DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

# Trabajo Práctico 3

25 de junio de 2017

Algoritmos y Estructuras de Datos III
Primer Cuatrimestre de 2017

**Grupo "Dijkstraidos"**


| Integrante | LU | Correo electrónico |
|---|---|---|
| Barylko, Roni Ariel | 750/15 | rbarylko@dc.uba.ar |
| Giudice, Carlos | 694/15 | cgiudice@dc.uba.ar |
| Szperling, Sebastián Ariel | 763/15 | sszperling@dc.uba.ar |
| Tarrío, Ignacio | 363/15 | itarrio@dc.uba.ar |





**Facultad de Ciencias Exactas y Naturales**
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

# Índice

# 1. Descripcion de situaciones reales

El problema que vamos a analizar durante todo el informe es el de *clique de máxima frontera*(*CMF*) en un grafo, que consiste en hallar la clique tal que su frontera sea de cardinalidad máxima. Definimos la frontera de una clique K como el conjunto de aristas que tienen un extremo en K y otro por fuera de la clique.

Distintas situaciones problemáticas de la vida real podrían ser representadas por CMF, tales como:

- MarencoGames desarrolla un juego online y quiere que haya la menor cantidad de gente haciendo trampa. En este juego los jugadores juegan a través de la red, pero no todos están comunicados entre sí. Los desarrolladores quieren hacer un sistema de protección de trampa, por lo que deciden convertir varios jugadores en moderadores, los cuales van a reportar cuales de los participantes a los que están conectados están haciendo trampa. Los moderadores tienen que estar todos conectados entre sí para reportarse las trampas, y lo que se desea es cubrir la mayor cantidad de jugadores posibles. Cada jugador puede ser representado con un nodo, y la conexión entre dos jugadores a través de aristas. Con esta representación, tendríamos que una clique en el grafo podrá ser un conjunto de moderadores conectados entre sí, y la frontera será la cantidad de jugadores que abarca el sistema de protección.

# 2. Algoritmo exacto

## 2.1. Desarrollo

Al querer generar un algoritmo que nos de una respuesta exacta, y como no tenemos límites de complejidad ni alguna certeza con respecto al grafo de entrada, consideramos que el mejor enfoque es, simplemente, encontrar todas las cliques y chequear cual de ellas tiene la máxima frontera.

Para esto, es importante hacernos una idea general de los pasos a seguir, para ver no solo por qué esta solución nos dará un resultado preciso, sino también para visualizar los subproblemas donde el costo temporal se tornará alto. Para esto, dividiremos nuestro problema original en dos subproblemas:

- Encontrar todas las cliques
- Ordenarlas por tamaño de frontera (encontrar la mayor)

Si nos centramos primero en el tamaño de la frontera, considerando una clique $X$, veremos que esta suma implica simplemente tomar todos los ejes del grafo y, para cada arista $e$, sumarla si tiene un extremo adentro y otro afuera de $X$. Sin embargo, esto implicaría recorrer $m$ aristas cada vez que queremos obtener la frontera de una clique, por lo que sería óptimo recorrer estas aristas una sola vez y luego, considerando el grado de cada nodo y la clique a la que pertenece, calcular la frontera.

Para analizar mejor el cálculo a realizar, imaginemos una clique de un solo nodo c. Esta clique tiene frontera $D_c$, donde $D_c$ es el grado de c (porque todos los nodos adyacentes a él son fronterizos). Supongamos que le agregamos un nodo e. Ahora la clique pasa a tener frontera $D_c + D_e$ - 2, puesto que ahora contamos todos los nodos adyacentes a $c$ menos $e$, y a eso le sumamos todos los nodos adyacentes a $e$ menos $c$. Es decir, sumamos los grados de todos los nodos pertenecientes a la clique, y le restamos la suma de las aristas contadas que forman parte de la clique. Este último número es facil de obtener, puesto que en una clique cualquiera de N nodos, cada uno de sus vértices debe tener una arista hacia los otros (N-1) nodos (puesto que sino no sería una clique). Por ende, acabaríamos teniendo la siguiente suma:

$$\text{Para toda clique C, con } v_i \in \text{C}$$
$$\text{Frontera(C)} = (\ \textstyle\sum_{i=0}^{N} d(v_i)\ ) \text{ - N } \times \text{(N-1)}$$

Ahora, si reformulamos esta cuenta, tenemos

$$\text{Frontera(C)} = \textstyle\sum_{i=0}^{N} (d(v_i) + 1 \text{ - N})$$

Por ende, bastará con calcular el grado de cada vértice y luego, para cada clique, realizar la sumatoria correspondiente.

Ahora, nos queda conseguir todas las cliques del grafo dado. Para esto, es importante notar que todo grafo completo G con n $\geq$ 1 tiene, al menos, (n-1) subgrafos completos, lo cual es facil de ver: pensemos un grafo completo de N nodos. Todos los nodos tienen grado $N-1$ porque son adyacentes a los demás nodos del grafo, por lo que si sacamos un nodo $c$ cualquiera junto a todos sus ejes, los demás nodos pasarán a tener grado $N-2$ en un grafo de N-1 nodos. Por ende, seguiría siendo un grafo completo con un nodo menos que el original (es decir, un subgrafo completo de N-1 nodos). Esta misma operación podría realizarse varias veces, lo cual nos dejaría con cada uno de los (n-1) subgrafos completos existentes.

Sin embargo, hay dos cosas que remarcar: nuestro grafo completo G es en realidad una clique de un grafo aún más grande, y cada uno de los nodos pertenecientes al grafo G es distinguible (al menos, en los términos de nuestro problema). Por lo tanto, como consideramos que sacar dos nodos $c$ y $e$ de G nos devuelve dos subgrafos distintos aún siendo isomorfos, tenemos que ser más exhaustivos con la cantidad de cliques a obtener. Por ende, si tomamos de nuevo al grafo completo G con N nodos, y consideramos todos los grafos completos que se pueden formar, tendremos:

- Tamaño 1: cada uno de los grafos triviales (es decir, N grafos)
- Tamaño 2: cada combinación posible de dos nodos sobre los N nodos de G

  **...**
- Tamaño N-1: cada combinación posible de N-1 nodos sobre los N nodos de G
- Tamaño N: el grafo G original

Es decir, si generamos todos los subgrafos completos de G de tamaño i, con i entre 0 y N, tendremos la combinatoria entre i y N: $\binom{N}{i}$. Por lo tanto, al generar todos los subgrafos completos del grafo G, obtendremos la cuenta:

$$\sum_{i=0}^{N} \binom{N}{i}$$

Donde cada uno de estos subgrafos completos es, por definición, una clique a considerar.

Por lo tanto, para encontrar todas las cliques mencionadas realizaremos una búsqueda invertida. Es decir, en vez de empezar por la clique más grande y obtener todos los subgrafos de ella, tomaremos cada uno de los nodos como una clique de tamaño 1 y extenderemos cada una de ellas de forma que, llegado el momento donde no se puedan agregar nodos, hayamos alcanzado la clique de tamaño máximo que puede alcanzar ese nodo. Con esta idea, cada vez que agregando un nodo obtengamos un subgrafo completo lo agregaremos a la lista de cliques y seguiremos agregando nodos desde él, para así limitarnos a obtener cada clique una única vez y reducir la complejidad tanto del código como temporal.

Sin embargo, pensemos el caso en el cual tenemos los nodos 1, 2 y 3 donde todos los nodos tienen aristas entre sí. Como expusimos anteriormente, la idea es empezar teniendo tres cliques triviales (de un único nodo) y extenderlas nodo a nodo. Por lo tanto, con el nodo 1 generaremos la clique (1,2) y la clique (1,3). De ellas dos, generaremos dos veces la clique (1,2,3). A su vez, el nodo 2 generará la clique (2,3) y (2,1), las cuales generarán dos veces más la clique (1,2,3), etc.

Si bien el hecho de generar varias veces la misma clique no es un problema en cuanto a soluciones (al fin y al cabo, nuestra CMF seguirá siendo la misma y la alcanzaremos), el tiempo que tarde la generación y el análisis de las cliques crecerá de manera dramática si permitimos que esto ocurra. Por lo tanto, utilizaremos una matriz de adyacencia triangulada que considere solo una de las dos conexiones. Así, si volvemos a considerar el caso inicial, el nodo 1 generará las cliques (1,2) y (1,3), pero el nodo 2 generará solo (2,3) y el nodo 3 no generará ninguna clique. De este modo, nos aseguramos una reducción importante en los costos temporales y de memoria. Así, acabaremos teniendo un algoritmo de este estilo:

```
agregarTodasLasCliques (in matrizAdyacencia : matriz, in n : nat) → res: conjunto(clique)
  res ← AgregarCliquesTriviales(matrizAdyacencia)
  for j < n do
  |  Agregar(res, expandirClique(j, matrizAdyacencia, n, res))
  end for
```

```
expandirClique (in subclique : clique, in matrizAdyacencia : matriz, in n : nat, in/out cliques : conjunto(clique))
  for i < n do
  |  if esAdyacenteATodos(i, subclique, matrizAdyacencia) then
  |  |  clique: nuevaClique ← agregarNodoAClique(subclique, i)
  |  |  Agregar(cliques, nuevaClique)
  |  |  expandirClique(nuevaClique, matrizAdyacencia, n, cliques)
  |  end if
  end for
```

Es importante resaltar que, como explicamos anteriormente, nuestra matriz esta triangulada. Por ende, cuando nos fijamos si un nodo es adyacente a todos lo hacemos mirando su columna de la matriz, y si dos nodos son adyacentes solo una de las dos columnas tendrá el dato como verdadero. Asi evitamos la obtención de la misma clique varias veces.

Finalmente, con los dos subproblemas ya solucionados, solo nos queda ver cada clique y quedarnos con la de mayor frontera. Como obtuvimos todas las cliques posibles, y nos estamos quedando con la que tiene más nodos fronterizos, esta acaba siendo CMF, y por ende, la solución.

## 2.2. Cota temporal

Repasemos, de manera más puntillosa, los pasos a seguir en nuestro algoritmo. El primer paso importante será obtener todas las cliques, para lo cual necesitaremos el grado de todos los nodos y una matriz de adyacencia triangulada. Ambos pasos nos costarán $O(m)$, donde $m$ es la cantidad de aristas pertenecientes al grafo original, mientras que la creación de la matriz tendrá un costo de $O(n^2)$, por lo cual acabaríamos teniendo $O(m) + O(m) + O(n^2)$, y como m $\leq$ n $\times$ (n-1) tenemos $O(m) + O(m) + O(n^2) \subseteq O(3(n^2)) \subseteq O(n^2)$

Ahora si, pasemos a analizar los dos subproblemas mencionados en el desarrollo:

**Agregar todas las cliques**

Este algoritmo toma cada una de los nodos, genera una clique trivial y agrega recursivamente los demás nodos para generar nuevas cliques y agregarlas a la lista. Esto quiere decir que, pensando en el peor caso, por cada llamada a

expandirClique tendremos $n$ llamadas nuevas, lo cual nos acabaría dejando con una cota de $O(n^n)$. Sin embargo, como usamos una matriz triangular, vimos que solo se genera una vez cada clique. Por ende, suponiendo que nuestro grafo inicial fuera completo, tendríamos $\sum_{i=0}^{n} \binom{n}{i}$ llamadas, y por el **Teorema del Binomio**, sabemos que:

$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^k b^{n-k}$$

Por lo tanto, con a = 1, b = 1, tendríamos:

$$(1+1)^n = \sum_{k=0}^{n} \binom{n}{k} 1^k 1^{n-k}$$

Lo cual acaba siendo $2^n$ y es exactamente igual a la cantidad de llamadas que realizamos para el grafo completo de tamaño n. Por ende, la complejidad de nuestro algoritmo acaba siendo $O(2^n)$.

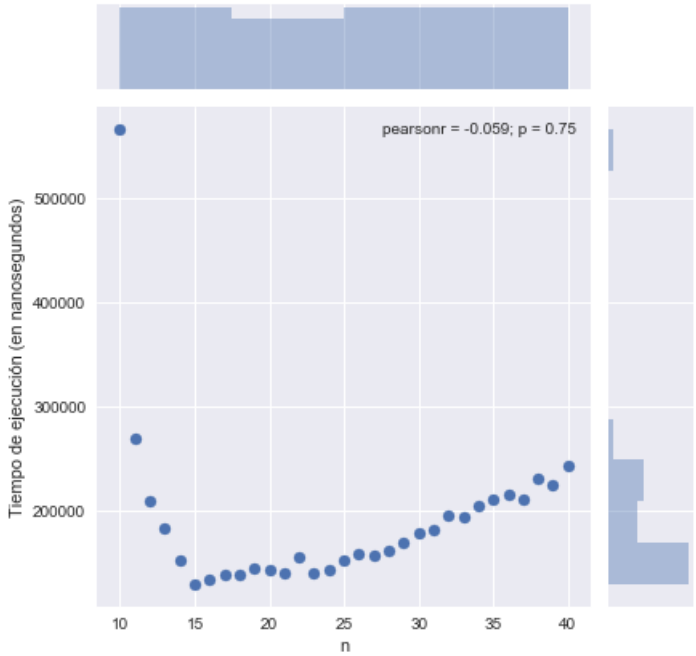
**Encontrar CMF**

Como vimos, en el peor caso tenemos $2^n$ cliques. Sabemos que para encontrar la mejor deberemos recorrerlas todas, y sumar la frontera de cada una de ellas. Esto significará que, para cada clique, deberemos recorrer los $k$ nodos que la componen. Y como en el peor caso tendremos una clique de $n$ nodos, nuestra cota temporal se reducirá al costo de recorrer $2^n$ veces $n$ nodos, es decir: $O(2^n \times n)$

Finalmente, considerando las complejidades vistas, nuestra cota quedará en $O(n^2) + O(2^n \times n)$, que acaba siendo $O(n^2 + 2^n \times n)$.

## 2.3. Experimentación

Al experimentar con el algoritmo exacto, nos encontramos con la dificultad de generar grafos con distribuciones similares de aristas. Al realizar un primer análisis, nos encontramos con la siguiente gráfica:

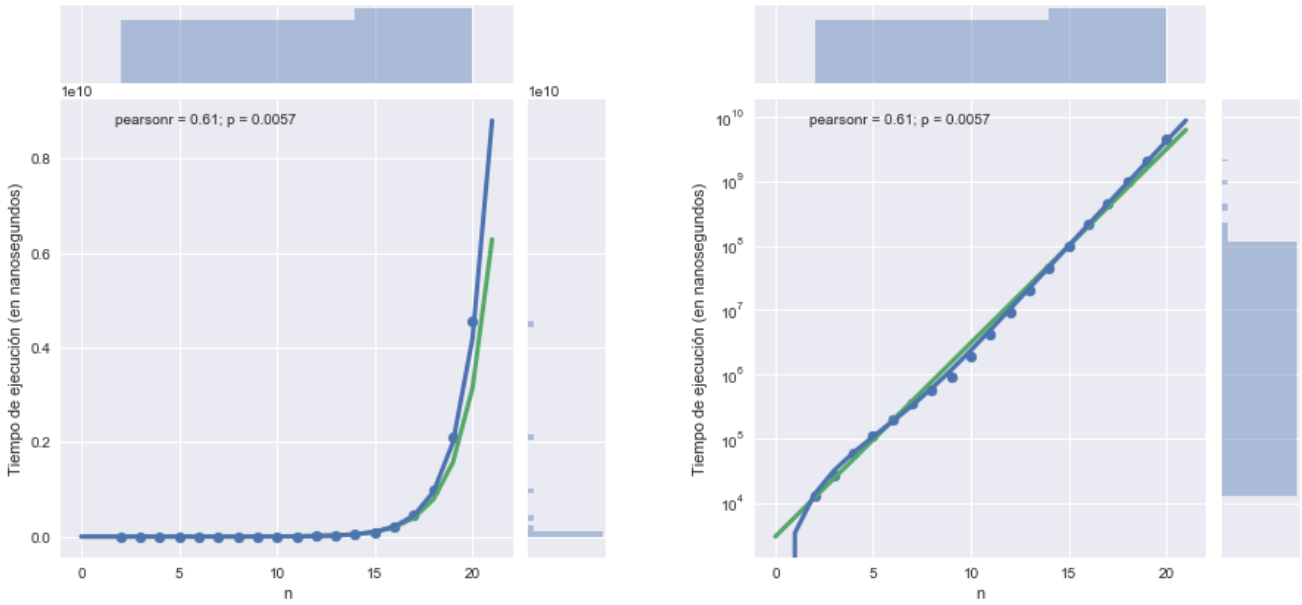



Datos del gráfico

| m = 40 |
|---|

Si bien nuestro análisis dio una cota de complejidad superior en base a $n$, no debemos descartar el hecho que un grafo con más nodos y misma cantidad de aristas puede poseer menos cliques. Por lo tanto, al generar cliques de mayor tamaño en base a otras más pequeñas, o al calcular la frontera de todas las cliques halladas, la cantidad de cliques a procesar puede ser mucho menor.

Por este motivo, decidimos analizar en particular el caso de los grafos completos. Allí podemos estudiar de manera determinística la cota propuesta:




Datos de los gráficos

- $f(x) = 200 * x * 2^x + 3000 * (x^2)$
- $f(x) = 3000 * 2^x$

Aqui podemos ver que el tiempo de ejecución se ajusta correctamente a la complejidad propuesta. A la derecha presentamos la misma gráfica en escala logarítmica, que nos permite apreciar la naturaleza exponencial del problema.

# 3. Heurística constructiva golosa

## 3.1. Introduccion

Dado que la complejidad del algoritmo exacto resulta prohibitiva en la práctica, podemos aplicar heurísticas que resuelvan el problema en tiempo polinomial, a cuestas de la calidad final de la solución.

Las heurísticas tienen un factor importante de intuición sobre por qué podrían llegar a funcionar mejor o peor y no justificación formal. Debido a esto es difícil decidir qué criterio utilizar puesto que siempre es posible encontrar instancias para las cuales la heurística sea mala.

Nosotros decidimos encarar el criterio de una manera simple y segura, agregar nodos a la clique que nos aumenten la frontera de nuestra clique actual, hasta que no nos queden más aristas por agregar a la clique que nos aumenten la frontera.

## 3.2. Desarrollo

A la hora de aplicar esta heurística sobre nuestro problema, debimos poner el foco sobre donde queríamos realizar la construcción golosa. Rápidamente, y considerando que al utilizar la idea greedy debemos siempre tomar la decisión que mejora nuestra solución de manera óptima a corto plazo (es decir, que dada una clique de tamaño $n$, se obtiene la mejor solución de tamaño $(n+1)$ que contiene a la clique anterior), notamos que la forma acertada de aplicar esta heurística sería, en cada paso, agregar el nodo de mayor grado que genere una clique y mejore la frontera. Es decir que en el primer paso tomaremos siempre al nodo de mayor grado (puesto que esta es la clique de máxima frontera de tamaño 1) y en cada paso le agregaremos el nodo de mayor grado que sea adyacente a toda la clique y que mejore la frontera.

Es importante, antes de seguir, marcar algunas cosas importantes sobre nuestra solución. Para empezar, consideremos que cada vez que agregamos un nodo nuevo a la clique, debemos sumar el grado del nuevo nodo y restar dos veces el tamaño de la subclique (una vez para descontar las aristas que van del nuevo nodo a la clique, y otra para descontar las aristas que iban de cada nodo de la clique al nuevo nodo). Simplificando, tenemos que realizar el siguiente cálculo:

$$\text{Para } V_i \text{ vértice que se agrega a subclique } Frontera(Clique) = Frontera(Subclique) + d(V_i) - tam(subclique) \times 2$$

Como vemos, la única variable que cambia cada vez que agregamos un nodo a la clique es el grado del mismo. Por lo tanto, sabemos que si agregar el nodo de mayor grado posible implica empeorar la frontera, entonces todos los nodos de menor grado a él también la empeoraran (lo cual, visto de otra forma, significa también que agregar ese nodo implica tomar la mejor decisión a corto plazo).

Por lo tanto, sabemos que estamos tomando siempre la mejor decisión rápida. Ahora bien, es importante marcar que, llegado el punto en el cual no conseguimos una manera de agrandar la clique mejorando la frontera, la mejor decisión es detenerse. Esto implica no solo que no podemos dar el próximo paso sin achicar la frontera, sino que a partir de este punto no se podrá mejorar la frontera en ningún paso. Esto es fácil de ver si consideramos lo que vimos antes: si el nodo de mayor grado posible empeora la frontera, todos los de grado menor a él también lo hacen. Supongamos que, efectivamente decidimos agregar un nodo $j$ a la clique, de manera que nuestra frontera disminuya, pero luego encontramos un nodo $i$ que podemos agregar a la clique de forma que la frontera crezca. Esto implicará lo siguiente:

$$\text{Para } C \text{ clique original, } C_i \text{ clique con nodo i, } C_{ij} \text{ clique con nodos i y j.}$$

$$Frontera(C_i) = Frontera(C) + d(i) - tamano(C) \times 2$$

$$Frontera(C_{ij}) = Frontera(C_i) + d(j) - (tamano(C) + 1) \times 2$$

$$\text{Vimos que } Frontera(C_{ij}) > Frontera(C_i), \text{ lo cual implica } d(j) > (tamano(C) + 1) \times 2), \text{ es decir,}$$
$$d(j) > tamano(C) \times 2$$
$$\text{Pero } Frontera(C_i) < Frontera(C), \text{ lo cual implica } d(i) < tamano(C) \times 2$$

Por lo tanto, tenemos $d(j) > tamano(C) \times 2 > d(i)$, es decir, $d(j) > d(i)$. Pero entonces, eso implica que al agregar el nodo $i$ no agregamos el de mayor grado, puesto que $j$ es adyacente a todos los nodos de $C$ y es de mayor grado que $i$.

Absurdo.

De este modo, tenemos una heurística constructiva golosa válida, que toma siempre la mejor decisión posible relacionada al próximo paso y se detiene cuando ya no hay forma de agrandar su frontera agregando nodos a la clique.

Considerando entonces este desarrollo, obtenemos el siguiente algoritmo:

```
constructivaGolosa (in listaAdyacencia : lista) → res: clique
  mayor ← nodoDeMayorGrado(lista)
  agregarNodoAClique(res, mayor])
  nodosAdyacentes ← adyacentes(lista, mayor)
  ordenarPorGrado(nodosAdyacentes)
  for i ← 0 to nodosAdyacentes.largo do
  |  if esAdyacenteATodos(nodosAdyacentes[i], res, listaAdyacencia) ∧
  |  aumentaLaFrontera(nodosAdyacentes[i], res, listaAdyacencia) then
  |  |  agregarNodoAClique(res, nodosAdyacentes[i])
  |  end if
  end for
```

## 3.3. Complejidad Temporal

Esta implementación es bastante sencilla y solo utiliza unas pocas funciones. Veamos la complejidad temporal en peor caso de ellas para después ver la del algoritmo general.

- **nodoDeMayorGrado**: Recorre los n nodos en la lista y se fija el largo de sus adyacentes en $O(1)$ por lo que la función es $O(n)$.
- **ordenarPorGrado**: Esta función utiliza por detrás el sort de la STD, y por ende su complejidad en peor caso es de $O(n * log(n))$.
- **esAdyacenteATodos**: Busca si hay un nodo en la clique que no sea adyacente a este nuevo nodo. Para eso, arma un arreglo de booleanos y recorre los nodos adyacentes al nodo a insertar (pueden ser hasta $n - 1$). Después, basta con recorrer los nodos de la clique y fijarse en el arreglo si son adyacentes o no. Siguiento este procedimiento, el algoritmo tiene una complejidad temporal $O(|adyacentes| + |clique|)$, donde $O(|adyacentes|) \subseteq O(n)$ y $O(|clique|) \subseteq O(n)$. Por lo tanto, tenemos $O(n + n) \subseteq O(n)$.
- **aumentaLaFrontera**: Hace una simple aritmética entre la cantidad de nodos y la cantidad de adyacentes de la clique por lo que su complejidad es $O(1)$.
- **agregarNodoAClique**: Se encarga de actualizar la clique agregando atrás del vector de nodos en la clique el nodo a insertar en $O(1)$.

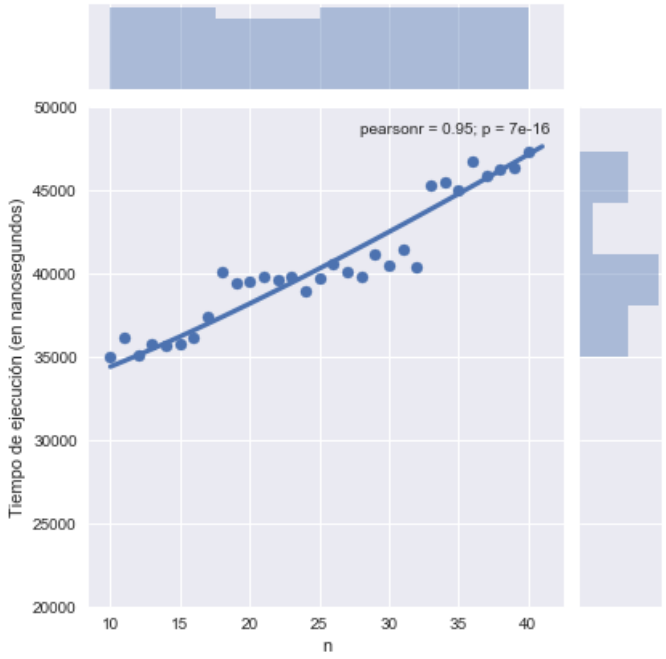
Ahora analicemos la complejidad de la heurística completa, inicialmente buscamos el nodo de mayor grado y ordenamos sus adyacentes, la operación más costosa aquí es la de ordenar que es $O(n * logn)$.

Nos queda ver la iteración sobre los nodos del grafo: para cada nodo del grafo vamos a llamar a esAdyacenteATodos, como vimos esta función tiene una complejidad de $O(|adyacentes| + |clique|)$ que dijimos que estaba acotado por $n$ por lo que una cota temporal del algoritmo burda sería $O(n^2)$, pero podemos ajustar más esta cota.

Cómo recorremos todos los nodos y para cada nodo iteramos entre sus adyacentes en esAdyacenteATodos, estamos recorriendo todos los ejes 2 veces($m$) en total, pero además en cada iteración recorremos la cantidad de nodos en la clique actual. Podemos acotar la cantidad de nodos en la clique por la cantidad de nodos de la clique máxima, por lo que podemos decir que el algoritmo tiene una complejidad temporal en peor caso de $O(n*log(n)+n*|cliqueMax|+m)$.

## 3.4. Experimentación

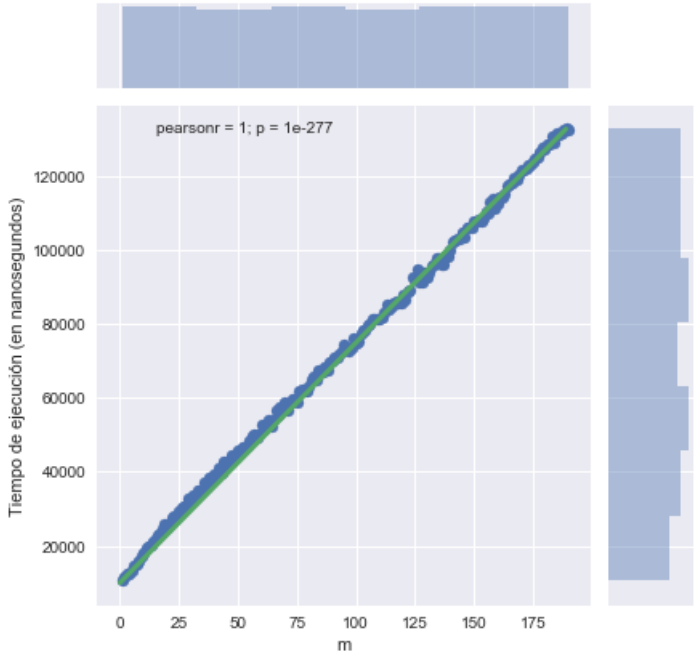
El mayor desafío al analizar nuestra complejidad fue visualizar el impacto de $n$ en el tiempo de ejecución.




Datos del gráfico

| | |
|---|---|
| Curva aproximada | $m = 40$<br>$f(x) = 100 * x * log(x)$<br>$+10 * x + 32000$ |

Nos resultó muy dificil medir el impacto del tamaño de la clique máxima, ya que la misma depende mucho de las aristas del grafo generado. Esto resultó en algunos resultados dispares. Sin embargo, podemos ver por el r de Pearson que la correlación se aproxima a lineal.




Datos del gráfico

| | |
|---|---|
| Curva aproximada | $n = 20$<br>$f(x) = 650 * x + 10000$ |

Por otro lado, el impacto lineal de la cantidad de aristas fue más que evidente, ya que el mismo coeficiente r dio exactamente 1 con margen de error ínfimo. La curva aproximada debio ser graficada en otro color por este motivo.

# 4. Heurística de búsqueda local

## 4.1. Introducción

Las heurísticas de búsqueda local son capaces de iterar sobre cualquier solución dada y mejorarla, buscando soluciones vecinas a la solución inicial.

Para plantear la búsqueda local tenemos que definir sobre qué soluciones vecinas buscará el algoritmo. Es decir, tenemos que definir una relación de vecindad entre el espacio de soluciones del problema. Para que el enfoque no sea complejo, definimos 4 vecindades simples:

1. Agregar un nodo a la clique, siempre y cuando siga manteniendo la propiedad de clique.
2. Eliminar un nodo perteneciente a la clique.
3. Intercambiar un nodo de la clique por uno que no pertenezca a ella.
4. Intercambiar un par de nodos de la clique por otro par que no pertenezca a ella.

Ya definidas las vecindades, lo que vamos a hacer en el algoritmo es buscar entre estas soluciones vecinas y quedarnos con la que más nos mejore la frontera, repitiendo este procedimiento hasta que las soluciones encontradas no mejoren.

## 4.2. Desarrollo

Si bien es cierto que la idea de realizar un LocalSearch puede ser aplicada de manera ajena a la solución constructiva golosa que vimos anteriormente, optamos por juntar ambas heurísticas de manera que cada una de las soluciones alternativas que se generen tengan un desarrollo constructivo goloso detrás. De este modo, lo que buscamos es darle a Local la responsabilidad de generar rutas alternativas, con nodos que descartamos en nuestra primera construcción, y a partir de ellas volver a encargarle a nuestro algoritmo constructivo que alcance una solución golosa.

De este modo, si repasamos las vecindades definidas anteriormente, notaremos que tanto agregar como eliminar nodos no son opciones que vayan a servirnos (por lo menos, hasta que apliquemos una nueva heurística sobre estas dos), puesto que Local Search recibirá siempre un algoritmo construido mediante la heurística golosa, donde todos los nodos agregados implican un crecimiento de la frontera y donde no hay ningun nodo externo que agrande la misma.

Por ende, dentro de las vecindades solo nos interesa intercambiar tanto uno como dos nodos de la clique, y a partir de esta nueva clique generar un resultado goloso. No es dificil notar que la vecindad, aún quitándole dos nodos y agregando dos nuevos, es pequeña: para casos con $n$ grande, el hecho de intercambiar solo dos nodos de una solución golosa dificilmente nos lleve a la CMF. Sin embargo, el hecho de correr el algoritmo hasta que no se consigan mejoras puede acercarnos bastante a esta solución, más si consideramos que detrás de la búsqueda local hay una heurística golosa que, para cada iteración, encuentra una solución constructiva aún mejor que la anterior.

Es importante ver que siempre será necesario devolver la solución alternativa luego de aplicarle el algoritmo goloso, puesto que de otro modo podríamos descartar una solución mejor a la actual de manera errónea. Si tomamos nuestra solución inicial, le quitamos dos nodos y le agregamos dos nuevos, si bien es posible que este nuevo grafo tenga una frontera mayor al anterior, también podría ocurrir que se genere una clique de peor solución parcial, pero que al aplicarle el algoritmo goloso nos lleve a una clique con una frontera mayor a la que teníamos. Esto pasaría porque conseguimos armar una clique adyacente a nodos que anteriormente no habíamos considerado, y que finalmente resultaron ser una mejor solución para nuestro problema.

Imaginémoslo con dos montañas, para acercarnos de manera intuitiva. Nosotros estamos parados actualmente en el pico de una montaña, pero al caminar dos pasos para abajo y dos hacia arriba a la derecha, encontramos un camino nuevo. Bien podríamos descartarlo, porque la altura en la que estamos parados actualmente es menor que la anterior, pero al subirlo podríamos acabar en un pico más alto que el anterior. Es por esto que, a la hora de obtener la solución, es mejor tener en cuenta cuál es el pico más alto que puede alcanzar la nueva clique.

Por lo tanto, acabaremos encontrando o bien nuevas soluciones, o bien la misma solución que encontramos en el algoritmo goloso inicial, de forma que siempre acabaremos teniendo un resultado mejor o igual al que se conseguía con la heurística constructiva aislada.

Teniendo esto en cuenta, quedarían definidas de la siguiente manera las vecindades y el resultado obtenido de cada una de ellas. Como las búsquedas de cada tipo de vecindad son diferentes entre sí, vamos a implementar una función que nos encuentre la mejor solución para cada tipo y después vamos a comparar estas soluciones. Esto nos trae el beneficio de que, en caso de agregar nuevas vecindades, no deberemos tocar mucho la implementación.

#### 4.2.1. Vecindad: Agregar Nodo

Esta vecindad es la que se aplica en la heurística golosa constructiva. Como ya vimos, la solución que nos va a mejorar más la frontera será agregar el nodo que mantenga la clique y tenga mayor grado.

```
localAgregar (in listaAdyacencia : lista, in/out clique : solucion)
  nodosAdyacentes ← adyacentes(lista, mayor)
  ordenarPorGrado(nodosAdyacentes)
  for i ← 0 to nodosAdyacentes.largo do
    if esAdyacenteATodos(nodosAdyacentes[i], solucion, listaAdyacencia) ∧
    aumentaLaFrontera(nodosAdyacentes[i], solucion, listaAdyacencia) then
      agregarNodoAClique(solucion, nodosAdyacentes[i])
      break
    end if
  end for
  solucion ← algoritmoGoloso(solucion, listaAdyacencia)
```

#### 4.2.2. Vecindad: Eliminar Nodo

Aquí queremos encontrar nodos que al eliminarlos nos aumenten la frontera. Estos nodos van a tener la característica de tener más adyacentes en la clique que afuera, ya que si por ejemplo el nodo $j$ perteneciente a la clique tiene 2 adyacencias por afuera de la misma y esta es de grado 4(por lo que tendrá 3 adyacentes pertenecientes a la clique) al eliminarlo, la frontera perderá los 2 ejes adyacentes a $j$, pero ganará los 3 ejes que llegan de la clique a $j$. Por lo tanto, podemos definir el beneficio de eliminar un nodo de la clique como $2 * (gradodelaclique - 1) - cantidaddeadyacentesalnodo$.

La implementación es bastante directa, recorremos los nodos de la clique y buscamos el que más nos aporte a la frontera si lo eliminamos.

```
localEliminar (in listaAdyacencia : lista, in/out clique : solucion)
  mejora ← 0
  for i ← 0 to solucion.nodos.largo do
    mejoraNodo ← (solucion.nodos.largo - 1) - adyacentesA(lista, solucion.nodos[i]).largo
    if mejoraNodo > mejora) then
      nodoABorrar ← i
      mejora ← mejoraNodo
    end if
  end for
  if mejora > 0) then
    borrarNodoClique(solucion, solucion.nodos[nodoABorrar])
  end if
  solucion ← algoritmoGoloso(solucion, listaAdyacencia)
```

#### 4.2.3. Vecindad: Intercambiar Nodo

Para encontrar el par de nodos(uno interno a la clique y otro externo) que mejoren la clique, lo que vamos a hacer es recorrer los nodos internos y, por cada nodo, buscar en los ejes externos cuanto nos mejora la solución si los

intercambiamos.

```
localIntercambiar (in listaAdyacencia : lista, in/out clique : solucion)
  mejora ← 0
  for i ← 0 to solucion.nodos.largo do
  |  for j ← 0 to solucion.nodosExternos.largo do
  |  |  mejoraNodo ← mejoraEnFronteraAlIntercambiar(solucion.nodosExternos[j], solucion.nodos[i], solucion,
  |  |   listaAdyacencia)
  |  |  if esAdyacenteATodosMenosA(solucion.nodosExternos[j], solucion.nodos[i], solucion, listaAdyacencia) ∧
  |  |  mejoraNodo ¿mejora then
  |  |  |  nodoABorrar ← i
  |  |  |  nodoAAgregar ← j
  |  |  |  mejora ← mejoraNodo
  |  |  end if
  |  end for
  end for
  if mejora > 0) then
  |  borrarNodoClique(solucion, nodoABorrar[nodoABorrar])
  |  agregarNodoClique(solucion, solucion.nodos[nodoAAgregar])
  end if
  solucion ← algoritmoGoloso(solucion, listaAdyacencia)
```

#### 4.2.4. Vecindad: Intercambiar par de nodos

La idea de intercambiar un par de nodos es similar a la explicada a la hora de intercambiar un único nodo, con la diferencia que a la hora de generar la solución final (es decir, a la hora de aplicarle el algoritmo goloso) deberá primero haberse hecho el intercambio de ambos nodos. Por lo tanto, la idea sería:

- Sacar dos nodos
- Meter dos nodos
- Algoritmo goloso sobre la nueva clique

#### 4.2.5. Algoritmo final

Finalmente, considerando estas cuatro vecindades, acabaremos teniendo un algoritmo general de este estilo:

```
heuristicaDeBusquedaLocal (in listaAdyacencia : lista, in/out clique : solucion)
  while esMejor do
  |  esMejor ← falso
  |  posibleMejorSolucion ← buscarMejorSolucionVecina(lista, solucion)
  |  if posibleMejorSolucion.fronter ¿solucion.frontera) then
  |  |  esMejor ← verdadero
  |  |  solucion ← posibleMejorSolucion
  |  end if
  end while
```

Donde buscarMejorSolucionVecina corre las cuatro vecindades y se fija cual es la mejor.

### 4.3. Complejidad Temporal

Para analizar la complejidad temporal de este ejercicio, debemos analizar la complejidad de cada una de los algoritmos que corremos para realizar la búsqueda en sus vecindades. Por lo tanto, tendremos las siguientes complejidades:

**Agregar nodo**

Este algoritmo recorre los nodos que no pertenecen a la clique (que son, como máximo, $n$) y, para cada uno de ellos, chequea si es o no adyacente a los nodos de la clique, y si agregarlo aumenta la frontera. Por lo tanto, para $n$ nodos, se fijará si el vértice es adyacente a la clique y realizará el cálculo correspondiente. Como vimos anteriormente, el algoritmo **esAdyacenteATodos** tiene una cota temporal de $O(n)$, mientras que el cálculo correspondiente a la frontera se realiza en $O(1)$. Por ende, acabamos teniendo n veces que realizar una operación de complejidad $O(n)$ (es decir, complejidad

de $O(n^2)$). A esta solución obtenida le aplicamos finalmente el **Algoritmo Goloso**, cuya complejidad vimos que es $O(n * log(n) + n * |cliqueMax| + m)$, donde $O(n * log(n))$, $O(n * |cliqueMax|)$ y $O(m)$ se pueden acotar por $O(n^2)$, por lo que terminaríamos teniendo $O(n^2 + n^2)$. Por lo tanto, acotando por última vez esta complejidad, concluímos que la complejidad de Agregar Nodo es $O(n^2)$.

### Eliminar nodo

Este algoritmo recorre los nodos de la clique (máximo $n$ nodos) y para cada uno de ellos se fija si, al quitar este nodo, la frontera de la clique mejora. Como nuevamente, lo único que tenemos que hacer es realizar el cálculo de la frontera en $O(1)$, la complejidad de este ciclo será $O(n)$. Finalmente, como vimos, le corremos el **Algoritmo Goloso** a la solución, lo cual tiene una complejidad $O(n * log(n) + n * |cliqueMax| + m)$ pero que, como vimos, se puede acotar por $O(n^2)$. Por ende, como sabemos que esta complejidad se sumará con la de Agregar Nodo, y el resto de la complejidad de Eliminar Nodo es despreciable frente al Algoritmo Goloso, definimos que la cota temporal de Eliminar Nodo es $O(n^2)$.

### Intercambiar nodos

Si bien tenemos dos tipos de intercambios, de la manera en que definimos cada uno de ellos podemos asegurarnos que la cota temporal de intercambiar un solo nodo es menor a la de intercambiar dos. Por lo tanto, como a la larga sus complejidades se sumarán y una de ellas será despreciable, vamos a centrarnos en la complejidad de intercambiar dos nodos internos por dos externos.

Este algoritmo tomará la clique original deberá sacar dos nodos de ella (donde la cantidad máxima es $n$) e ingresar dos nuevos. Es facil ver que extraer cada par de nodos es igual a separarlos y realizar la siguiente operación:

**Para cada nodo $i \in$ clique, para cada nodo $j \in$ clique, j $\neq$ i, extraer j e i.**

Esto nos deja con una complejidad de $O(n^2)$, a la cual se le debe sumar la de agregar los nodos externos de la clique. De nuevo, tendremos la misma operación, pero esta vez por afuera de la clique, es decir:

**Para cada nodo $i$ fuera de clique, para cada nodo $j$ fuera de clique, j $\neq$ i, agregar j e i.**

Por lo tanto, como debemos tanto agregar como sacar nodos, acabaremos teniendo $O(n^2 * n^2)$, lo cual acaba siendo $O(n^4)$, el hecho de tan solo recorrer todas las cliques que podríamos generar. A su vez, para cada una de estas posibles cliques debemos asegurarnos si los dos nodos agregados son adyacentes a todos los demas, y como vimos **esAdyacenteATodos** tiene una cota temporal de $O(n)$, y al realizarlo dos veces (una por cada nodo del par) tendremos $O(2n) \subseteq O(n)$. Por lo demás, son cálculos realizables en $O(1)$ que no afectan nuestra complejidad, por lo que acabaremos teniendo, para cada posible clique, una complejidad de $O(n)$, lo cual acaba siendo al considerar todas las complejidades $O(n^5)$.

Así, alcanzamos las complejidades de cada una de nuestras vecindades. Por lo tanto, la complejidad de buscar la mejor solución vecina será $O(n^2+n^2+n^5)$, que por propiedades de O grande se acota finalmente con $O(n^5)$. Puesto que el algoritmo general es, simplemente, tomar una clique y correrle el algoritmo hasta que no mejore, el mismo correrá un máximo de $n$ veces (o bien agrega $n$ nodos, o bien los saca). Por lo tanto, al considerar el peor caso, habremos corrido $n$ veces un algoritmo de costo $O(n^5)$, dejándonos con una complejidad temporal final de $O(n^6)$

# 5. Metaheurística GRASP

## 5.1. Desarrollo

Hemos propuesto dos métodos que computan una solución, formulándola en base a criterios heurísticos. La limitación de los enfoques anteriores reside en que se recorre el espacio de soluciones hasta que no es posible mejorar la solución. Como sabemos que una solución maximal no necesariamente es máxima, sería útil poder contrastar distintas soluciones maximales y elegir a la mejor. En otras palabras quisiéramos ramificar la exploración del espacio de soluciones.

Habiendo notado que heurísticas determinísticas siempre toman la misma decisión en el mismo paso, se propone la utilización de una heurística pseudo-greedy. Esta heurística aleatoriamente tomará decisiones localmente buenas pero no necesariamente óptimas. Se considerará que la decisión de agregar un nodo es buena si agregarlo aumenta el tamaño de la frontera. Para tener cierto control sobre que tan goloso es el comportamiento de esta heurística se puede pedir que la elección aleatoria sea tomada teniendo en cuenta solo el mejor porcentaje de las decisiones posibles. Si el porcentaje es chico, solo se consideraran las mejores opciones y el comportamiento será muy similar a la heurística greedy pura. Esto restringiría la ramificación que buscabamos. Si el porcentaje es muy alto existirá la posibilidad de agregar un nodo de grado muy bajo a la clique, lo cual restringirá en gran medida la cantidad de nodos que se pueden agregar, obteniendo muy posiblemente una clique pequeña.

```
randomGreedy (in listaAdyacencia : lista, in float : porcentajeConsiderado) → res: clique
  nodosConsiderados ← nodos(lista)
  ordenarPorGrado(nodosConsiderados)
  indiceNodoAleatorio ← nodoAleatorio(nodosConsiderados, porcentajeConsiderado)
  nodoPorAgregar ← nodosConsiderados[indiceNodoAleatorio]
  agregarNodoAClique(res, nodoPorAgregar)
  nodosConsiderados ← nodoPorAgregar.adyacentes()
  ordenarPorGrado(nodosConsiderados)
  res ← recurRandomGreedy(lista, res, nodosConsiderados, porcentajeConsiderado)
```

```
recurRandomGreedy (in listaAdyacencia : lista, in clique : cliqueParcial, in listaNodos :
nodosConsiderados in float : porcentajeConsiderado) → res: clique
  if nodosConsiderados.size() = 0 then
  |  return cliqueParcial
  end if
  for nodo ∈ nodosConsiderados do
  |  if nodo.grado() < cliqueParcial.size() * 2 ∨ nodo.esAdyacenteATodos(clique, lista) then
  |  |  nodosConsiderados.borrar(nodo)
  |  end if
  end for
  if nodosConsiderados.size() = 0 then
  |  return cliqueParcial
  end if
  indiceNodoAleatorio ← nodoAleatorio(nodosConsiderados, porcentajeConsiderado)
  nodoPorAgregar ← nodosConsiderados[indiceNodoAleatorio]
  agregarNodoAClique(res, nodoPorAgregar)
  nodosConsiderados.borrar(nodoPorAgregar)
  res ← recurRandomGreedy(lista, cliqueParcial, nodosConsiderados, porcentajeConsiderado)
```

```
nodoAleatorio (in listaNodos : nodosConsiderados in float : porcentajeConsiderado) → res: clique
  cantidadPorConsiderar ← nodosConsiderados.size() ∗ (1 - porcentajeConsiderado)
  res ← random(rango(cantidadPorConsiderar))
```

Cada vez que se elige un nodo, se hace eligiendo un nodo aleatorio que esté entre los mejores de la lista de nodos agregables. En un principio, todos los nodos son elegibles para formar una clique trivial de tamaño uno. El criterio utilizado para elegir alguno es la priorización de nodos de grado alto. Es por esto que en primer lugar se ordenan los nodos en base a su grado. Después se elige un nodo aleatorio entre los de mayor grado. El porcentaje a considerar es una variable de entrada que determinará el comportamiento de la heurística.

La función recursiva tiene un procesamiento muy similar, pero toma como parámetro a una clique y una lista de nodos a considerar. En el caso base, si no quedan nodos por considerar, la clique es maximal. Sino, toma la lista y le

filtra los nodos que no son adyacentes a todos los nodos de la clique o que no agrandarían la frontera por tener un grado muy chico. Vuelve a preguntar si quedan nodos a considerar y en caso afirmativo elige un nodo aleatorio entre los mejores y lo agrega a la clique. Elimina a ese nodo de la lista de nodos a considerar y se llama recursivamente. Como en cada paso la cantidad de nodos a considerar disminuye al menos en una unidad, sabemos que la función eventualmente llega al caso base.

La metaheurística GRASP utiliza tanto búsqueda local como greedy aleatorio. La idea esta en que greedy aleatorio avanza estocásticamente por el espacio de soluciones hasta que llega a una solución maximal. Posteriormente esta solución se pasa como parametro a la búsqueda local. Si hacemos esto muchas veces tenemos la posibilidad de llegar a muchas soluciones diferentes y así quedarnos con la mejor. Se memoriza la mejor encontrada y en cada iteración del ciclo se compara una nueva solución. Si iteramos lo suficiente, tendremos seguridad de que la solución que guardamos es la mejor entre muchas posibilidades.

```
grasp (in listaAdyacencia : lista, in unsignedint : iteraciones, in float : porcentajeConsiderado) → res: clique
    bestClique ← ∅
    for i ∈ rango(iteraciones) do
        tempClique ← local(randomGreedy(lista, porcentajeConsiderado))
        if bestClique.frontera() < tempClique.frontera() then
            bestClique ← tempClique
        end if
    end for
    res ← bestClique
```

## 5.2. Cota temporal

La complejidad de randomGreedy está dada por:

- •xtbfnodos(lista): Devuelve una lista que contiene a todos los nodos del grafo en $O(n)$.
- **ordenarPorGrado**: Esta función utiliza por detrás el sort de la STD, y por ende su complejidad en peor caso es de $O(n * log(n))$.
- **indiceNodoAleatorio**: Devuelve un número aleatorio en $O(1)$.
- **agregarNodoAClique**: Se encarga de actualizar la clique agregando atrás del vector de nodos en la clique el nodo a insertar en $O(1)$.
- **agregarNodoAClique**: Se encarga de actualizar la clique agregando atrás del vector de nodos en la clique el nodo a insertar en $O(1)$.

Vemos que el costo de esta función es dependerá del costo de la función recursiva:

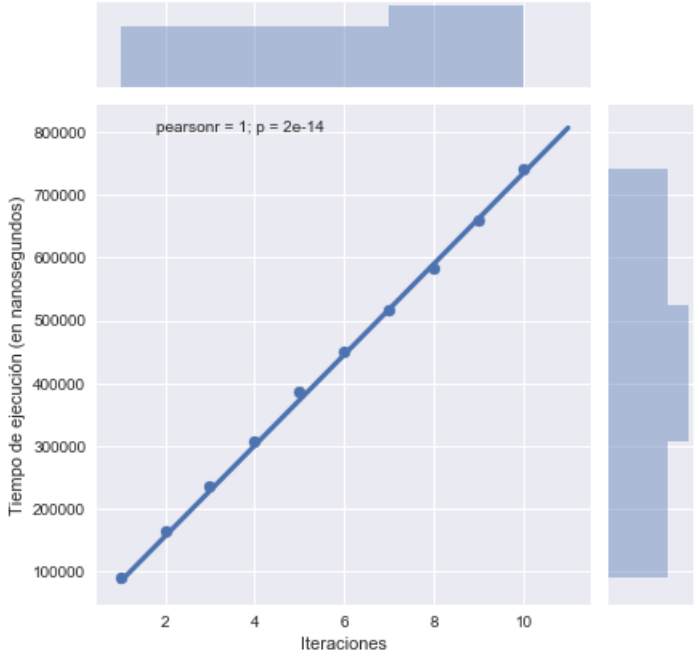
- **esAdyacenteATodos**: Busca si hay un nodo en la clique que no sea adyacente a este nuevo nodo. Para eso, arma un arreglo de booleanos y recorre los nodos adyacentes al nodo a insertar (pueden ser hasta $n - 1$). Después, basta con recorrer los nodos de la clique y fijarse en el arreglo si son adyacentes o no. Siguiento este procedimiento, el algoritmo tiene una complejidad temporal $O(|adyacentes| + |clique|)$, donde $O(|adyacentes|) \subseteq O(n)$ y $O(|clique|) \subseteq O(n)$. Por lo tanto, tenemos $O(n + n) \subseteq O(n)$.
- **agregarNodoAClique**: Se encarga de actualizar la clique agregando atrás del vector de nodos en la clique el nodo a insertar en $O(1)$.

Dado que la función recursiva termina cuando el parámetro nodosPorConsiderar es de tamaño cero y que en el peor de los casos puede empezar siendo de tamaño $O(n)$ y decrecer en una unidad en cada llamada recursiva, se concluye que en el peor de los casos se realizarán $O(n)$ llamadas recursivas. En cada una hay un ciclo de $O(nodosPorConsiderar)$ iteraciones y adentro se llama a la funcion esAdyacenteATodos que es $O(n)$ como en el peor de los casos nodosPorConsiderar decrece de a una unidad, la complejidad será $O(n * \Sigma in) = O(n^2 * \Sigma i) = O(n^3)$.

Dado que GRASP esta compuesto por un ciclo que corre tantas veces como se le especifique en el parámetro iteraciones, el costo temporal va a depender linealmente del número de iteraciones. En cada iteración del ciclo se realiza una llamada a localSearch(randomGreedy()) lo cual cuesta $O(n^3 + n^6) = O(n^6)$. Por lo tanto el costo de la metaheuristica GRASP será $O(iteraciones * n^6)$

## 5.3. Experimentación

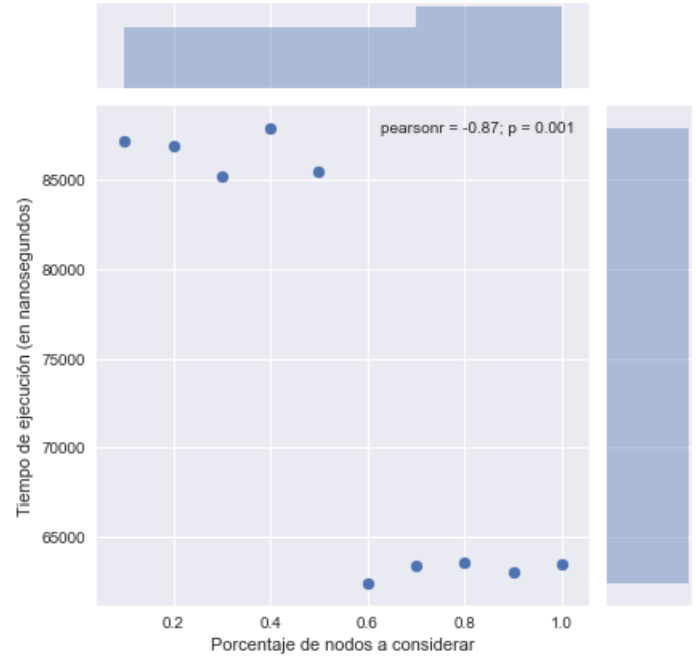
Al experimentar con la metaheurística, primero decidimos analizar el impacto lineal de las iteraciones:




Datos del gráfico

| | |
|---|---|
| | $n = 10$ |
| | $m = 20$ |
| Porcentaje de nodos considerados | $p = 0{,}5$ |
| Curva aproximada | $f(x) = 72500 * x * 10000$ |

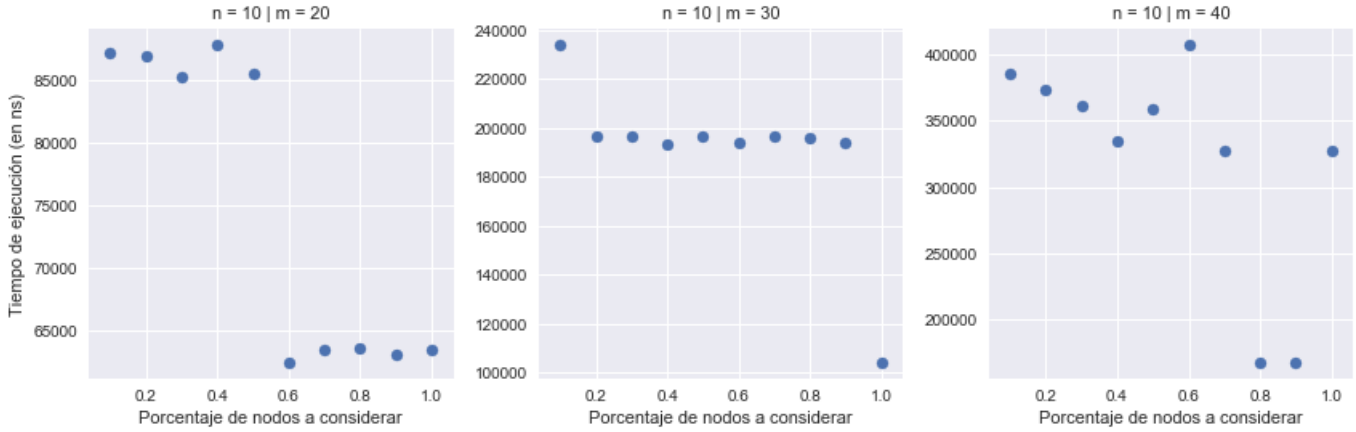
Este resultado era más que esperado, ya que resulta de ejecutar las mismas operaciones una cantidad fija de veces. Teniendo esto en cuenta, podemos analizar los demás factores en el caso de una única iteración.




Datos del gráfico

| | |
|---|---|
| | $n = 10$ |
| | $m = 20$ |
| Iteraciones | $it = 1$ |

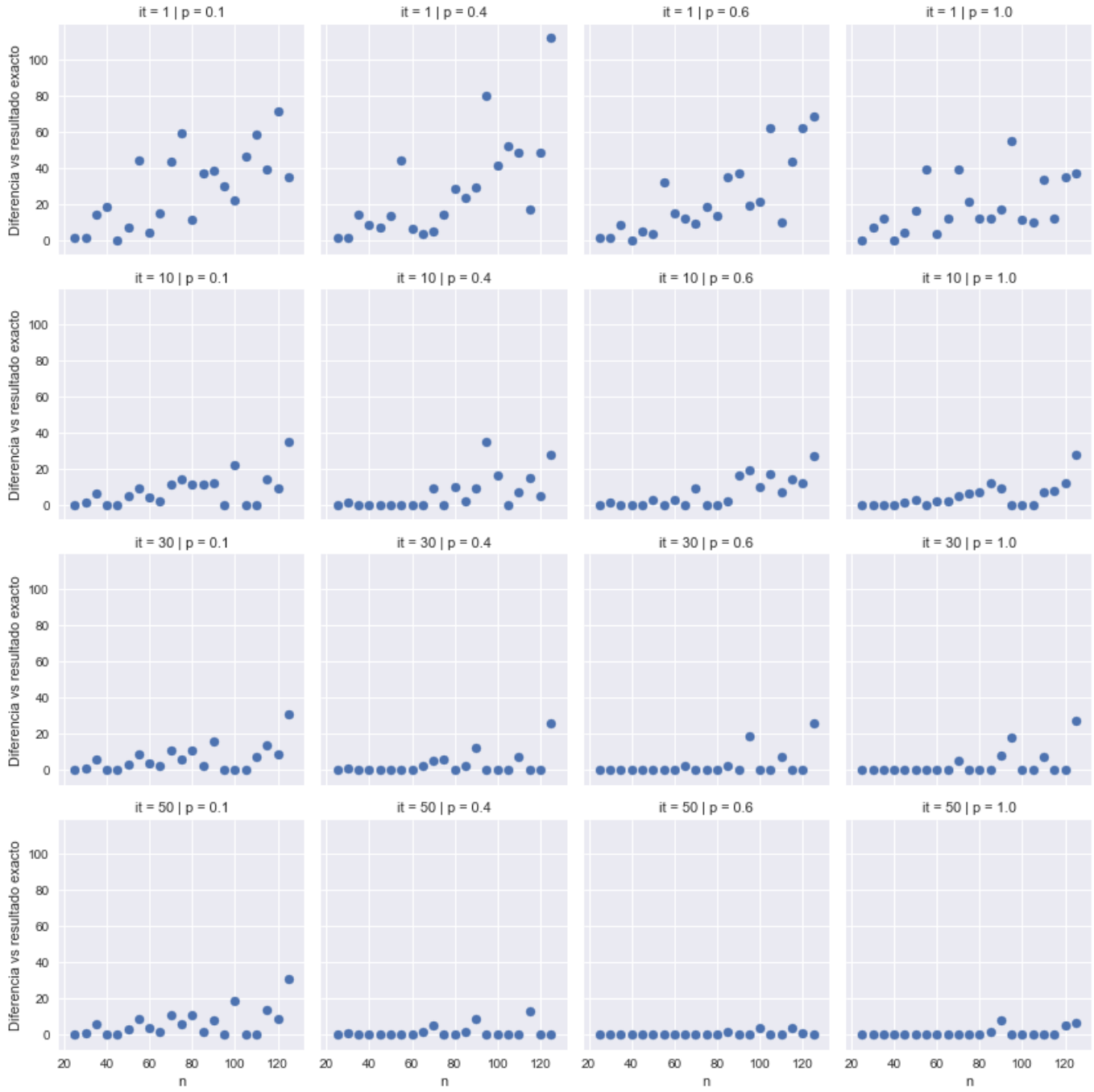
Por el otro lado, el porcentaje de nodos del grafo a considerar se comporta de manera muy peculiar con respecto a la complejidad. Procedimos a analizar más casos para comprender este comportamiento:




La variación en performance se debe en particular a 2 motivos. Por un lado, las búsquedas golosas aleatorias generan una cierta variabilidad en los resultados obtenidos, por lo que resulta difícil determinar qué corresponde a ruido y qué a un camino de decisiones distinto. Por el otro, a medida que aumentamos el porcentaje de los nodos a considerar, permitimos que ciertas instancias "menos óptimas" (desde una perspectiva golosa) sean utilizadas, lo cual puede llevar a decisiones no tan útiles y, por ende, cliques más pequeñas. Esto aplica en particular a los grafos con menor cantidad de aristas, ya que mientras más tenga, mayor será en general el grado los nodos, y por consiguiente siempre se considerará agregar más nodos a la clique final.

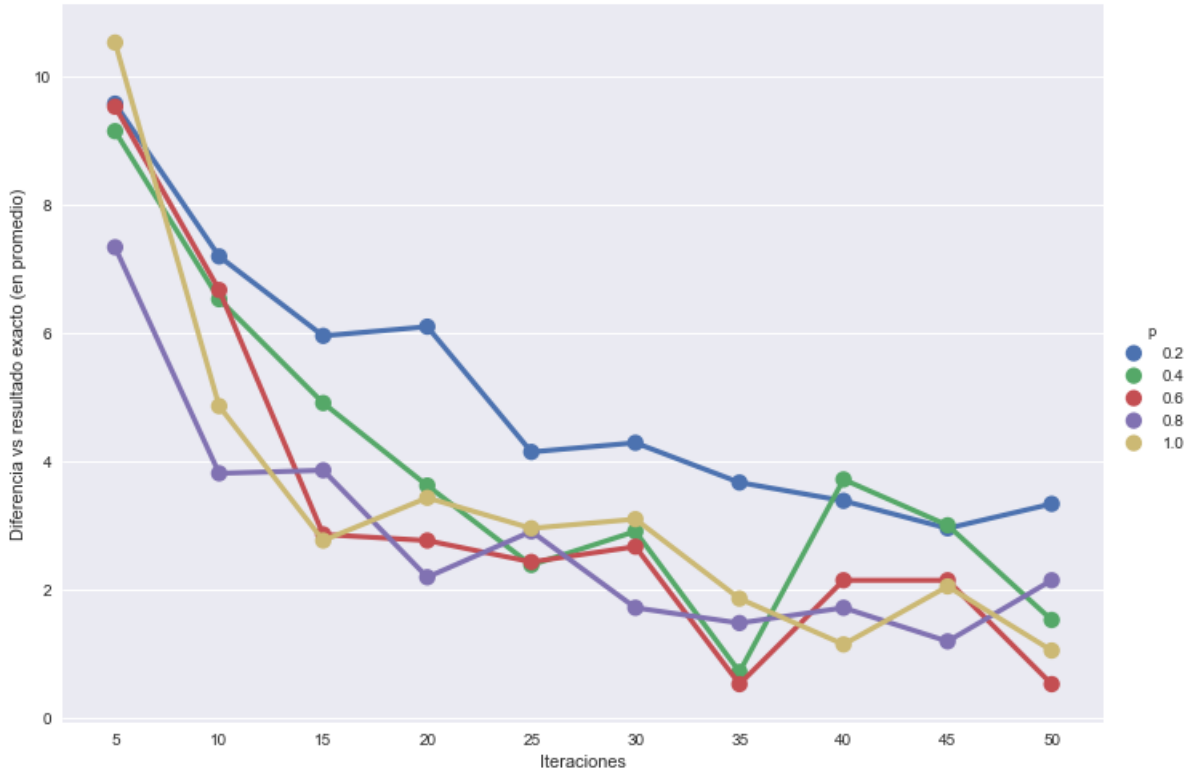
Además de experimentar con los tiempos de ejecución, decidimos analizar la influencia de ambos parámetros de entrada de la metaheurística en la precisión de los resultados obtenidos. Para esto, utilizamos un conjunto de tests de tamaño moderado, cuyas soluciones fueron obtenidas a través del algoritmo exacto. Para cada caso de prueba, probamos distintos valores para dichos parámetros y comparamos el resultado con el obtenido por fuerza bruta.

A fines prácticos, llamaremos $it$ al número de iteraciones realizadas, y $p$ al porcentaje de nodos a considerar (entre 0 y 1).




Este set de resultados nos resultó muy interesante. Pudimos comprobar que, casi siempre, realizar más iteraciones del algoritmo resulta en un resultado mejor (cosa que nos resultaba trivial, aunque podría no haber mejorado mucho). Por otro lado, también confirmamos que, por si solo, considerar más nodos no mejora de manera muy significativa los resultados. Sin embargo, nos tomó por sorpresa que, al realizar múltiples iteraciones ($it \geq 30$), considerar más nodos ($p \geq 0{,}6$) aparenta también mejorar bastante los resultados, aunque no en todas las situaciones.

Se debe tener en cuenta que estos resultados no son 100 % determinísticos, ya que los nodos escogidos varían de acuerdo al valor de p. Sin embargo, en base a este gráfico podemos afirmar que (para los casos de prueba utilizados) realizar aproximadamente 50 iteraciones y considerando el 60 % de los nodos de mayor grado, nuestra implementación de la metaheurística GRASP es casi tan precisa como una búsqueda por fuerza bruta.




Al ver más de cerca los datos, promediando las diferencias de todos nuestros casos de prueba, podemos observar la variabilidad de nuestro algoritmo: en promedio, nuestras pruebas dieron resultados igual de precisos con $p = 0{,}6$ realizando 35 o 50 iteraciones. Es más, para ningún valor de p obtuvimos una linea estrictamente decreciente, que sería lo esperable de algorítmos determinísticos.

También podemos ver que en varias ocasiones, para distintas cantidades de iteraciones, otros valores de p dieron resultados más precisos. Sin embargo, consideramos que aumentar la cantidad de iteraciones reduce la variabilidad introducida por las búsquedas golosas aleatorias (porque al iterar siempre conservamos el mejor resultado), por lo que la mayor cantidad de iteraciones es más representativa del valor óptimo de p.

# 6. Apéndices

## 6.1. Apéndice I: generación y análisis de datos

Para poder analizar las complejidades de los algoritmos propuestos, se utilizaron las siguientes herramientas:

- Un generador de grafos aleatorios con `std::random` (parte de la biblioteca estándar de C++11).

  Para poder probar las distintas implementaciones, creamos un generador de grafos aleatorios. El mismo consiste básicamente en un generador de grafos completos y una función de mezclado aleatorio, basado en los generadores de distribuciones uniformes provistas por la `std` de C++.

  Cada uno de los grafos generados se utilizó multiples veces para testear varios aspectos de un mismo algoritmo.

- Mediciones de tiempo con `std::chrono` (parte de la biblioteca estándar de C++11).

  Cada algoritmo fue probado 100 veces consecutivas con cada entrada, conservando solo el valor de tiempo menor para reducir el ruido por procesos ajenos al problema.

  La unidad de medición preferida fue nanosegundos (`std::chrono::nanoseconds`, $seg \times 10^{-9}$).

- Graficado con `matplotlib.pyplot`, `pandas` y `seaborn` (con Python y Jupyter Notebook)

  Se utilizaron los DataFrames de Pandas para el manejo de datos (guardados en `.csv`) y las funciones de regresión de Seaborn para el graficado, en conjunto con matplotlib.

  Algunos gráficos incluyen el coeficiente de correlación (o "r") de Pearson, al igual que un p-valor para la hipotesis nula que los valores pueden haber sido generados sin correlación real. Esto quiere decir, un R cercano a 1 con un p-valor mínimo indica que hay una fuerte correlación positiva entre los valores graficados; un p-valor elevado, por otro lado, invalida una posible correlación positiva o negativa.

## 6.2. Apéndice II: herramientas de compilación y testing

Durante el desarrollo se utilizaron las siguientes herramientas:

- CMake

  Se decidió utilizar CMake para la compilación por su simplicidad y compatibilidad con otras herramientras. Junto con el código se provee el archivo `CMakeLists.txt` para compilar el mismo.

- Google Test

  Para generar tests unitarios con datos reutilizables se usó Google Test. Dichos archivos eran importados por otro `CMakeLists.txt` y no están incluídos en la presente entrega del trabajo práctico.

- Namespace `Utils`

  Dentro de `Utils.h` se definieron algunas funciones ajenas a los algoritmos en sí e independientes de implementación. Entre ellas se encuentran fuciones de parseo de input, así como una función de logging que fue utilizada al programar para detectar errores y ver otros detalles del proceso.

  La función `log` sigue estando incluída en los algoritmos, pero su funcionalidad se encuentra apagada por un `#define` y no debería generar ningún costo adicional (ya que usa printf por detrás y no computa el output salvo que sea necesario).