, y a un costo de tiempo elevadamente superior.

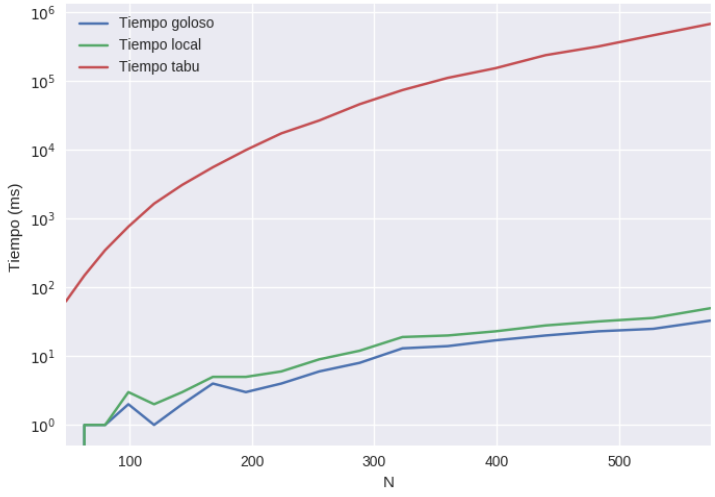

Figure 23: Tiempos de los casos patológicos en escala logarítmica.

Como se ve reflejado en el gráfico siguiente, el algoritmo logra mejorar notablemente la solución obtenida. Sin embargo, incluso mejorando la solución, no logra reducir a la mitad de la diferencia del tamaño de las fronteras de las soluciones.

7 Problema V: experimentación sobre un nuevo conjunto de instancias

7.1 Comparación de costo temporal y calidad entre Goloso y Local vs Tabú para grafos aleatorios

Vamos a comparar el tiempo y calidad de dichos algoritmos

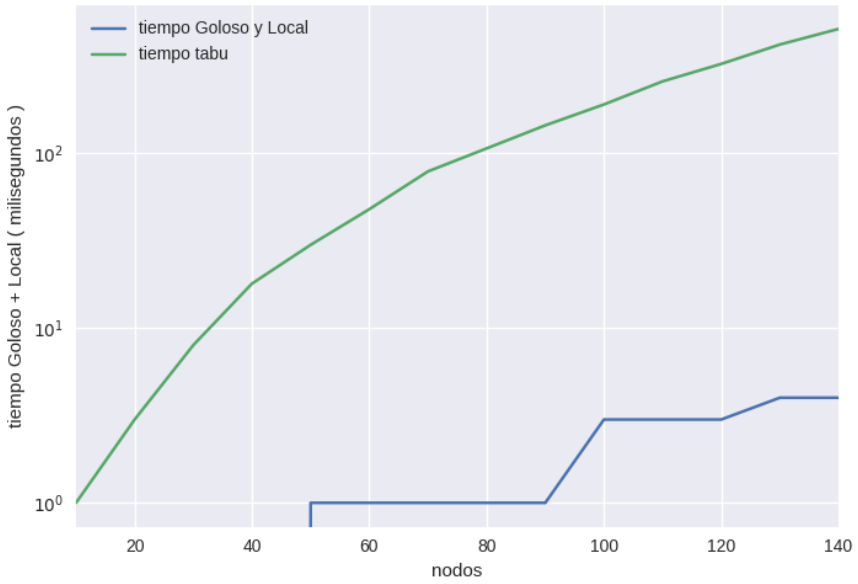

Figure 24: Curva comparativa de velocidad en escala logarítmica

Como se puede apreciar, es claro que el algoritmo de Tabú Search va a ser mucho mas costoso que los dos anteriores ya que este aplica a los dos anteriores sin embargo la cantidad de tiempo tampoco es demasiado grande por lo que habría que analizar en cuanto mejoran las soluciones y si es conveniente gastar mas tiempo o no en realizar esta búsqueda.

Para ver la calidad de las soluciones tenemos este otro experimento

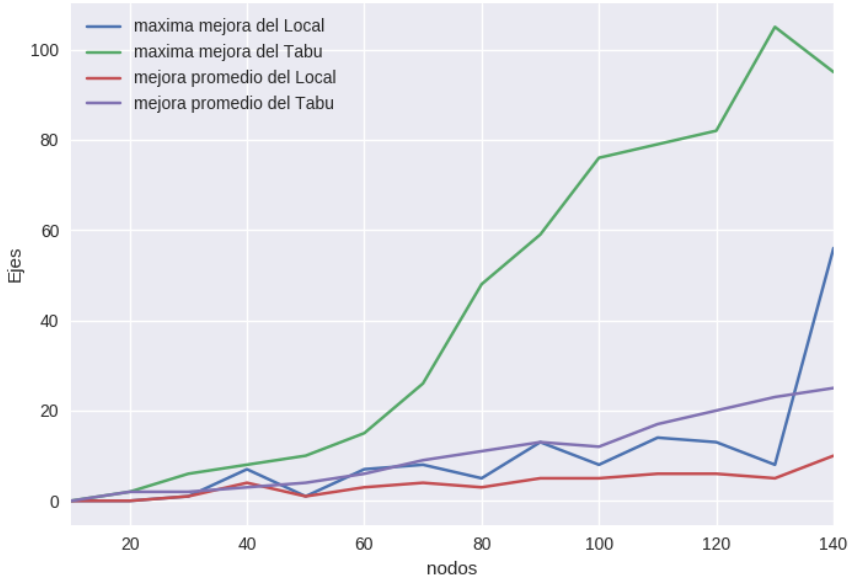

Figure 25: Calidad de las soluciones

Hemos comparado la calidad de las soluciones y lo que podemos es que, el tabú efectivamente ser muy bueno ya que, en promedio la diferencia que nos da el tabú es el doble que el Goloso y Local, osea en promedio el tabú nos está mejorando nuestra solución un 100% aproximadamente y si miramos el máximo a los que ha alcanzado también son mucho mayores los de la meta-heurística Tabú.

Por lo tanto podemos concluir que vale la pena el invertir tiempo extra si vamos a conseguir mejores soluciones.

7.1.1 Grafos clásicos:

Nuestro algoritmo goloso resuelve de manera exacta los grafos más clásicos. Como la solución ya es la máxima, los algoritmos de búsqueda local y tabú no van a poder mejorarla. En esta sección describiremos cuáles son los grafos que resuelve así, y brevemente por qué los resuelve de esta manera.

- **Grafo completo de tamaño n:** Como cualquier subgrafo de este tipo de grafos se compone de nodos que poseen aristas a todos los nodos que no son parte del subgrafo, la cantidad de ejes de la frontera que va a tener es la cantidad de nodos del subgrafo (n_s) multiplicada la cantidad de nodos que no están en el subgrafo ($n - n_s$). Es decir, la frontera va a ser de $n_s * (n - n_s)$. Siendo n un valor fijo, el máximo de esta función se encuentra en $n_1 = n/2$. Si n es par, $n/2$ es el tamaño de la CMF. Si n es impar, entonces los candidatos máximos enteros van a ser $\lceil n/2 \rceil$ y $\lfloor n/2 \rfloor$. Como $n - \lfloor n/2 \rfloor = \lceil n/2 \rceil$, cualquiera de los dos candidatos va a dar una CMF, siendo la cardinalidad de dicha frontera $\lceil n/2 \rceil * \lfloor n/2 \rfloor$. Esta función nos dice también que partiendo desde una clique de 0 nodos, ir agregando nodos hasta el valor máximo la frontera va creciendo (hasta el máximo es creciente). De forma análoga, tener una clique de n nodos e ir sacándole hasta llegar al máximo incrementa la frontera. Como nuestro algoritmo parte desde una clique mínima y va agregando nodos sea cual sea, mientras crezca la frontera, va a llegar hasta el máximo y encuentra así la CMF.

- **Grafo bipartito:** En este caso sólo se pueden formar cliques de hasta tamaño 2 y los nodos pueden ser de hasta grado 3. De esta forma, la frontera de cardinalidad máxima va a ser la suma máxima entre los grados de dos nodos que forman una 2-clique, restado dos (el eje que los conecta). Como nuestro algoritmo itera sobre todos los nodos y para cada uno va al de mayor grado, encuentra la mejor 2-clique que contiene a determinado nodo para todo nodo. Y luego quedándose con la mejor de entre ellas, obtiene la mejor 2-clique de todo el árbol.

- **Camino, árbol, ciclo y grafo estrella:** Como éstos son grafos bipartitos y nuestro algoritmo los resuelve correctamente, también resuelve correctamente a este tipo de grafos.

7.2 Grafos k-regulares

Finalmente presentamos una nueva familia de grafos donde el goloso no da la solución óptima. En este caso la diferencia de calidad entre el goloso y el exacto no supera para ningún n testeado las 6 aristas. Y a diferencia de la familia de grafos presentada con el algoritmo goloso, la búsqueda tabú devuelve la solución exacta.

El gráfico esta armado con errorbars, donde el punto indica el promedio de las mediciones y el extremo de la barra es el valor máximo alcanzado en alguna iteración.

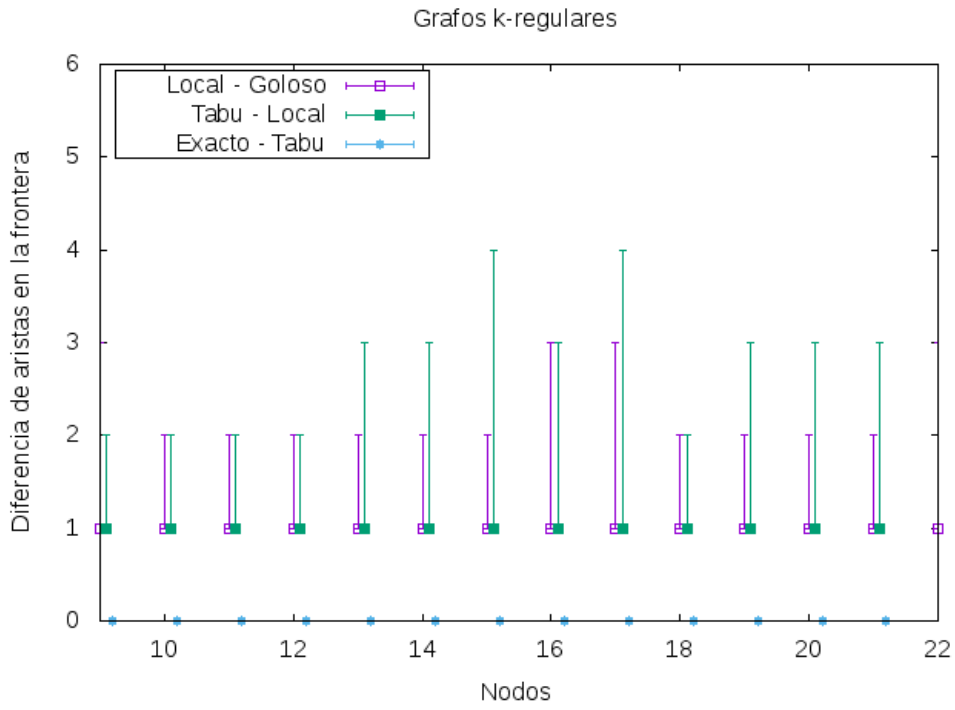

Figure 26: Calidad de las soluciones para grafos k-regulares con errorbars (promedio - máximo)

Como se puede observar en el gráfico, en todos los casos la heurística tabú encontró la solución exacta.

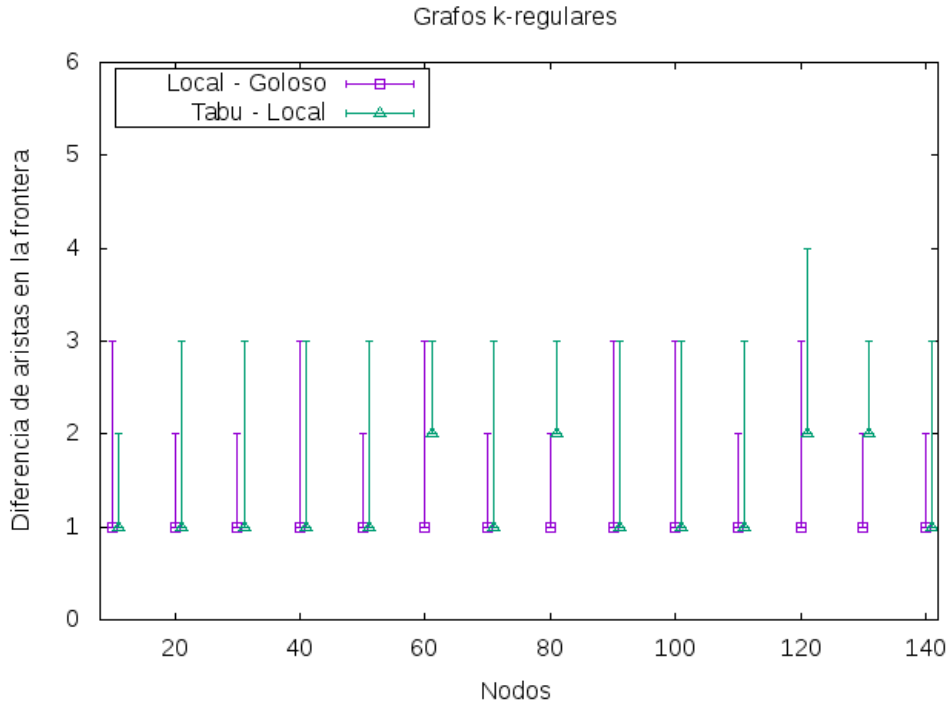

Figure 27: Calidad de las soluciones para grafos k-regulares comparando las mejoras de las heurísticas

En cuanto a los tiempos, nuevamente el algoritmo exacto tarda un tiempo mucho mayor al algoritmo tabú, demorando apenas 4ms para un grafo de 22 nodos.

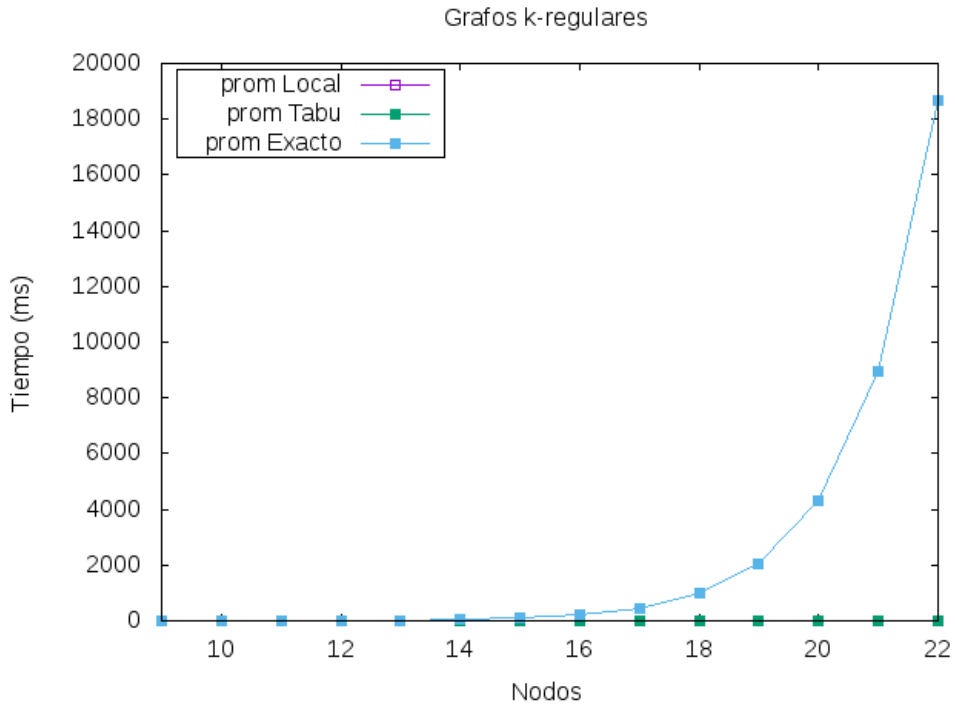

Figure 28: Tiempos para resolver grafos k-regulares

La diferencias de tiempos entre tabú y local también son muy notables, aunque no se acercan a los tiempos del exacto.

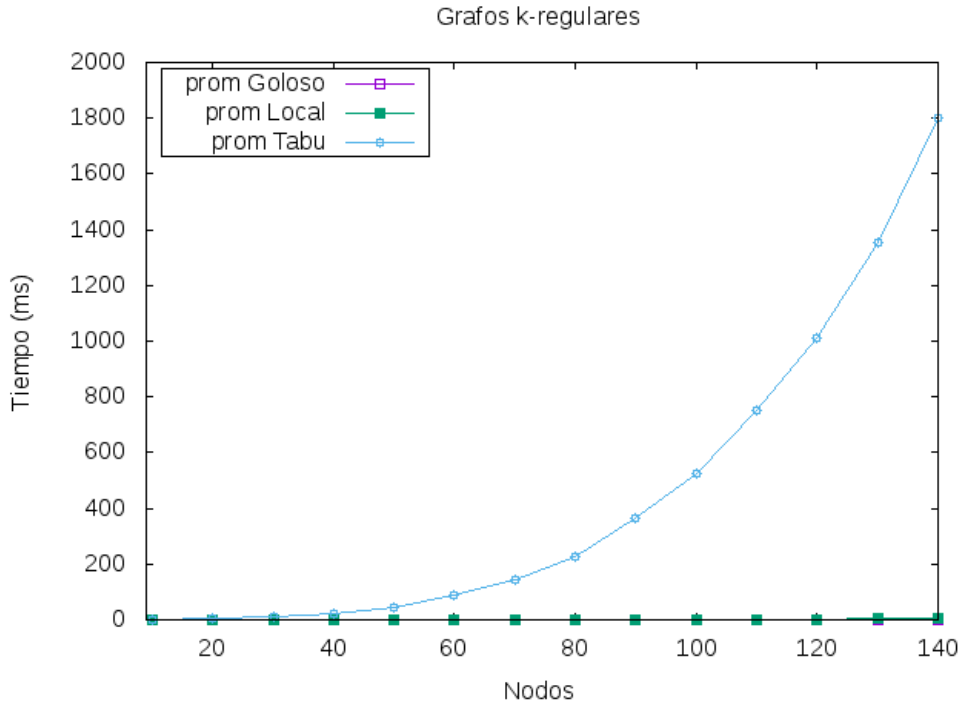

Figure 29: Tiempos para resolver grafos k-regulares