Algoritmos y Estructuras de Datos III

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Junio 2017

Trabajo Práctico 3

Alumno	LU	Correo electrónico
Seijo, Jonathan Adrián	592/15	jon.seijo@gmail.com
Reyes Mesarra, Darío René	838/15	eran6reyes@gmail.com
Cordoba, Lucas	094/15	lmcordobaa@gmail.com
Balbachan, Alexis	994/12	alexisbalbachan@gmail.com

Índice

1. Motivación

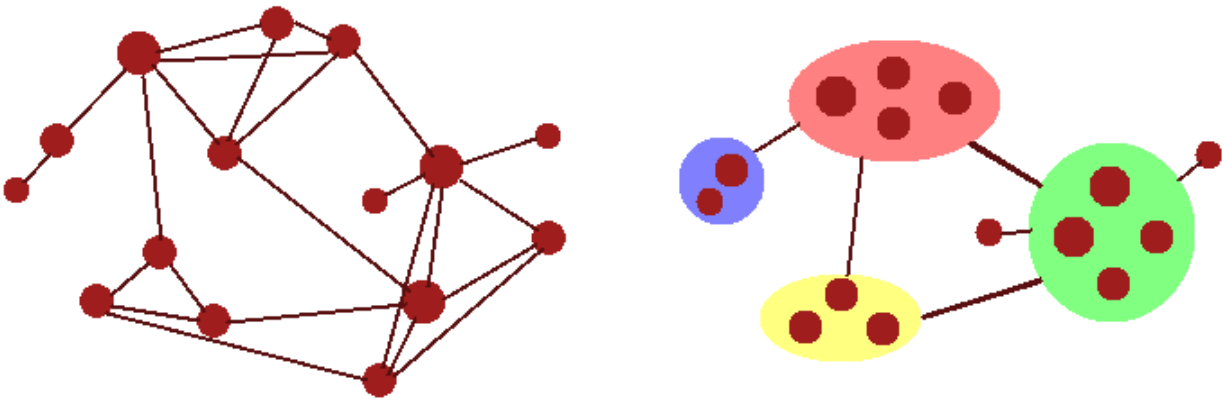
Un *clique* en un grafo G es un subconjunto de vértices que induce en G un subgrafo completo. La *frontera* de un subconjunto de vértices S es el conjunto de vértices con un extremo en S y otro en $G \setminus S$. El presente informe analiza el problema de encontrar una clique que maximice la cardinalidad de su frontera en un grafo determinado. Llamaremos al problema CMF: Clique de Máxima Frontera. Dicho problema pertenece a la clase NP, por lo que es un problema difícil de resolver, y será de interés qué calidad de respuestas podemos conseguir en un tiempo razonable.

Antes de entrar de lleno en analizar posibles resoluciones, veamos por qué calcular la clique con máxima frontera de un grafo es un problema interesante con posibles aplicaciones reales. Primero hablaremos un poco de qué puede llegar a modelar un 'clique', y qué información de éste puede darnos su frontera.

La palabra 'clique' proviene de la sociologia, en donde representa un grupo unido de individuos con intereses en común. Modelando redes sociales con grafos, la definición formal de clique que acabamos de realizar hasta cierto punto coincide con esta noción. Esto ocurre porque el hecho de que un grupo de personas en una red induzca un subgrafo completo, implica que todas las personas están conectadas entre sí. Esto nos da la pauta de que podríamos calcular 'cliques' para conocer mejor las dinámicas dentro de una red social. También pueden utilizarse en el contexto de la química, por ejemplo, para detectar similaridades estructurales entre proteínas distintas [1]. Pensando en el complemento de un grafo, también pueden ayudarnos a encontrar conjuntos de vértices independientes (pues si en una clique todos los nodos están conectados entre sí, en el complemento del grafo no lo estarán).

Orientados al problema CMF, el caso de las redes sociales es de particular interés. Mencionamos que un 'clique' puede representar un grupo cohesivo de personas. Una observación razonable es que adyacencia total dos a dos puede ser una condición demasiado restrictiva para caracterizar a un grupo en un modelo estándar de relaciones personales. Sin embargo, hay formas de sortear estos problemas, como utilizar la definición de *n-cliques* (Luce 1950, Alba 1973) que, grosso modo, es una estrategia para relajar las adyacencias dentro de un grafo. El objetivo es admitir grupos como n-cliques que no calificarían estrictamente como cliques. De cualquier forma, lo que queremos transmitir es que generalmente es posible asumir que la definición de clique se corresponde bien con los grupos en redes sociales modeladas como grafos.

Lidiando con redes sociales con mucha información, podemos utilizar los cliques para presentar la información de forma más compacta y más representativa, lidiando con grupos en vez de individuos.



A la izquierda un grafo que modela una pequeña hipotética red social. A la derecha representamos la misma red social agrupando los individuos mediante cliques disjuntos maximales. En esta última imagen, los ejes entre cliques representan la existencia de elementos en ambos conjuntos que están conectados entre sí.

Vemos como esto puede permitirnos identificar mejor las dinámicas grupales. Además, parece apropiado analizar redes sociales en torno a los grupos dentro de la misma, ya que son una unidad más representativa en términos del impacto que tienen en la red.

Al pensar en cliques representativos de una red, una idea sencilla es buscar los cliques más grandes, ma-

ximizando la cantidad de vértices. No es la única forma, y aquí es justamente donde entra la definición de frontera. ¿Qué representa una frontera para una clique? Por definición, son todas las conexiones que 'cruzan' la clique, esto es, que conectan algún miembro de la clique con alguién de afuera. Si medimos qué tan conocido es un individuo por la cantidad de amigos que tiene, entonces tiene sentido medir qué tan conocido es una clique por la cantidad de conexiones que tiene al exterior, en otras palabras, la cardinalidad de su frontera. Por lo tanto, en este caso resolver CMF representaría encontrar el grupo con más influencia dentro de una red social.

2. Algoritmo Exacto

2.1. Explicación

Nuestro problema a resolver es CFM: Clique de Frontera Máxima. Como aclaración, una clique es un subgrafo completo pero no necesariamente es maximal (aunque podría serlo). Debemos encontrar una clique en el grafo que tenga frontera máxima. Como una clique está formada por nodos del grafo, una forma de resolver CFM es revisando todos los posibles subconjuntos de nodos. En caso de que formen clique, calculamos su frontera y nos quedamos con el máximo de ellas.

Para saber si un conjunto vs de vértices es clique, revisamos si todos los vértices en vs son vecinos entre sí. Esto significa que forman un subgrafo completo (pues los vértices y aristas forman parte del grafo principal), y por lo tanto una clique.

Para calcular la frontera, dada una clique c recorremos cada vértice $v \in c$ y contamos cuántos son los vecinos de v que **no** están en c.

Si recorremos todos los subconjuntos posibles de nodos estamos recorriendo todas las posibles cliques. Entonces podemos quedarnos con aquella que tenga máxima frontera. El algoritmo siempre encuentra la solución óptima, pues encuentra todas las soluciones, implicando que efectivamente es un algoritmo exacto.

La implementación de la generación de todos los posibles subconjuntos está hecha recursivamente, utilizando el árbol binario que está implícito en la recursión. Considerando que cada nodo es un elemento, en cada etapa puede estar o no estar incluido, por lo que consideramos ambos estados posibles. Al final se terminan considerando todas las combinaciones posibles.

Una vez obtenido algún subconjunto (cuando ya no hay mas elementos que considerar), revisamos si lo acumulado forma clique, y en caso que sí, calculamos su frontera. Nos quedamos con la máxima de todas ellas.

2.2. Pseudocódigo

Referencias de variables globales para el pseudocódigo:

- n: La cantidad de nodos
- *solucion*: Secuencia que contiene la clique solución
- $fronteraMax$: El cardinal de la frontera de la clique solución

```
function RESOLVER
    LeerInput()                                              ▷ O(n^2)
    solucion ← ∅                                             ▷ O(1)
    fronteraMax ← 0                                          ▷ O(1)
    GenerarSubconjuntos(∅, 0)                                ▷ O(2^n * n^3)
```

```
function GENERARSUBCONJUNTOS(conjNodos, actual)   ▷ O(2^n * n^3), ver detalle en sección Complejidad.
    if actual = n then                                       ▷ O(1)
        if EsClique(conjNodos) then                          ▷ O(n^3)
            fronteraActual ← Frontera(conjNodos)             ▷ O(n^2)
            if fronteraActual > fronteraMax then             ▷ O(1)
                fronteraMax ← fronteraActual                 ▷ O(1)
                solucion ← conjNodos                         ▷ O(1)
    else                                                     ▷ Ver sección Complejidad.
        GenerarSubconjuntos(conjNodos, actual + 1)
        GenerarSubconjuntos(conjNodos ∪ {actual}, actual + 1)
```

```
function EsClique(conjNodos)
    for v ∈ conjNodos do                                          ▷ O(n^3)
        for w ∈ conjNodos do                                      ▷ O(n^2)
            if (v ≠ w) ∧ (¬SonVecinos(v, w)) then                 ▷ O(n)
                return False                                      ▷ O(1)
    return True
```

```
function SonVecinos(v1, v2)
    for w ∈ vecinos[v1] do                                        ▷ O(n)
        if w = v2 then                                            ▷ O(1)
            return True                                           ▷ O(1)
    return False                                                  ▷ O(1)
```

```
function Frontera(clique)
    enClique ← vector(n, False)                                   ▷ Vector con n Falses, O(n)
    for v ∈ clique do                                             ▷ O(n)
        enClique[v] ← True                                        ▷ O(1)
    contador ← 0                                                  ▷ O(1)
    for v ∈ clique do                                             ▷ O(n^2)
        for vecino ∈ vecinos[v] do                                ▷ O(n)
            if ¬enClique[vecino] then                             ▷ O(1)
                contador + +                                      ▷ O(1)
    return contador                                               ▷ O(1)
```

2.3. Complejidad

- LeerInput es $O(n^2)$: Leo como máximo todas las aristas posibles.
- SonVecinos es $O(n)$: Recorro la lista de adyacencia de un vértice, como mucho tiene $O(n)$ vecinos.
- EsClique es $O(n^3)$: Para cada vértice se recorren todos los demás y se revisa si son vecinos.
- Frontera es $O(n^2)$: Para cada vértice de la clique se revisan todos sus vecinos.
- GenerarSubconjuntos es $O(2^n * n^3)$: Hay $O(2^n)$ llamados recursivos (pues cada vértice está o no está), y en cada llamado recursivo se revisa si EsClique en $O(n^3)$ y luego su Frontera en $O(n^2)$ $(= O(n^3))$. También puede pensarse que se generan 2^n subconjuntos, y por cada subconjunto se hacen $O(n^3)$ operaciones.

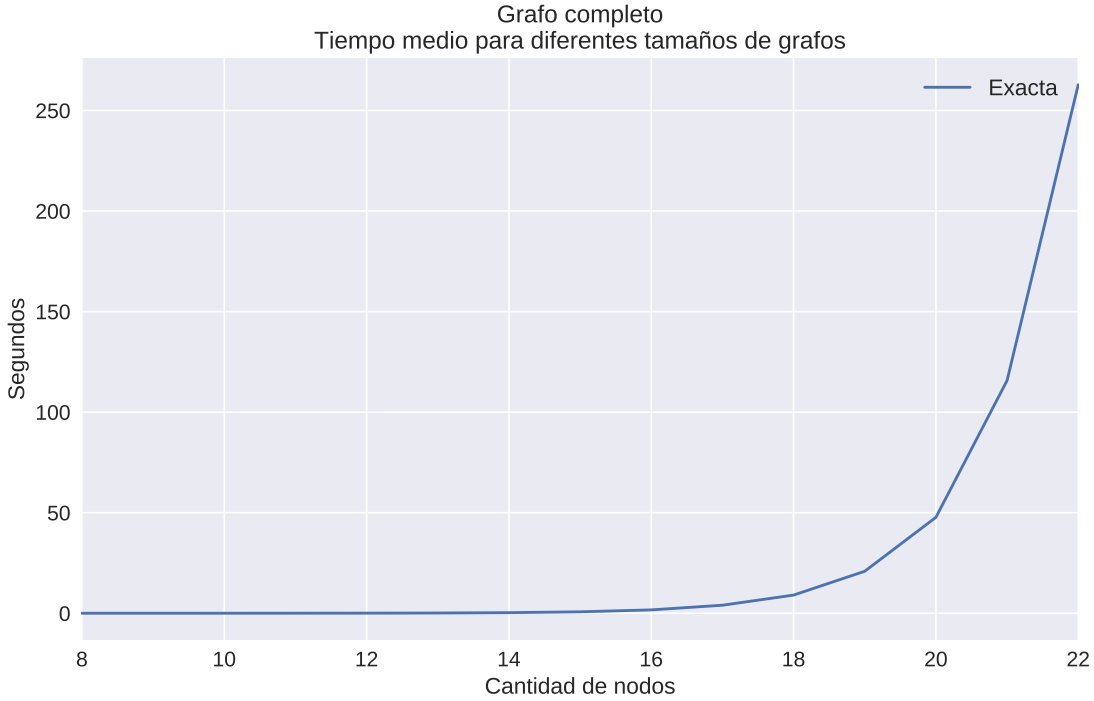
Para Resolver() se lee el input y luego se llama a GenerarSubconjuntos. La complejidad final es:

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

2.4. Experimentación

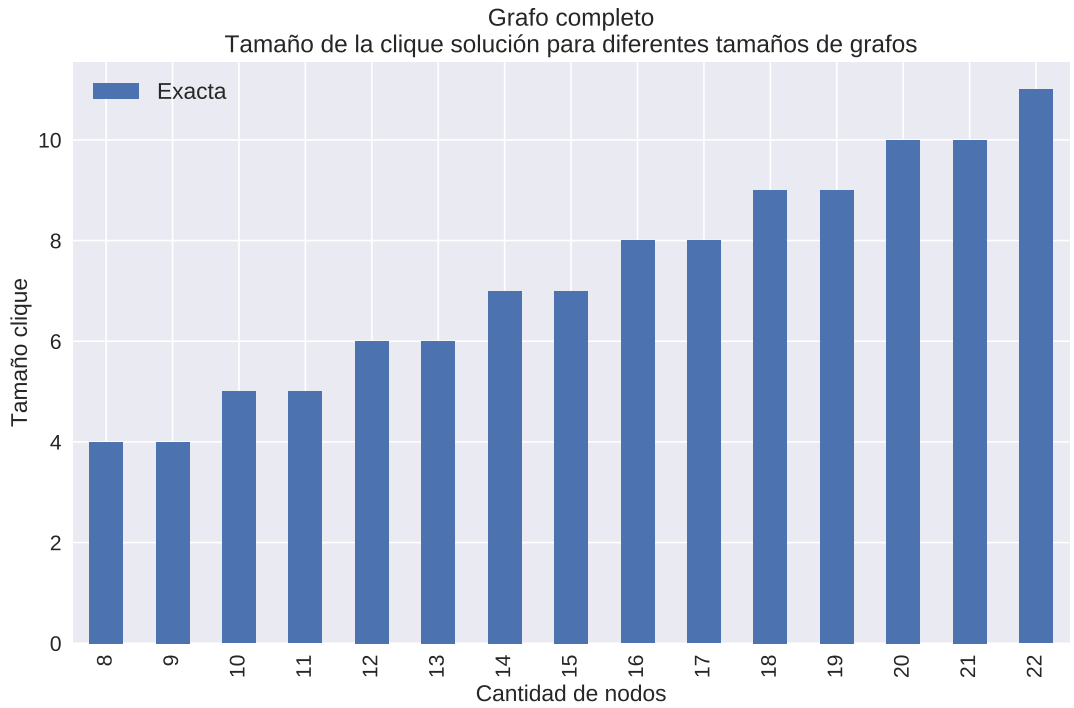
Por lo que vimos en la sección de Complejidad, obtenemos la solución exacta pero a un costo temporal **muy** alto. Veamos empíricamente cuánto se tarda en resolver para cada tamaño de n. Consideramos para esta muestra grafos completos ya que tienen la máxima cantidad de vecinos por nodos, y son por lo tanto el peor caso.

Se puede observar en el siguiente gráfico cómo los tiempos aumentan exponencialmente con respecto al tamaño del grafo, y también puede observarse que para $n = 22$, se tardan aproximadamente 4 minutos. Es claro que es completamente inutilizable para grafos de mayor tamaño.



Aprovechemos esta sección para tratar de entender mejor el por qué de nuestras futuras estrategias. Una primera idea para encontrar la clique con máxima frontera es intentar encontrar la clique máxima. Esta es una *pésima* idea, porque si consideramos un grafo completo de n nodos, la clique máxima es K_n con una frontera de tamaño 0.

Por ejemplo, vemos en el caso particular de los grafos completos que la clique que maximiza la frontera tiene tamaño $n/2$.



Como curiosidad, si nuestro grafo es completo no es necesario correr el algoritmo para conocer cuál será la frontera máxima. Basta considerar que la clique que la maximiza tiene $\frac{n}{2}$ nodos, y por cada nodo de la clique hay $\frac{n}{2}$ vecinos exteriores. Por lo tanto, la frontera máxima de un grafo completo es $\frac{n^2}{4}$.

Si la clique es demasiado grande, es posible que disminuyamos la cantidad de vecinos que están fuera de la clique. Por lo tanto, la estrategia en las seguientes secciones no será buscar una clique máxima, sino ir construyéndolas pero solo mientras la frontera va aumentando.

Como vimos, esta forma de resolver el problema resulta impracticable. En las siguientes secciones analizaremos estrategias para poder resolverlo de formas mucho mas rápidas, pero a coste de no tener certeza de conseguir soluciones óptimas.

3. Greedy

3.1. Explicación

Vimos que construir la mejor solución de forma exacta es algo que, en la práctica, es completamente inútil por su complejidad temporal. Consideremos entonces la siguiente heurística: teniendo un conjunto solución S que forma una clique, en cada paso podemos golosamente agregar a S el mejor nodo que haga aumentar el tamaño de la frontera.

Intuitivamente es un algoritmo rápido, pues en cada iteración principal tomamos un nodo que forme clique con lo que ya tenemos, y ya no quitamos ese nodo. Además, parece razonable que el algoritmo encuentre soluciones buenas: en cada iteracion estamos maximizando la frontera con el mejor nodo posible. Luego de presentar el algoritmo veremos qué tan buena era nuestra intuición.

Mas detalladamente, el algoritmo funciona de la siguiente manera:

Primero consideramos que todos los nodos son candidatos, y vamos a ejecutar el algoritmo siempre que exista algún candidato en la lista. Además mantenemos un vector S que representa la solución actual, inicialmente vacío. Consideramos que nuestra frontera máxima FM inicia con valor -1.

Para cada iteración del ciclo principal, queremos agregar el mejor candidato posible a la solución. Esto significa iterar por todos los candidatos y agregar a la solución aquel que maximice la frontera.

Recorremos la lista de candidatos y tomamos c alguno. Consideramos c como parte de la solución. Si la solución S no forma una clique, entonces quitamos c de S , consideramos que ya no es más un candidato y volvemos al comienzo del algoritmo. Si S forma una clique, entonces calculamos su frontera f.

Comparamos la frontera f con la frontera máxima FM. Si $f < FM$, entonces quitamos a c de S, pues no hace que la solución mejore. Si $f \geq FM$, entonces mantenemos a c, ahora c es nuestro mejor candidato, pues la frontera máxima no empeoró. Actualizamos FM con el valor de f. En ambos casos quitamos también a c de la lista de candidatos, para no repetirlo dos veces.

Una vez encontrado el c que maximice la solución, lo agregamos definitivamente y seguimos iterando hasta que ya no se pueda considerar más candidatos.

Finalizado el algoritmo, S es una clique elegida de manera golosa, con la mayor frontera que se puedo encontrar.

3.2. Pseudocódigo

Referencias de variables globales para el pseudocódigo:

- n: La cantidad de nodos
- *solucion*: Secuencia que contiene la clique solución
- *candidatos*: Secuencia con los nodos a considerar
- $fronteraMax$: El cardinal de la frontera de la clique solución

Las funciones "EsClique()" y "Frontera()" son las mismas que en el algoritmo exacto. Dado que se usan en todos los algoritmos, son omitidas. Tienen complejidad $O(n^3)$ y $O(n^2)$ respectivamente.

```
function RESOLVER
    LeerInput()                                  ▷ O(n^2)
    solucion ← ∅                                 ▷ O(1)
    candidatos ← {1, 2, 3, .., n}                ▷ O(n)
    fronteraMax ← −1                             ▷ O(1)
    ObtenerSolucion()                            ▷ O(n^5)
```

```
function ObtenerSolucion
    while candidatos ≠ ∅ do                                          ▷ O(n^5)
        mejorCandidato ← −1                                          ▷ O(1)
        for c ∈ candidatos do                                        ▷ O(n^4)
            solucion.push(c)                                         ▷ O(1)
            if ¬EsClique(solucion) then                              ▷ O(n^3)
                solucion.erase(c)                                    ▷ O(n)
                candidatos.erase(c)                                  ▷ O(n)
            else
                fronteraActual ← Frontera(solucion)                  ▷ O(n^2)
                if fronteraActual ≥ fronteraMax then                 ▷ O(1)
                    fronteraMax ← fronteraActual                     ▷ O(1)
                    mejorCandidato ← c                               ▷ O(1)
                else
                    candidatos.erase(c)                              ▷ O(n)
                solucion.erase(c)                                    ▷ O(n)
        if mejorCandidato ≠ −1 then                                  ▷ O(1)
            solucion.push(mejorCandidato)                            ▷ O(1)
            candidatos.erase(mejorCandidato)                         ▷ O(n)
    return solucion
```

3.3. Complejidad

La complejidad principal del algoritmo se encuentra en "ObtenerSolucion()", que es $O(n^5)$. Esto es fácil de ver: como nunca se agregan candidatos y siempre se quita al menos uno, el ciclo exterior se ejecuta a lo sumo $O(n)$ veces. El *For* recorre la lista de candidatos, que son también a lo sumo $O(n)$. Dentro del *For* encontramos solo dos funciones grandes, "EsClique" de $O(n^3)$ y "Frontera" de $O(n^2)$. El resto son comparaciones, asignaciones y borrados, de $O(n)$.

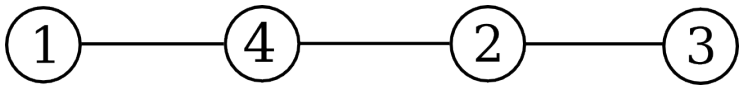
Más intuitivamente, EsClique ($O(n^3)$) se ejecuta $O(n)$ veces en el For, y a su vez se ejecuta $O(n)$ veces en el While. Por lo tanto, es $O(n^5)$.

Si bien $O(n^5)$ puede parecer un polinomio "grande", es exponencialmente mejor que el algoritmo exacto que tenía $O(2^n * n^3)$. El trade-off es que no siempre se producen soluciones óptimas.

3.4. Optimalidad

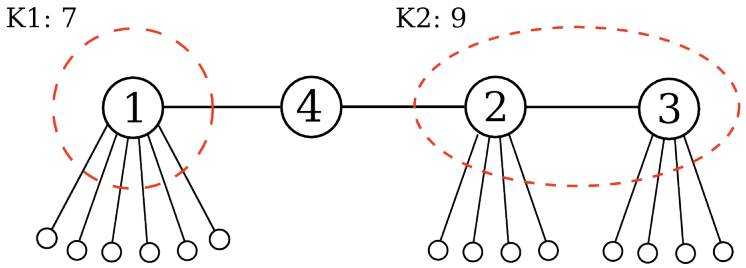
Como podrá verse más adelante, las soluciones para grafos aleatorios son bastante buenas, pero existe al menos un tipo específico de grafos donde la diferencia entre el exacto y el greedy puede ser **arbitrariamente grande**.

Veamos como podemos construir este tipo de grafos.

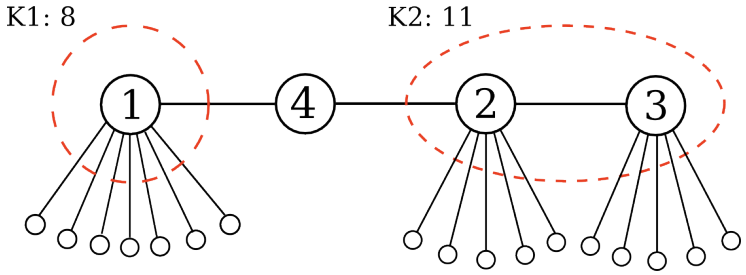


Grafo inicial

Agreguemos 6 vecinos al nodo 1 y agreguemos 4 vecinos a los nodos 2 y 3. El algoritmo goloso toma el primer nodo como solución inicial, es el que tiene más vecinos. Luego intenta agregar nodos a la solución de forma tal que formen clique y que tengan mayor frontera. Esto no puede suceder: lo mejor que puede obtener el algoritmo goloso es una clique con frontera de tamaño 7. Sin embargo, podemos ver que la solución óptima está formada por la clique con los nodos 2 y 3, con una frontera de tamaño 9.



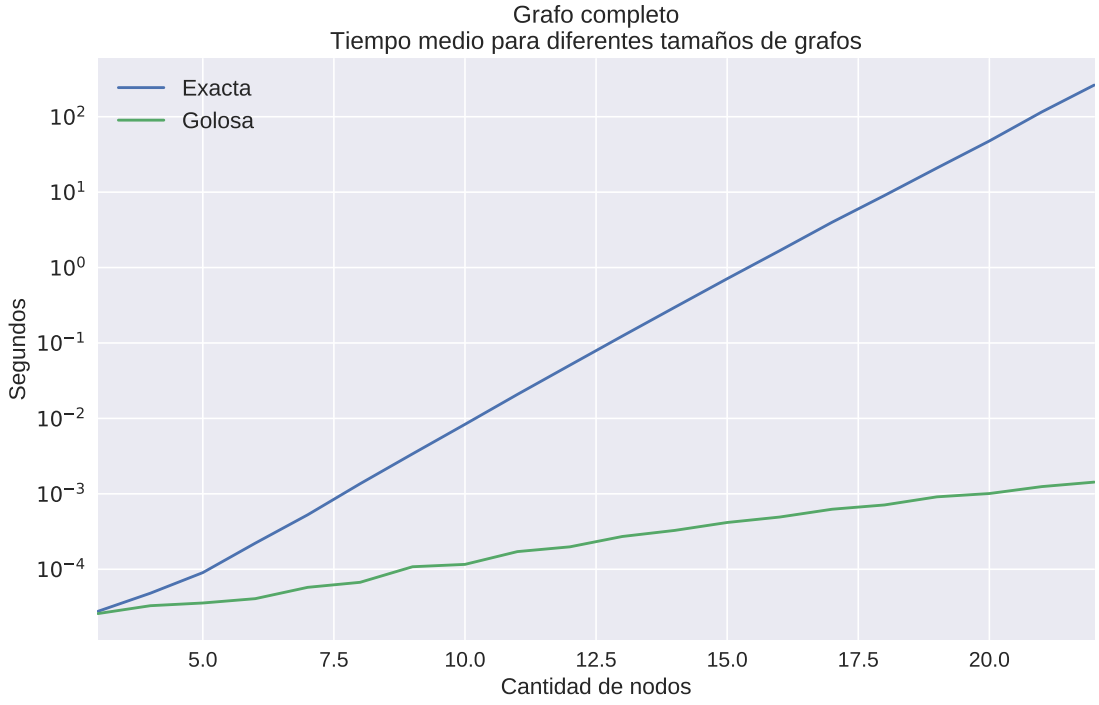
Este es un procedimiento que podemos seguir repitiendo. Agregamos un vecino al nodo 1, y un vecino a los nodos 2 y 3. La frontera máxima golosa aumenta en 1, mientras que la frontera máxima aumenta en 2. De esta forma, podemos construir grafos donde la diferencia entre la solución óptima y la solución golosa sea arbitrariamente grande. Haremos una demostración más formal al final de la sección.



3.5. Experimentación

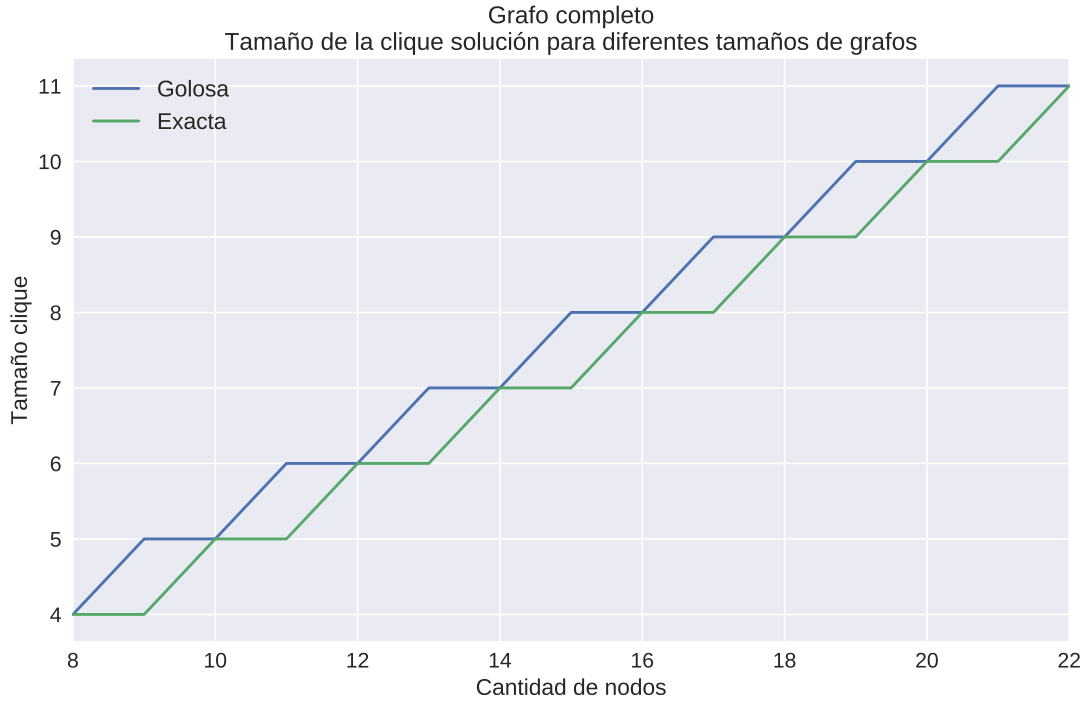
La principal motivación de buscar heurísticas fue disminuir el tiempo de cómputo. Analizando la cota temporal teórica vimos que $O(n^5)$ es exponencialmente mejor que $O(2^n * n^3)$. Veamos cómo son los tiempos en la práctica, considerando un grafo completo para el análisis de peor caso.

Dado que el algoritmo greedy es extremadamente rápido, decidimos mostrar un gráfico en **escala logarítmica** sobre los tiempos. Lo que puede verse es lo esperado.

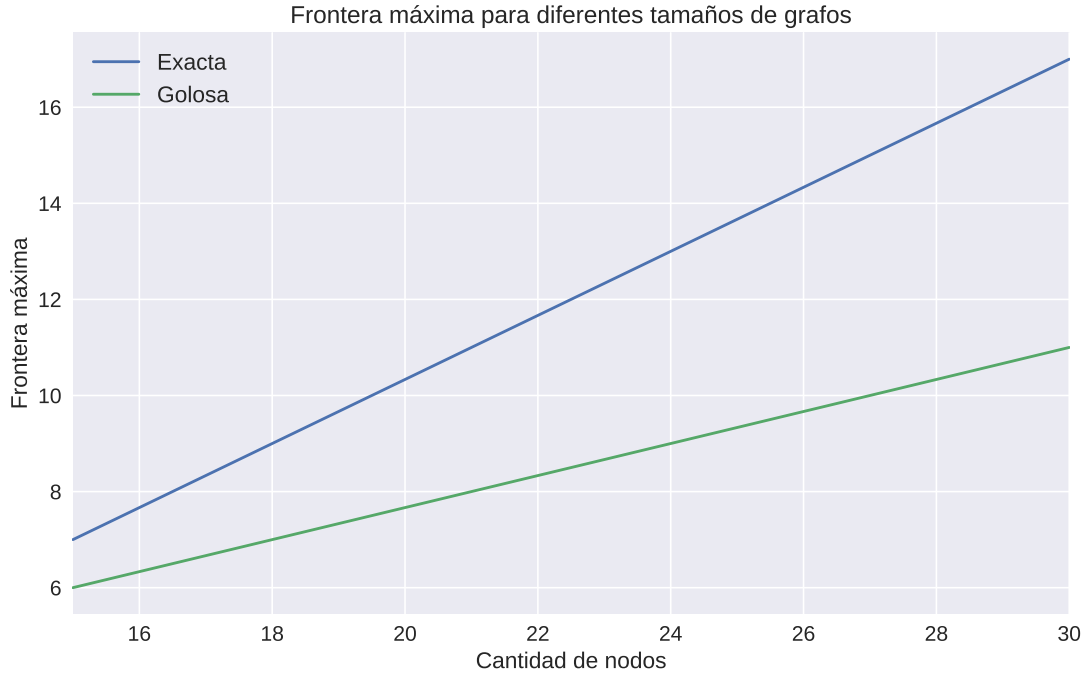


↑ Escala logarítmica

Si bien en el caso de los grafos completos, la frontera solución coincide con la óptima ($\#K_{n/2}$), el tamaño de la clique solución no es igual. Puede verse que el algoritmo greedy se queda con cliques más grandes que la estrictamente necesaria, y esto se debe a que la condición de selección incluye un $\geq$.



Retomando el análisis con los casos malos, habíamos mostrado que existen al menos un tipo de grafos (desde ahora simplemente **"grafos malos"**) donde la solución golosa podía estar tan lejos de la óptima como se quiera. Veamos gráficamente la diferencia en las soluciones.



↑ Frontera para casos malos

Puede verse que a medida que n crece, ambas soluciones se van alejando. No nos quedemos con la intuición, consideremos una demostración más formal:

Dado que estos "grafos malos" tienen su construcción bien definida, calculemos analíticamente cuál es el valor de la solución óptima para un tamaño de grafo n dado. Esto nos será de utilidad para analizar optimalidad en secciones posteriores.

Como la solución óptima se encuentra en la clique formada por los nodos 2 y 3, consideremos el proceso de construcción. En cada paso se agregan 3 nodos, cada nuevo nodo como vecino de 1, 2 y 3 respectivamente. Podemos ver que 2 de cada 3 nodos contribuyen a aumentar nuestra solución inicial, por lo tanto, la solución óptima tiene que ser *aproximadamente* $\frac{2}{3}n$, con alguna constante por los nodos iniciales.

Podemos comprobar empíricamente, observando los valores de frontera obtenidos por nuestro algoritmo exacto, que la solución óptima para "grafos malos" es exactamente $f(n) = \frac{2}{3}n - 3$. Este es un valor bien definido para n divisibles por 3, dado que siempre los contruimos agregando 3 nodos por vez.

Con un razonamiento análogo, de cada 3 nodos que se agregan, un único nodo se agrega a lo que será la solución golosa. De hecho, puede verse que la solución golosa para un grafo malo es exactamente $g(n) = \frac{1}{3}n + 1$, también bien definido para n divisibles por 3.

Veamos finalmente qué sucede con la diferencia entre ambas soluciones a medida que crece n:

$$g(n) - f(n)$$
$$(\frac{2}{3}n - 3) - (\frac{1}{3}n + 1)$$
$$\frac{2}{3}n - 3 - \frac{1}{3}n - 1$$
$$\frac{1}{3}n - 4$$
$$\lim_{x \to \infty} (\frac{1}{3}n - 4) = \infty$$

□

Demostramos entonces que la diferencia entre la solución óptima y la solución greedy para un mismo grafo puede ser arbitrariamente grande. Es decir, dado un n lo suficientemente grande las soluciones pueden ser tan diferentes como uno quiera. La motivación de las siguientes heurísticas será intentar de mejorar las soluciones en los grafos malos, para intentar acercarnos lo más posible al óptimo.

Para finalizar este apartado, es importante notar que en promedio el algoritmo greedy da resultados razonablemente buenos, sobre todo teniendo en cuenta el mínimo costo computacional que tiene asociado. Veremos más sobre casos promedios al final del informe, donde realizaremos una comparación entre las diferentes heurísticas.

4. Búsqueda Local

4.1. Explicación

Búsqueda local es una estrategia utilizada para mejorar una solución obtenida mediante una heurística. La idea es dar una definición de vecindad para toda solución (por ejemplo, si mis soluciones son nodos, los vecinos pueden ser sus adyacentes), y buscar el candidato óptimo dentro de ese conjunto. Este proceso puede repetirse para tender hacia un máximo local.

Para este problema particular consideramos que todo clique es una solución posible, y que en su vecindad se encuentran aquellos cliques que pueden obtenerse aplicando una operación de swap (reemplazar un nodo en la clique por uno que no está), una operación de agregado (de un nodo) o una operación de borrado (de un nodo). En cada iteración del algoritmo, revisamos toda la vecindad y nos quedamos con la clique de mayor frontera. Una vez que nos detenemos en un máximo local o iteramos suficientes veces, obtenemos la solución final de la búsqueda local.

4.2. Pseudocódigo

Las funciones EsClique() y Frontera() no son incluidas aquí por ser iguales a las incluidas previamente. Las complejidades son $O(n^3)$ y $O(n^2)$ respectivamente.

```
function RESOLVER(solucion, iteraciones)                          ▷ O(iteraciones * n^5)
    fronteraMaxima ← Frontera(solucion)                           ▷ O(n^2)
    for i ∈ [1..iteraciones] do                                   ▷ O(iteraciones * n^5)
        solucionActual ← BusquedaLocal(solucion)                  ▷ O(n^5)
        fronteraActual ← Frontera(solucionActual)                 ▷ O(n^2)
        if fronteraActual > fronteraMaxima then                   ▷ O(1)
            fronteraMaxima ← fronteraActual                       ▷ O(1)
            solucion ← solucionActual                             ▷ O(1)
    return solucion
```

```
function BUSQUEDALOCAL(solucionInicial)                           ▷ O(n^5)
    complementoInicial ← Complemento(solucionInicial)             ▷ O(n)

    solucionSwap ← MaximoPorSwap(solucionInicial, complementoInicial)   ▷ O(n^5)
    fronteraSwap ← Frontera(solucionSwap)                         ▷ O(n^2)

    solucionAdd ← MaximoPorAdd(solucionInicial, complementoInicial)     ▷ O(n^4)
    fronteraAdd ← Frontera(solucionAdd)                           ▷ O(n^2)

    solucionSub ← MaximoPorSub(solucionInicial)                   ▷ O(n^4)
    fronteraSub ← Frontera(solucionSub)                           ▷ O(n^2)

    solucionSuprema ← solucionSwap                                ▷ O(1)
    fronteraSuprema ← fronteraSwap                                ▷ O(1)

    if fronteraAdd > fronteraSuprema then                         ▷ O(1)
        fronteraSuprema ← fronteraAdd                             ▷ O(1)
        solucionSuprema ← solucionAdd                             ▷ O(1)

    if fronteraSub > fronteraSuprema then                         ▷ O(1)
        fronteraSuprema ← fronteraSub                             ▷ O(1)
        solucionSuprema ← solucionSub                             ▷ O(1)

    return solucionSuprema                                        ▷ O(1)
```

Al swappear no nos quedamos con el primer swappeo que se pueda, sino que vemos las n^2 combinaciones de swaps para elegir la mejor (siempre que el swap mantenga una clique como solución).

```
function MaximoPorSwap(solucionInicial, complementoInicial)                    ▷ O(n^5)
    candidatos ← solucionInicial
    maxFrontera ← Frontera(solucionInicial)                                    ▷ O(n^2)
    max_i ← −1                                                                 ▷ O(1)
    max_j ← −1                                                                 ▷ O(1)

    for i ∈ [0..|solucionInicial|) do                                          ▷ O(n^5) Ver sección Complejidad
        for j ∈ [0..|complementoInicial|) do
            candidato[i] ← complementoInicial[j]                               ▷ Hacemos swap O(1)
            if EsClique(candidatos) then                                       ▷ O(n^3)
                fronteraCandidato ← Frontera(candidato)                        ▷ O(n^2)
                if fronteraCandidato > maxFrontera then                        ▷ O(1)
                    max_i ← i                                                  ▷ O(1)
                    max_j ← j                                                  ▷ O(1)
                    maxFrontera ← fronteraCandidato                            ▷ O(1)
            candidato[i] ← solucionInicial[j]                                  ▷ Restauro swap O(1)
    if max_i ≠ −1 then                                                         ▷ O(1)
        candidato[max_i] ← complementoInicial[max_j]                           ▷ Swapeo definitivamente O(1)
    return candidatos                                                          ▷ O(1)
```

En "MaximoPorSub" calculamos la frontera al sacar un único nodo (lo hacemos para todos los nodos) y vemos si la frontera aumenta. Puede pasar que al disminuir el tamaño de una clique queden muchas aristas "libres" que hagan que la frontera aumente. Probamos con todos y nos quedamos con la respuesta que mejoró la frontera, si es que hubo alguna.

```
function MaximoPorSub(solucionInicial)                                         ▷ O(n^4)
    candidatos ← solucionInicial
    maxFrontera ← Frontera(solucionInicial)                                    ▷ O(n^2)
    max_c ← −1
                                                                               ▷ O(1)
    for c ∈ candidatos do                                                      ▷ O(n^4)
        if EsClique(candidatos − {c}) then                                     ▷ O(n^3)
            fronteraCandidato ← Frontera(candidatos − {c})                     ▷ O(n^2)
            if fronteraCandidato > maxFrontera then                            ▷ O(1)
                max_c ← c                                                      ▷ O(1)
                maxFrontera ← fronteraCandidato                                ▷ O(1)

    if max_c ≠ −1 then                                                         ▷ O(1)
        candidatos ← (candidatos − {max_c})                                    ▷ O(1)
    return candidatos
```

En "MaximoPorAdd" probamos agregar de a un nodo, viendo que forme clique y calculando la nueva frontera. Si la frontera aumenta, nos guardamos la nueva solución.

```
function MaximoPorAdd(solucionInicial, complementoInicial)                    ▷ O(n^4)
    maxFrontera ← Frontera(solucionInicial)                                   ▷ O(n^2)
    max_c ← −1                                                                ▷ O(1)
    candidatos ← solucionInicial                                              ▷ O(1)

    for c ∈ complementoInicial do                                             ▷ O(n^4)
        if EsClique(candidatos + {c}) then                                    ▷ O(n^3)
            fronteraCandidato ← Frontera(candidatos + {c})                    ▷ O(n^2)
            if fronteraCandidato > maxFrontera then                           ▷ O(1)
                max_c ← c                                                     ▷ O(1)
                maxFrontera ← fronteraCandidato                               ▷ O(1)

    if max_c ≠ −1 then                                                        ▷ O(1)
        candidatos ← (candidatos + {max_c})                                   ▷ O(1)
    return candidatos                                                         ▷ O(1)
```

Para calcular el complemento, armamos un vector de booleanos que dicen si un nodo pertenece o no a la solución input. Luego recorremos el vector y armamos un nuevo vector con aquellos nodos que no estaban en la solución original.

```
function Complemento(solucionInicial)
    pertenece[i] = True ⟺ i ∈ solucionInicial                                 ▷ O(n)
    complemento ← {}                                                          ▷ O(1)
    for i ∈ [0..n) do                                                         ▷ O(n)
        if pertenece[i]=True then                                             ▷ O(1)
            complemento.PushBack(i)                                           ▷ O(1) amortizado
    return complemento                                                        ▷ O(1)
```

4.3. Complejidad

Las complejidades de cada linea fueron marcadas en el pseudocódigo, por lo que calcular la complejidad total es bastante sencillo.

- MaximoPorAdd: $O(n^4)$. Son $O(n)$ iteraciones de algo $O(n^3)$.
- MaximoPorSub: $O(n^4)$. Son $O(n)$ iteraciones de algo $O(n^3)$.
- MaximoPorSwap: $O(n^5)$. El primer For recorre un vector v y el For anidado recorre su complemento. Tanto v como su complemento tienen $O(n)$ elementos cada uno, por lo que en total se recorren $O(n^2)$ pares. Son $O(n^2)$ iteraciones de algo $O(n^3)$.
- BusquedaLocal: $O(n^5)$. Esto es por la complejidad de MaximoPorSwap, que es la mayor, ya que todo el resto es $O(n^4)$ o menor.
- Resolver: $O(\#iteraciones * n^5)$. Son $\#iteraciones$ veces de BusquedaLocal.

Por lo tanto, mejorar una solución inicial con este algoritmo de búsqueda local cuesta $O(\#iteraciones * n^5)$.

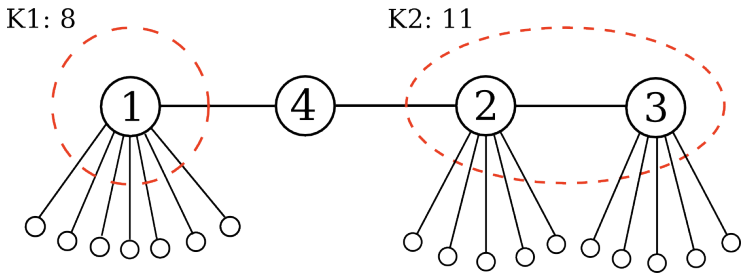
4.4. Optimalidad

Algo que notamos al probar este método es que el algoritmo goloso presentado en la sección anterior ya llega a un máximo local, por lo que es redundante aplicarle una búsqueda local para mejorarla. ¿Por qué ocurre esto? Porque si hubiese un nodo al que swappear con alguno de afuera para conseguir una mejor frontera, como este algoritmo siempre se queda con el mejor nodo fuera de la clique, entonces el nodo de afuera ya tendría que haber sido elegido. Tampoco podemos agregar nodos, pues si pudiesemos tomar más, el algoritmo goloso ya lo hubiera hecho.

Sin embargo, aún así podemos encontrarle un uso. En la sección de Optimalidad del algoritmo goloso notamos que la diferencia entre la solución proporcionada por el algoritmo y la solución real puede ser arbitrariamente grande, con lo cuál pueden haber casos muy malos. Una forma de paliar este problema es que el algoritmo tome como entrada un nodo inicial, y que a partir de ahí construya una solución golosa. Llamaremos a esta variante *golosoB*, y al original *golosoA*. Por cuestiones de tiempo y del alcance de este informe no detallaremos la implementación de *golosoB*, pero es el algoritmo que motivará este apartado. De todas maneras, no es más que lo que acabamos de decir: fijar un nodo incial antes de aplicar el algoritmo.

¿Por qué sirve nuestro algoritmo de búsqueda local para *golosoB*? Esto ocurre porque el primer nodo no fue elegido golosamente, es un nodo arbitrario que fijamos en nuestra solución. Como consecuencia, una vez que la solución golosa queda construida, eliminar o swappear el primer nodo podría llegar producir un aumento en la frontera. Si esto llegase a ocurrir, aparecerían nuevos nodos candidatos a entrar en nuestra solución que no podiamos tomar antes porque no estaban conectados con el primer nodo. De esta forma, podemos continuar moviéndonos por las soluciones para llegar a algún máximo local incluso mejor que el nos devolvió *golosoA*.

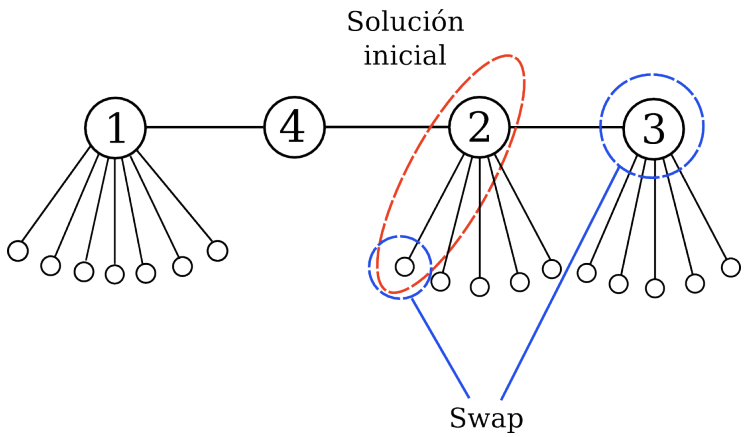
Tomemos el mismo ejemplo que complicaba a *golosoA* utilizando *golosoB* utilizado en *búsquedaLocal*.

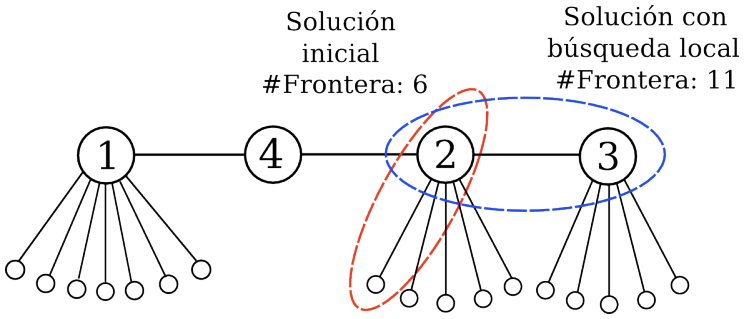


Si a *golosoB* le pasamos como entrada un nodo perteneciente al arbol 1, o el nodo 4; es imposible mejorar la solución de *golosoA*. Es el riesgo que tenemos al dejar libre el nodo inicial. Es importante notar que la solución **no** es peor, sino que simplemente no se mejora (aunque en otros grafos, con un mal nodo inicial, podría llegar a ocurrir que *golosoB* y *busquedaLocal* devuelvan un peor resultado que *golosoA*). Desarrollaremos un poco más en unos párrafos. Sin embargo, si elegimos un nodo cualquiera del arbol 2 o 3, la situación es distinta.

El caso fácil es si elegimos los nodos 2 o 3, en donde *golosoB* se queda con la solución $[2, 3]$, ya que ni las hojas ni el nodo 4 mejoraran la frontera de ese par de nodos.

Más complicado es lo que ocurre al tomar una hoja de los arboles 2 y 3 como inicial. Asumamos que tomamos una hoja del 2 (tomar una hoja del 3 es análogo). La hoja sólo está conectada con el 2, por lo que el 2 se incluye en la solución. Una vez realizado esto, como nadie más está conectado con la hoja, *golosoB* devuelve $[hoja, 2]$. Es en este caso en donde *búsquedaLocal* ayuda, pues a partir de $[hoja, 2]$ busca los cliques que se pueden obtener haciendo swaps, con lo cual considera como candidato a la clique producto de swappear la *hoja* con el 3, que es la clique $[2, 3]$. Al ser la solución óptima, el algoritmo la toma en la primera iteración, y corta en la segunda. Termina devolviendo lo que queriamos: $[2, 3]$.



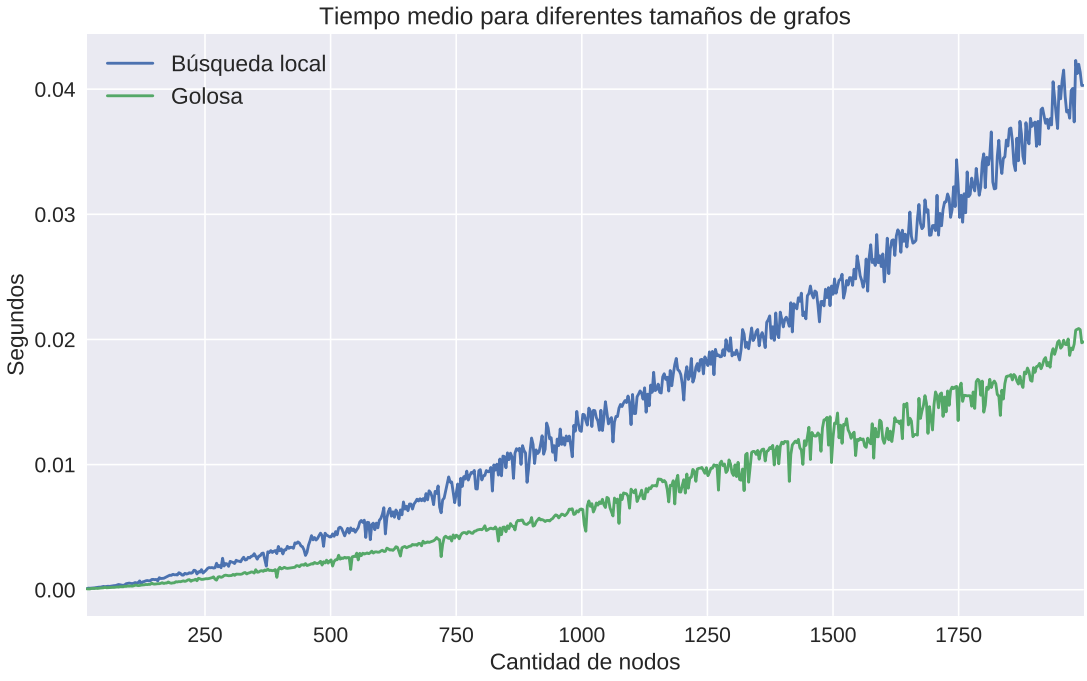


Retomando lo mencionado hace unos párrafos, ¿que sucede si se elije un nodo del árbol 1? Claramente *golosoB* se vuelve tan malo como *golosoA*, pero **existe una solución muy sencilla**: podríamos elegir al azar el nodo inicial y correr el algoritmo muchas veces, para quedarnos con la mejor. Con suficiente cantidad de elecciones, es esperable que se elimine el problema de empezar con nodos que no pueden seguir mejorándose. Exploraremos esta idea en la siguiente sección del informe: *GRASP*.

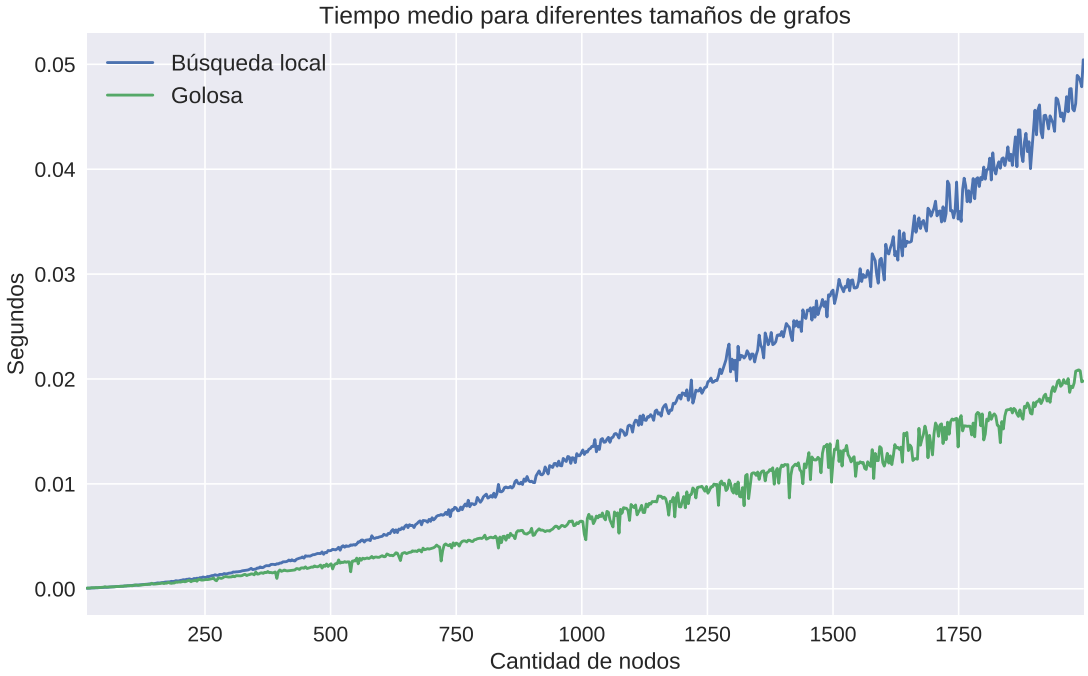
4.5. Experimentación

Como contamos antes, el algoritmo goloso que utilizamos en búsqueda local depende de un nodo inicial. Creemos que la muestra mas representativa para *grafos malos* se obtiene partiendo de un nodo al azar. Lo que hicimos para conseguir los datos es, para cada tamaño de n, correr 50 veces el experimento y quedarnos con la media de los datos, lo que explica las fuertes variaciones de la curva. Con esto queda claro que esta técnica en promedio proporciona mejores resultados, pero es muy dependiente del nodo inicial. Intentaremos solucionar este problema en el siguiente apartado.

Es claro que al aplicar búsqueda local se agregan muchas operaciones extra que antes no existían, por lo que es esperable que el tiempo de ejecución aumente. Posibles preguntas son ¿Cómo influye en esto la cantidad de iteraciones? ¿Aumenta el tiempo al punto de hacerlo inutilizable? **Los siguientes gráficos fueron realizados tomando el grafo malo que presentamos anteriormente**.



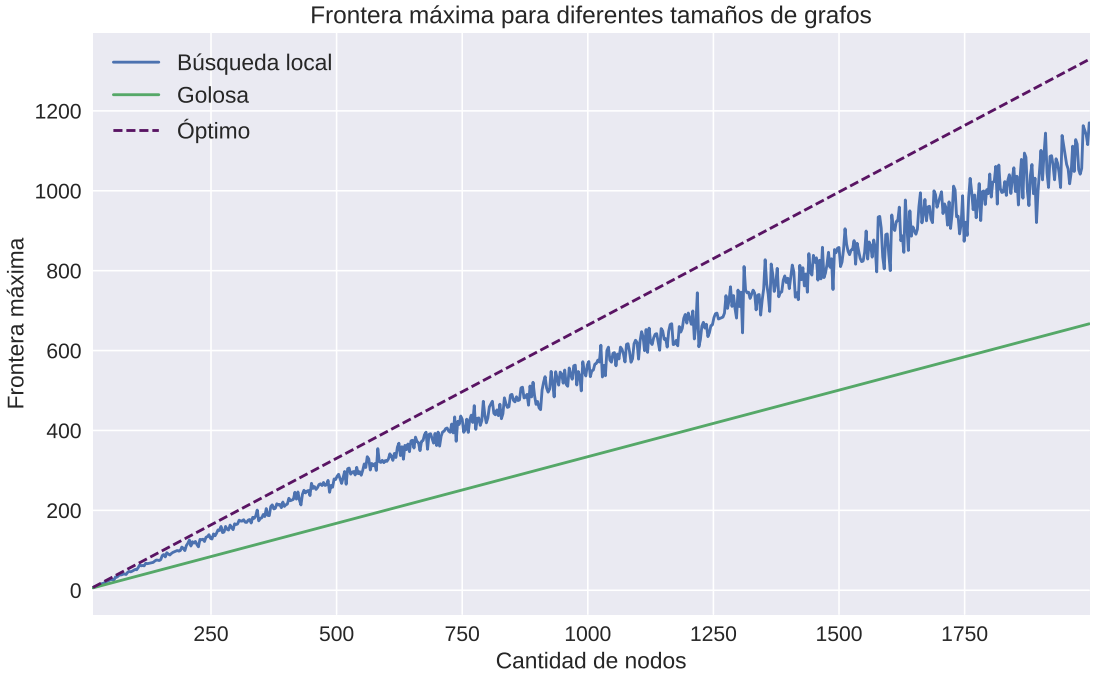
↑ 10 iteraciones.



↑ 2000 iteraciones.

El primer gráfico fue realizado con 10 iteraciones, mientras que el segundo fue obtenido con 2000 iteraciones. Si bien el tiempo en el de 2000 es mayor, lo es solo por una *pequeña* diferencia. Lo que sucede es que nuestro algoritmo deja de ejecutarse cuando se estanca en un extremo local, pues una vez dentro ya no tiene forma de escapar.

La siguiente pregunta que surge fue, ¿mejoramos la solución del caso malo? Considerando una búsqueda local de 2000 iteraciones, veamos como es la solución promedio.



Recordemos que por construcción, en la sección anterior pudimos calcular analíticamente cuál era la solución óptima para cada n de nuestros *grafos malos*. Recordemos también que lo que graficamos es **el promedio** de las soluciones, es decir, *en realidad hay soluciones que alcanzan al óptimo*, pero muchas otras que no. Con una cantidad suficiente de repeticiones vemos que el promedio se encuentra aproximadamente en el centro: por ese motivo decimos que en promedio no consigue la solución óptima.

En general con búsqueda local se logró una mejora con respecto a la solución golosa, pero no lo suficiente. Sigue siendo un método que está condenado a caer en extremos locales, y una vez dentro no puede escapar. La motivación de la siguiente sección será intentar resolver esta falencia.

5. Grasp

5.1. Explicación

GRASP es sigla de Procedimientos Golosos Aleatorios Adaptativos de Búsqueda (Greedy Randomized Adaptive Search Procedures). Es una metaheurística que se basa en utilizar métodos golosos constructivos y de búsqueda local para resolver problemas computacionalmente difíciles, utilizando el azar para evitar estancarse en un único máximo local. Cada iteración de un algoritmo GRASP construye golosamente una solución desde cero, admitiendo cierto grado de aleatoriedad al añadir elementos, para después mejorarla con un método de búsqueda local. Realizamos varias iteraciones deteniéndonos bajo algún criterio de corte, ya sea por que conseguimos una solución lo suficientemente buena o porque nos pasamos de una cantidad de iteraciones predeterminada. La mejor solución conseguida entre todas las iteraciones es el resultado final del algoritmo. [2]

El algoritmo goloso constructivo que vamos a utilizar va a ir un poco más allá del algoritmo *golosoB* del que hablamos en la sección anterior. En vez de maximizar de forma directa nuestra función greedy (cardinalidad de la frontera) sobre los nodos no agregados, construiremos un conjunto de candidatos llamado RCL (Restricted Candidate List). La misma contendrá los candidatos que cumplan con un cierto nivel de optimalidad, del que será elegido un nodo al azar para ser agregado a la solución final.

El criterio con el cuál se tomarán candidatos al RCL estará dictaminado por un parámetro *alpha*, que indicará qué porcentaje del rango de valores posibles que pueden tomar los candidatos se admiten en la RCL. Formalmente, un nodo c que resulta en una frontera f_c al ser agregado a la solución parcial, pertenece al RCL solo si $f_c \geq f_{max} + \alpha * (f_{min} - f_{max})$, con max nodo localmente óptimo, y min nodo localmente peor. A grandes rasgos, un *alpha* grande implica darle más peso al azar, y uno más chico, a la porción golosa del algoritmo. (Notemos que para que esto tenga sentido, $\alpha \in \mathbb{R}$ y $0 \leq \alpha \leq 1$). Una vez que el RCL está formado, se elije un nodo al azar, y se comienza el proceso de nuevo, hasta no poder continuar.

Obtenida una solución inicial, como nuestro método no es (en general) completamente goloso, podemos utilizar un algoritmo de búsqueda local para movernos hacia un máximo local, mejorando nuestra solución y completando una iteración de GRASP. El algoritmo que usamos para este propósito es el mismo de la sección anterior.

5.2. Pseudocódigo

Las funciones local.resolver(), EsClique() y Frontera() no son incluidas aquí por ser iguales a las incluidas previamente. Las complejidades son $O(n^5)$, $O(n^3)$ y $O(n^2)$ respectivamente.
Referencias de variables globales para el pseudocódigo:

- n: La cantidad de nodos
- *solucion*: Secuencia que contiene la clique solución

```
function RESOLVER(α, repsGrasp, repsLocal)
    fronteraMax ← 0                                              ▷ O(1)
    fronteraNueva ← 0                                            ▷ O(1)
    for reps ∈ [1..repsGrasp] do                                 ▷ O(repsGrasp * repsLocal * n^5)
        actual ←GreedyRandom(α)                                  ▷ O(n^5)
        nueva ← local.resolver(actual, repsLocal)                ▷ O(repsLocal * n^5)
        fronteraNueva ← Frontera(nueva)                          ▷ O(n^2)
        if fronteraNueva > fronteraMax then                      ▷ O(1)
            solucion ← nueva                                     ▷ O(1)
            fronteraMax ← fronteraNueva                          ▷ O(1)
    return solucion
```

Para calcular el RCL hacemos un primer pasaje para calcular el máximo y el mínimo de los nodos candidatos a agregarse al clique parcial. En una segunda pasada es cuando vamos agregando los nodos a medida que encontramos aquellos que cumplen con la fórmula planteada en la Explicación.

```
function GreedyRandom(α)
    fronteraMax ← −1                                                          ▷ O(1)
    candidatosInicial ← {1..n}                                                ▷ O(n)
    solucion ← ∅                                                              ▷ O(1)
    puedoConstruirClique ← True                                               ▷ O(1)

    while puedoConstruirClique ∧ |candidatosInicial| > 0 do                   ▷ O(n^5)
        puedoConstruirClique ← False                                          ▷ O(1)
        candidatos ← ∅                                                        ▷ O(1)
        RCL ← ∅                                                               ▷ O(1)

        for c ∈ candidatosInicial do                                          ▷ O(n^4)
            if EsClique(solucion + {c}) then                                  ▷ O(n^3)
                front ← Frontera(solucion)                                    ▷ O(n^2)
                candidatos ← candidatos + {(front, c)}                        ▷ O(1)
                puedoConstruirClique ← True                                   ▷ O(1)

        fronteraMinTmp ← ∞                                                    ▷ O(1)
        fronteraMaxTmp ← −1                                                   ▷ O(1)
        for cand ∈ candidatos do                                              ▷ O(n)
            fronteraMinTmp ← min(fronteraMinTmp, cand.first)                  ▷ O(1)
            fronteraMaxTmp ← max(fronteraMaxTmp, cand.first)                  ▷ O(1)

        for cand ∈ candidatos do                                              ▷ O(n)
            if cand.first ≥ fronteraMaxTmp + α ∗ (fronteraMinTmp − fronteraMaxTmp) then  ▷ O(1)
                RCL ← RCL + {cand.second}                                     ▷ O(1)

        if puedoConstruirClique then                                          ▷ O(1)
            randomIndex ← random({1..|RCL|})                                  ▷ O(1)
            solucion ← solucion + {RCL[randomIndex]}                          ▷ O(1)
            candidatosInicial ← candidatosInicial − {RCL[randomIndex]}        ▷ O(1)

    fronteraMax ← Frontera(solucion)                                          ▷ O(1)
    return solucion
```

5.3. Complejidad

Analicemos en primer lugar la complejidad de "GreedyRandom":

- Iteraciones del while son $O(n)$: Se corta cuando $candidatosInicial.size = 0$, y en cada iteración a $candidatosInicial$ se le resta RCL, que nunca está vacío pues siempre el nodo golosamente óptimo pertenece, por lo cual la cantidad de iteraciones es $O(n)$.
- Calcular fronteras de candidatos es $O(n^4)$: Recorremos la lista de candidatos ($O(n)$), y para cada uno preguntamos si es clique en $O(n^3)$ y si lo es, su frontera en $O(n^2)$.
- Calcular fronteras máximas y mínimas es $O(n)$: Calculadas todas las fronteras de los candidatos, solo hay que recorrer esa secuencia y quedarme con el máximo y el mínimo
- Construir el RCL es $O(n)$: Consta de recorrer los candidatos y verificar para cada uno si se cumple una fórmula calculable en $O(1)$

Entonces, su complejidad es:

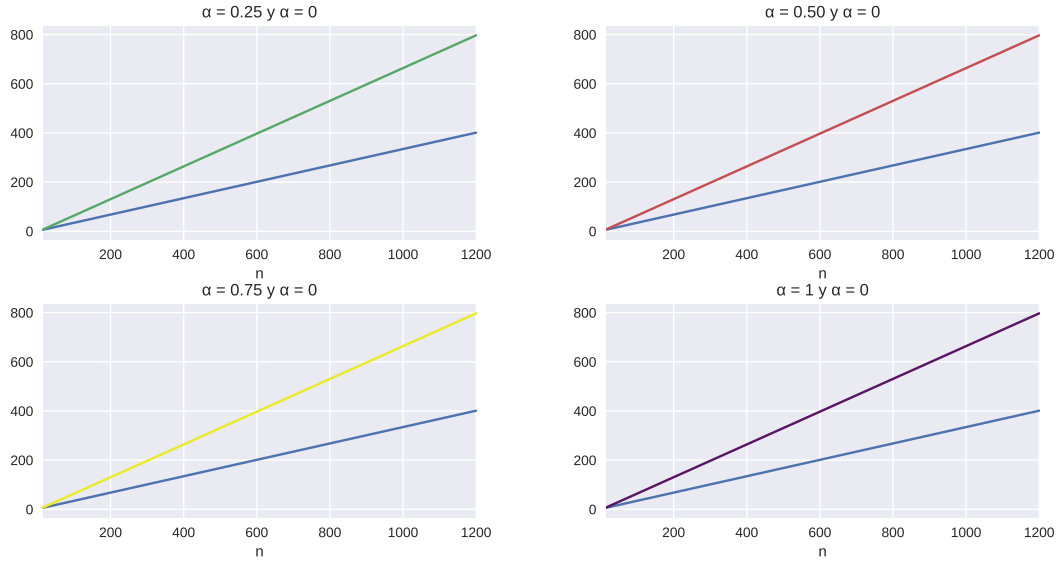
$$O(n) * (O(n^4) + O(n^3) + O(n)) = O(n^5)$$

Con esta información en mano, el grueso de la complejidad de GRASP proviene simplemente de aplicar "GreedyRandom" y luego "BusquedaLocal" que son ambas $O(n^5)$ (aunque busqueda local también depende de una cantidad de iteraciones), un número $repsGrasp$ de veces. Por ende, la complejidad final del algoritmo es $O(repsGrasp * repsLocal * n^5)$.

5.4. Experimentación

El nivel de aleatoriedad con el que se crea la RCL del GRASP depende de nuestro parámetro α. $\alpha = 0$ representa una elección puramente greedy, mientras que $\alpha = 1$ hace una elección puramente aleatoria. Dado nuestros grafos son bastantes particulares, al introducir un poco de aleatoriedad con el alpha ya logramos obtener las mejores fronteras posibles. Puede verse que, como adelantamos, cuando alpha es 0 se produce una elección puramente greedy, por lo que siempre es peor.

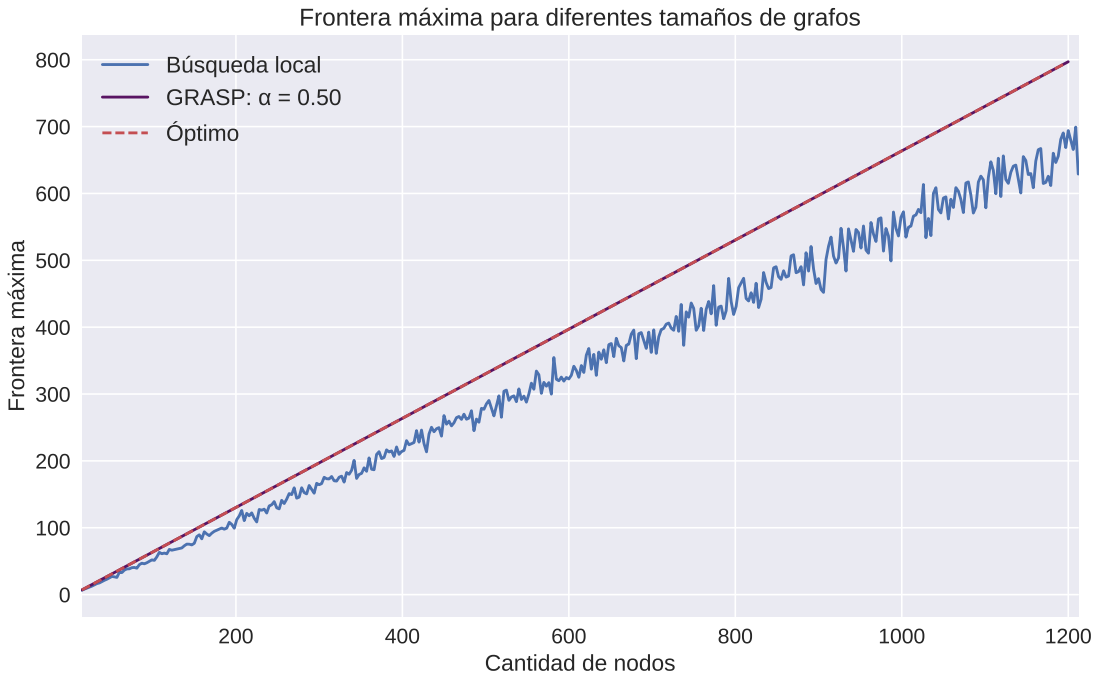
En el siguiente gráfico mostramos diferentes valores de α comparados con $\alpha = 0$. Puede observaser que todos son mejores al greedy. Más aún, los mostramos con 4 plots diferentes porque todos los $\alpha \neq 0$ poseen el mismo valor y al incluirlos en un mismo gráfico no pueden distinguirse. En eje x representa la cantidad de nodos mientras que el eje y representa la frontera solución. El tipo de grafos utilizado son nuestros "grafos malos".



Es importante aclarar que esto es así por el tipo particular de grafos que estamos tratando, en el caso general esto puede no ser cierto. Intuitivamente los mejores resultados se obtienen en el equilibrio, con α =0.5, por lo que en los siguientes casos utilizaremos este valor.

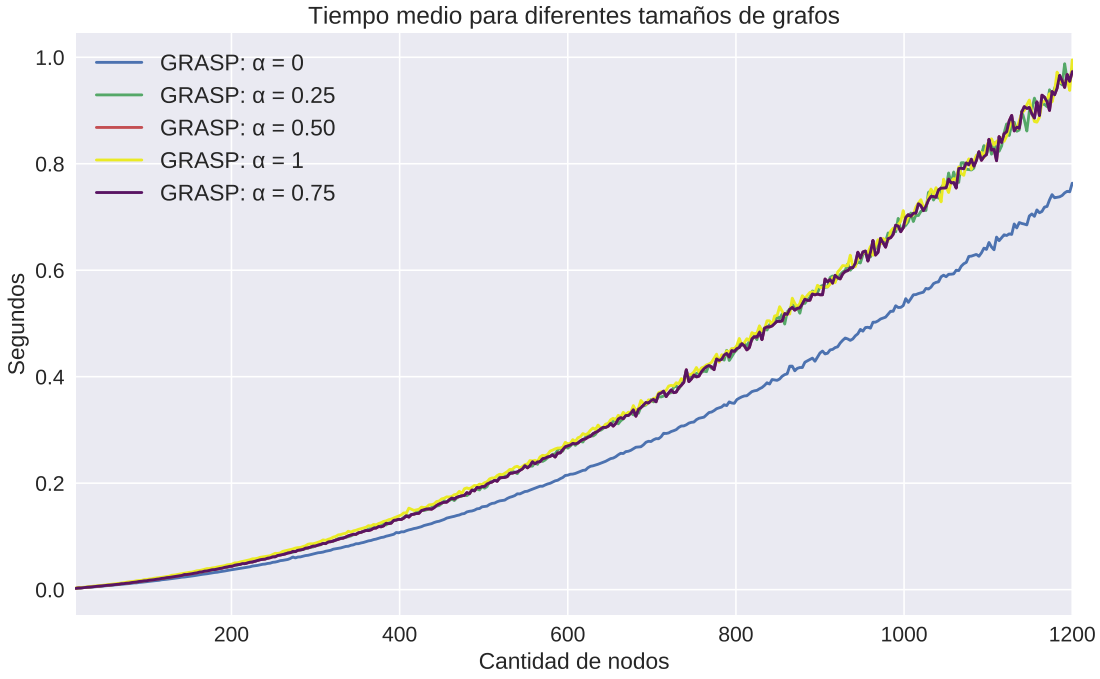
Otro parámetro importante es la cantidad de iteraciones a realizar, tanto en GRASP como en la búsqueda local interna. Para búsqueda local utilizamos $repsLocal = 2000$, al igual que en el apartado anterior. Con $repsGrasp$ fuimos variándolo y no encontramos ninguna diferencia apreciable, creemos que es por el tipo de grafo. Para estos experimentos tomamos $repsGrasp = 50$.

Nos interesa averiguar entonces qué tan lejos estamos del óptimo para nuestros "grafos malos". Estamos interesados en saber si la estrategia para escapar de los extremos locales funciona. Veamos lo que nos muestran los datos, recordando que anteriormente pudimos calcular cuál es la solución óptima para este tipo de grafos.



¡La solución promedio de GRASP es **exactamente** la solución óptima!

Logramos resolver el problema de forma óptima para el caso en los que las demas técnicas fallaban. Es esperable que GRASP también sea bastante bueno para grafos en general. Veremos qué tan cierta es esta intuición en la siguiente sección. Antes de concluir con este apartado, es interesante considerar cuál es el costo temporal de utilizar GRASP dependiendo el valor de α. Aquí unas comparaciones:

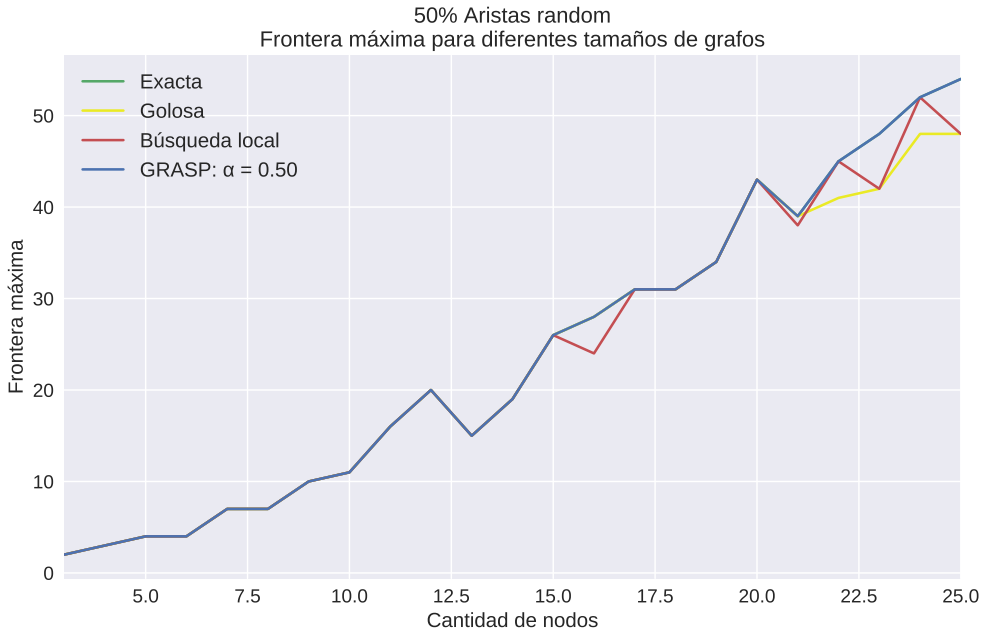


6. Experimentación general

Dado que nuestros análisis estaban enfocados en los peores casos, es momento de considerar cómo son nuestras soluciones si consideramos casos promedios.

Consideramos que un grafo de n nodos es promedio cuando generamos sus aristas al azar. Esto significa que las conexiones entre nodos será aleatoria, pero que la cantidad de aristas estará predeterminada con algun porcentaje, para poder separar mejor los diferentes casos de análisis. En particular, mostraremos los casos donde hay 50 % y 75 % de aristas. Demás porcentajes resultaron muy poco interesantes por tener muy pocas aristas o demasiadas. Como última aclaración, para todas las instancias de búsqueda local, la cantidad de repeticiones usadas es 2000, a menos que se mencione explícitamente.

Nuestra intención es comparar exactamente qué tan mejores o peores son los diferentes algoritmos utilizados. Para esto, para cada tamaño de n generamos un grafo al azar (con cierto porcentaje de aristas) y comparamos las soluciones obtenidas por todas las técnicas. Lo que sigue no son promedios, sino las soluciones finales para algun caso aleatorio dado.

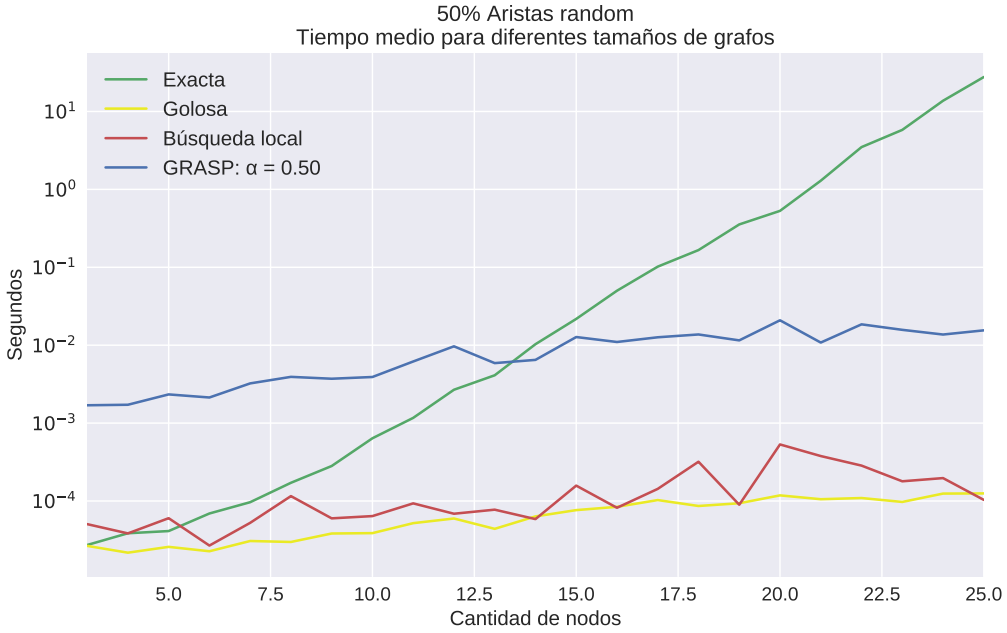


↑ Exacta y GRASP comparten la misma curva

Los resultados son los que nuestros análisis anteriores apuntaban. Debemos tener en cuenta que son valores de n pequeños (para poder comparar con el algoritmo exacto), veremos más adelante comparaciones para n mayores.

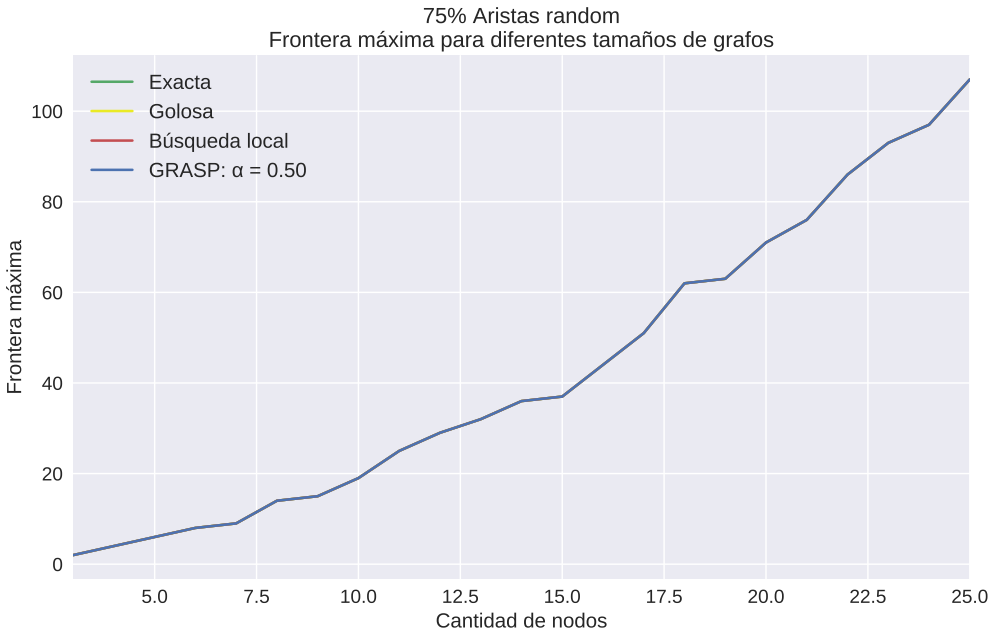
Observamos que GRASP es la mejor técnica de las presentadas para resolver el problema. De hecho, comparte la misma curva que el algoritmo exacto. Le siguen el algoritmo de búsqueda local, y por último el goloso. Como sospechábamos, si bien en el análisis para un "grafo malo" el greedy era extremadamente malo, aquí podemos notar que para grafos en general conseguimos una aproximación bastante buena.

Analicemos tambíen que pasa con los tiempos de cómputo. Como era esperable, el tiempo del algoritmo exacto crece muy rápidamente, así que mostramos el gráfico en *escala logarítmica*. En general son tiempos esperados, pero es interesante notar que GRASP tiene una constante asociada mucho mas elevada que los demás.

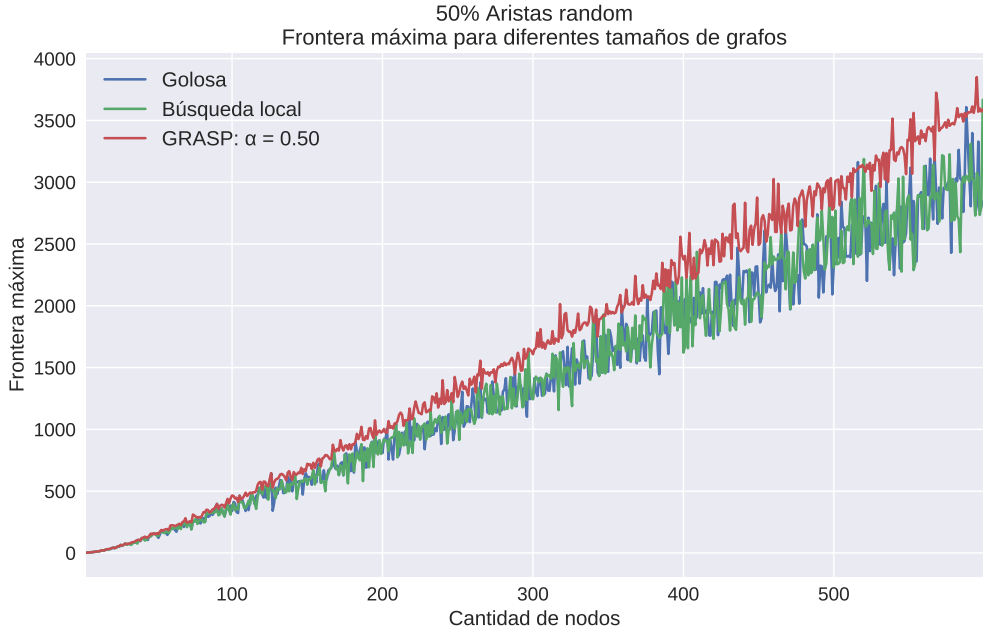


↑ Escala logarítmica

Repetimos el experimento pero para grafos con 75 % de aristas. Sin embargo, todos los algoritmos nos dan la misma solución (¡La óptima!). Esto tiene que ver con la densidad del grafo, y probablemente porque estamos viendo tamaños pequeños.

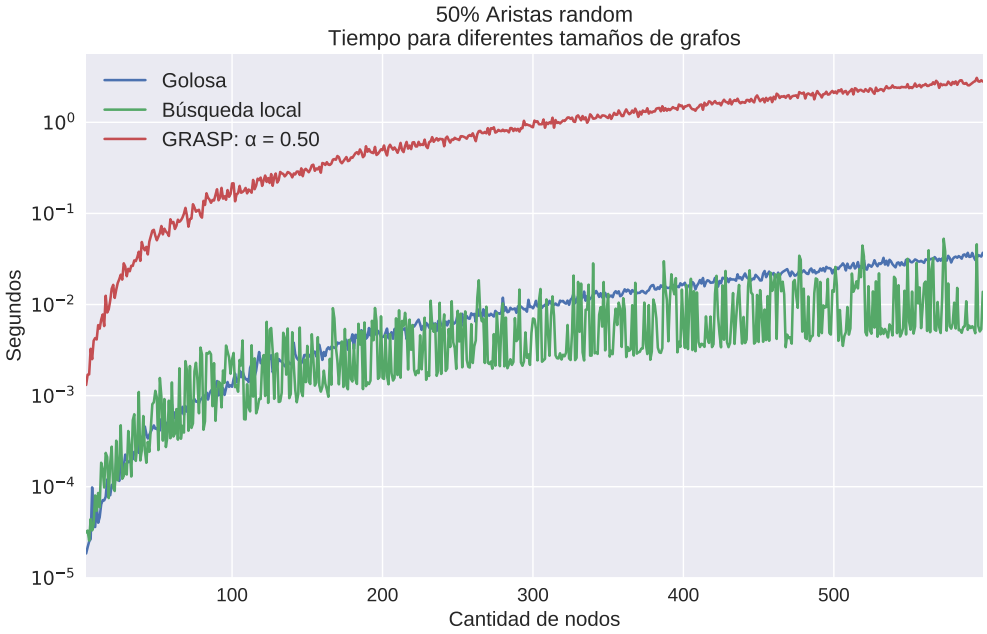


Veamos que sucede si hacemos crecer n. Dado que no podemos realizar las pruebas para el algoritmo exacto debido a su complejidad temporal, no lo incluimos en el gráfico. No tenemos ninguna certeza de cuán cerca o lejos están de la solución óptima, pero es interesante observar como varían las diferentes soluciones. No incluímos 75 % aristas pues es idéntico al de 50 %.



Como esperábamos, GRASP es la mejor alternativa que tenemos para resolver el problema de CMF. El componente random sumado a la cantidad de repeticiones hacen que sea superior a los demás algoritmos, en términos de aproximación a la solución óptima. Variando la cantidad de *repsGrasp* no obtuvimos diferencias significativas. En el gráfico se muestra con $repsGrasp = 50$.

GRASP es mejor, ¿pero a qué costo? Mostramos el gráfico en *escala logárítmica* debido al tamaño de los tiempos de GRASP en comparación a los demás. Podemos observar claramente que el costo temporal es incluso mayor al esperado, dónde para $n = 600$ se demora aproximadamente 3 segundos. Esto entra en linea con la constante que habiamos observado en el anterior gráfico de tiempos. Es más que razonable, ya que estamos repitiendo dos algoritmos $O(n^5)$ en este caso 50 veces.



El caso de tiempos de 75 % aristas no lo mostramos por ser idéntico en forma al anterior, pero en ese caso, con $n = 600$, GRASP tarda aproximadamente 17 segundos. Pueden disminuirse la cantidad de iteraciones para obtener una mayor eficiencia en términos del tiempo, pero a medida que n crece GRASP es el primero que se vuelve inutilizable.

7. Conclusión

El objetivo de este trabajo era la investigación de diferentes técnicas para resolver el problema de Clique de Máxima Frontera. Sabemos que no se conoce hasta el momento ningún algoritmo exacto que lo resuelva en tiempo polinomial, por lo que mostramos una única forma de resolverlo de forma exacta, con complejidad temporal exponencial.

Vimos que el algoritmo exacto es inutilizable en la práctica para grafos con mas de 30 nodos, así que en la práctica es necesario recurrir a heurísticas que resuelvan el problema.

Consideramos un algoritmo sencillo, **greedy**, para intentar conseguir una buena solución. Lo que logramos encontrar fue un algoritmo exponencialmente mas rápido que en general da aproximaciones razonables. Encontramos además, al menos un tipo particular de grafos, los "grafos malos", para los cuales demostramos que la diferencia entre la solución óptima y la solución greedy tendía a infinito.

Intentando ahora poder solucionar el problema de "grafos malos", consideramos una heurística de **búsqueda local** para mejorar soluciones preexistentes, en dónde analizamos distintas formas de movernos a soluciones similares. Si bien en ocasiones logramos mejorar el método anterior, seguimos cayendo en el problema de que una vez llegados a un extremo local ya nos era imposible salir de ahí pues ninguna solución en su vecindad la mejoraría.

Para finalizar, nos basamos en el paper citado [2] para construir una solución que use el método GRASP. Con la introducción de variables aleatorias en conjunto con cierta estrategia golosa, logramos escapar del problema de los extremos locales pudiendo lograr así soluciones óptimas (con una gran probabilidad) en nuestros grafos originales.

Una vez analizadas las distintas alternativas, probamos todos los algoritmos en una serie de grafos aleatorios, con diferentes porcentajes de aristas. Pudimos comprobar empíricamente lo analizado antes: GRASP produce los mejores resultados. Sin embargo, nos fue imposible dar una cota de distancia con respecto a la solución óptima porque no tenemos manera de conocer cuál es ésta solución sin poder correr el algoritmo exacto.

8. Bibliografía

[1] Ina Koch, *Enumerating all connected maximal common subgraphs in two graphs*

[2] Thomas A. Feo and Mauricio G. C. Resende. *Greedy randomized adaptive search procedures*