DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 3

16 de julio 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.

- En un proyecto en conjunto llevado a cabo por alumnos de Biología y Computación, se quiere desarrollar un robot que tenga un comportamiento similar al de las arañas. Para esto, los biólogos realizaron una investigación y llegaron a la conclusión de que los arácnidos, a la hora de situarse en sus telarañas, buscan cumplir dos requisitos: mantenerse seguras, y alcanzar sus presas con total rapidez. De este modo, a los computólogos se les ocurrió imaginar la telaraña como un grafo, donde cada hilo será una arista y cada intersección un vértice. De este modo, como son los hilos los que protegen a la araña, un lugar seguro será aquel donde todos los vértices tengan un hilo entre sí, y un sitio alcanzable será todo vértice que se encuentre relacionado a un hilo que llegue hacia donde esta ella. Nuestra araña, de manera instintiva, no permitirá que ninguna presa se acerce adonde esta situada, por lo que solo consideraremos los sitios alcanzables por afuera de su lugar seguro. Como los alumnos concluyeron que sería más divertido tener una araña violenta, optaron por configurarla de manera que priorice el alcance a las presas antes que la seguridad, de modo que busque la posición segura donde más sitios alcanzables haya.

- Una mina contiene puntos en los que se pueden poner dinamitas o detonadores de dinamita, entre los puntos hay cables, pero puede ser que haya puntos que no estén conectados entre sí. Los detonadores solamente pueden activar las dinamitas que tengan un cable al detonador y además para que los detonadores se sincronicen también tienen estar conectados entre sí. Una empresa minera quiere explotar la mayor cantidad de dinamitas en una sola carga, osea que la explosión sea sincronizada, ya que así el ruido que provoque la explosión no se prolongara demasiado. Para encontrar una buena disposición de detonadores y dinamitas los científicos de la empresa aprovecharon que ya tenían los algoritmos de CMF y representaron el problema con grafos, cada punto es un nodo y si tienen un cable entre sí quiere decir que son adyacentes, se busca encontrar una clique que son los detonadores que tienen que estar conectados por la sincronización que tenga la frontera máxima, donde la frontera serán las dinamitas que puedan explotar la clique.

- Un poco reconocido psicólogo inventó un nuevo tipo de terapia, la terapia de amigos, que consiste en sesiones con el psicólogo y un grupo en el que son todos amigos entre sí. El psicólogo quiere difundir su nueva terapia, por lo que va a ofrecer gratis una sesión a un grupo de amigos para que estos lo difundan entre sus otros amigos fuera del grupo. Para ello el psicólogo le va a pagar a una red social para que le provea esta información, por suerte la red social tiene a las personas representadas como un nodo del grafo y los ejes como las amistades, por lo que la empresa buscará usar el algoritmo de CMF para buscar la clique, el grupo de amigos, que tenga la mayor cantidad de amistades fuera del grupo, la frontera.

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 $O(2^n)$.

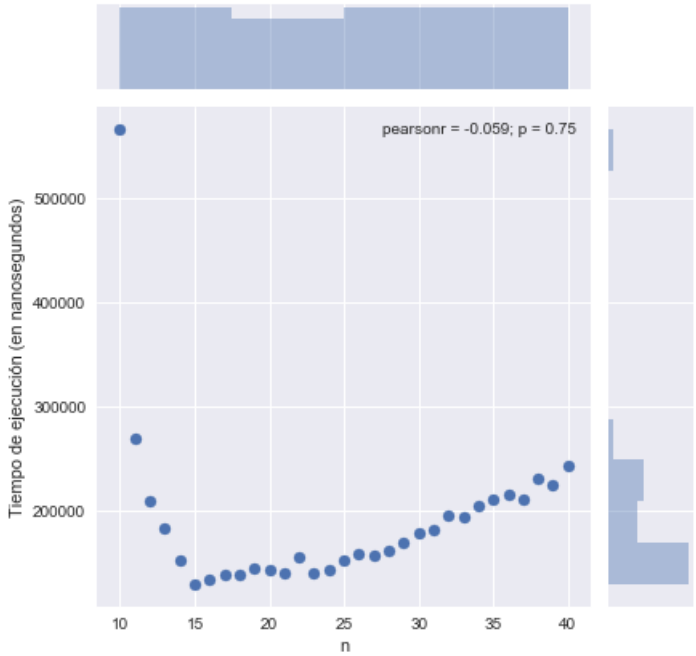
Encontrar CMF

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:

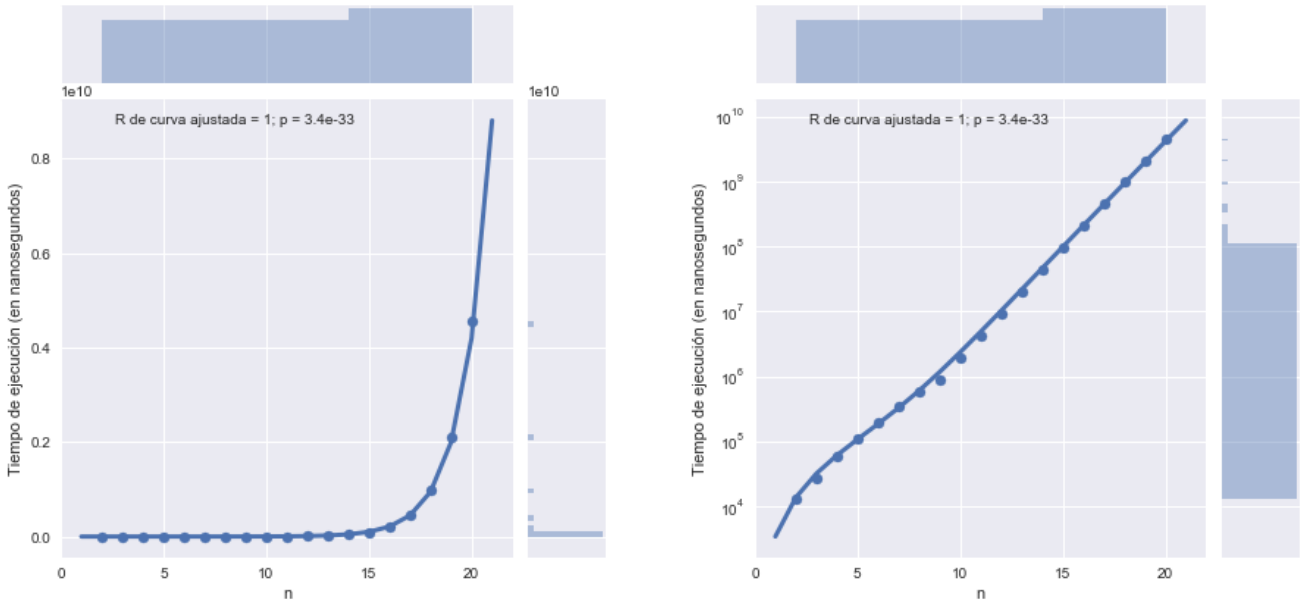


Datos del gráfico

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

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

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 $O(n * log(n))$. Como sabemos que no recorremos los n, sino $|maxAdyacente|$, podríamos acotarlo por $O(|maxAdyacente| * log(|maxAdyacente|))$
- **esAdyacenteATodos**: Busca si hay un nodo en la clique que no sea adyacente a este nuevo nodo. Para eso, arma un arreglo de booleanos del tamaño del grafo ($O(n)$) 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. Siguiendo este procedimiento, el algoritmo tiene una complejidad temporal $O(n + |adyacentes| + |clique|)$. Es facil ver que $O(|maxClique|) \subseteq O(|maxAdyacentes|)$, por lo que, si tomamos estas cotas, tendremos $O(n + |adyacentes| + |clique|) \subseteq O(n + |maxAdyacentes| + |maxClique|) \subseteq O(n + |maxAdyacentes| + |maxAdyacentes|)$. Veamos, entonces, que $|maxAdyacentes|$ esta acotado por $n - 1$ (puesto que un nodo puede llegar como máximo a todos los nodos del grafo que no son él) por lo que tenemos $O(|maxAdyacentes|) \subseteq O(n)$. Por ende, podemos afirmar $O(n + |maxAdyacentes| + |maxAdyacentes|) \subseteq O(n)$, y nuestra cota queda definida por el costo de armar el vector de booleanos.
- **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. Aquí, la operación más costosa es la de ordenar en un tiempo $O(|maxAdyacente| * log(|maxAdyacente|))$. Luego, tomamos el nodo de mayor grado y, para cada uno de sus nodos adyacentes, fijarnos si es posible agregarlo a la clique (es decir, si esAdyacenteATodos). Por lo tanto, para $|maxAdyacente|$ corremos un algoritmo de costo $O(n)$.

Sin embargo, así como vimos que $|maxAdyacente| \leq$ (n - 1), es fácil ver que en un grafo de 40 nodos y 5 aristas, $|maxAdyacente| \leq 5$ (puesto que un nodo jamás tendrá 6 o más adyacencias, ya que eso implicaría tener 6 o más aristas en el grafo). Por ende, si extendemos esto a todos los grafos, vemos que $O(|maxAdyacente|) \subseteq O(min(n, m))$, por lo que acabamos teniendo $O(min(n, m)) \times O(n)$.

Por ende, al implementar la misma cota sobre ordenarPorGrado, tenemos las siguientes operaciones a considerar: se encuentra el nodo de mayor grado en $O(n)$, se ordena por grado en $O(min(n, m) \times log(min(n, m)))$, y se arma la clique en $O(min(n, m) \times n)$. Entonces, nuestra cota temporal acaba siendo $O(n + min(n, m) \times log(min(n, m)) + min(n, m) \times n)$. Simplemente, vemos que $O(n) \subseteq O(min(n, m) \times n)$, y como $O(min(n, m)) \subseteq O(n)$ (porque siempre n será mayor o igual al mínimo entre n y m) , tenemos que $O(min(n, m) \times log(min(n, m))) \subseteq O(min(n, m) \times n)$. Por lo tanto, nuestra cota acaba siendo $O(min(n, m) \times n)$.

3.4. Casos Patológicos

La heurística falla en el resultado máximo, cuando uno de los nodos de mayor grado que siguen formando la clique no pertenece a la del resultado esperado, si esto sucede, otros nodos de menor grado nos permiten conseguir en conjunto una mayor frontera.

En la sección de Análisis de precisión de heurísticas analizaremos más a fondo estos casos, y los compararemos con las otras heurísticas a presentar.

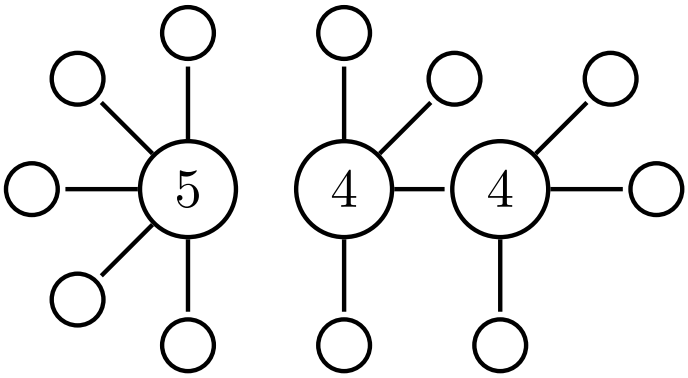


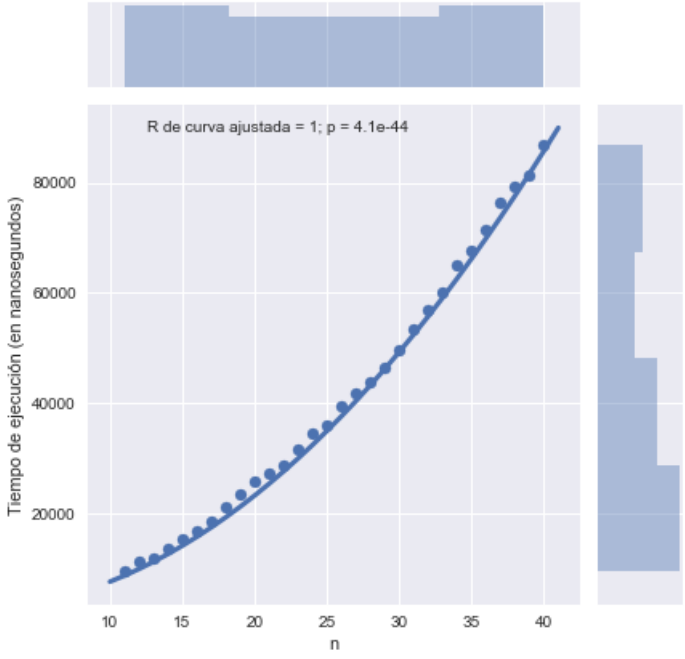
Figura 1: En este caso podemos apreciar que si elegimos el nodo de máximo grado(5), esta va a ser nuestra frontera maximal. Pero en cambio la clique formada por los nodos de menor grado(4) en conjunto su frontera es mayor, obteniendo una frontera de 6.

Podemos generar este tipo de grafos, teniendo una diferencia en el resultado máximo tan grande como queramos. Podemos definir una familia de instancias, que el nodo de mayor grado, digamos de grado k, tal que sus adyacentes son solo adyacentes a él (formando una frontera k), a este mismo grafo pertenece una clique de $\frac{k}{2}$ nodos y cada uno de estos además es adyacente a $\frac{k}{2}$ de grado 1. Los nodos de esta clique tienen grado $k-1$ por lo que en la golosidad hubiésemos preferido el de grado k, pero la clique paralelamente construída tiene $\frac{k}{2}$ nodos con $\frac{k}{2}$ adyacentes no pertenecientes a la clique, por lo que la frontera es de $\frac{k}{2} \times \frac{k}{2} = \frac{k^2}{4}$.

Notemos que la diferencia de frontera es de $\frac{k^2}{4} - k$ por lo basta con tomar un k mayor para que la diferencia sea tan grande como queramos, por lo que los grafos con esta particularidad la heurística arrojará malos resultados y estimaciones.

3.5. Experimentación

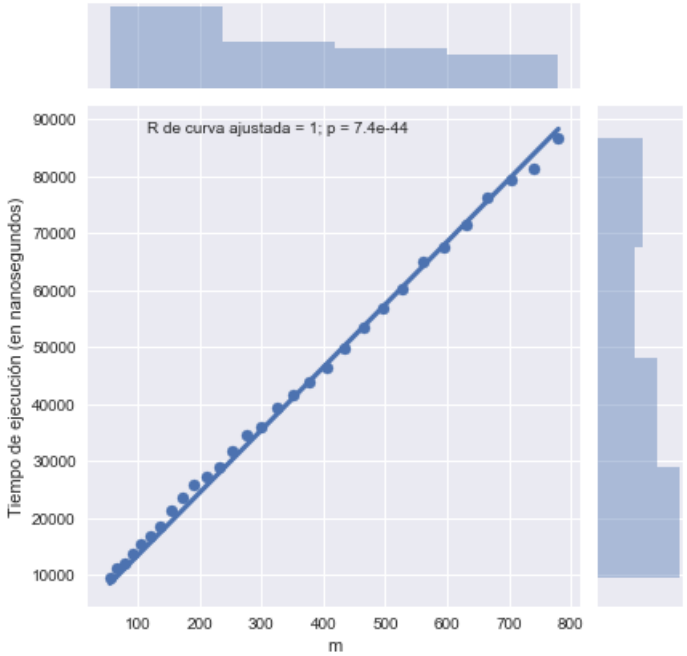
El desafío que nos encontramos al analizar la complejidad de nuestras heurísticas fue que las mismas dependían no solo de n y m, sino de la relación entre las mismas (el grado máximo de los nodos). Decidimos estudiar el peor caso en términos de tiempos de ejecución, que es el grafo completo. Por otro lado, consideramos analizar algunos casos promedio, pero el tiempo de ejecución depende de muchos aspectos (grado máximo, tamaño de la clique, etc.), por lo que resultó demasiado difícil medir y analisar estos casos.



Datos del gráfico

Grafo completo	$m = \frac{n*(n-1)}{2}$
Curva aproximada	$f(x) = 52 * x^2 + 2500$

Como se puede observar, nuestra cota teórica de $O(min(n, m) \times n)$ aplica correctamente para nuestro peor caso. En este contexto, n siempre es menor a m, por lo que $O(min(n, m) \times n) = O(n^2) = O(m)$. Podemos analizar esto mismo graficando los mismos datos en funcion de m:



Datos del gráfico

Grafo completo	$m = \frac{n*(n-1)}{2}$
Curva aproximada	$f(x) = 110 * x + 2500$

Cabe reiterar que esto solo aplica en nuestro contexto de grafo completo. El tiempo de ejecución puede variar de forma distinta si no se tienen encuenta los factores anteriormente mencionados, como el grado máximo o el tamaño de la clique máxima del grafo.

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 3 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.

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. 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.frontera > solucion.frontera) then
  |  |  esMejor ← verdadero
  |  |  solucion ← posibleMejorSolucion
  |  end if
  end while
```

Donde buscarMejorSolucionVecina corre las tres 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 las adyacencias de uno de los nodos de la clique (cualquiera, por lo que recorre como máximo $|maxAdyacente|$ nodos) 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, con cada nodo 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 $|maxAdyacente|$ veces que realizar una operación de complejidad $O(n)$, y como vimos en el análisis de la heurística golosa que $O(|maxAdyacente|) \subseteq O(min(n, m))$, acabamos teniendo $O(min(n, m) \times n)$. A esta solución obtenida le aplicamos finalmente el **Algoritmo Goloso**, cuya complejidad vimos que es $O(min(n, m) \times n)$, por lo que acabaríamos teniendo $O(2(min(n, m) \times n))$, que termina siendo $O(min(n, m) \times n)$.

Eliminar nodo

Este algoritmo recorre los nodos de la clique (como máximo, $min(n, m)$ 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 $O(min(n, m))$. Finalmente, como vimos, le corremos el **Algoritmo Goloso** a la solución, lo cual tiene una complejidad $O(min(n, m) \times n)$. Por ende, acotamos y definimos que la cota temporal de Eliminar Nodo es $O(min(n, m) \times n)$.

Intercambiar nodos

Este algoritmo, básicamente, consta de dos ciclos: el primero, que recorre cada uno de los nodos de la clique (a través de la variable i); y el segundo, que recorre la lista de adyacencia de uno de los nodos de la clique (particularmente, el del nodo $i+1$). Por lo tanto, tenemos:

Para cada nodo i en la clique, para cada nodo j adyacente a $i+1$, agregar j y sacar i

Esto significa que tenemos dos ciclos que dependen del tamaño de la clique y, por ende, del mayor grado de la clique (es decir, $|maxAdyacente|$). Por lo tanto, al tener dos ciclos anidados, acabamos teniendo $O(|maxAdyacente| \times |maxAdyacente|)$, que acotamos por $O((min(n,m))^2)$.

Ahora, queda ver que es lo que ocurre adentro del ciclo. Luego de quitar el nodo i, debemos fijarnos si el nodo j es adyacente a todos los que quedaron en la clique. Para esto, utilizamos **esAdyacenteATodos**, que tiene una complejidad de $O(n)$, y en caso de que efectivamente sea adyacente, agregamos el nodo j a la clique y le corremos el **Algoritmo Goloso**, para luego comparar si el resultado obtenido es mejor. Esto quiere decir que, para cada iteración, corremos $O(n + min(n,m) \times n)$. Por ende, al acotar este algoritmo, tendremos un ciclo de $min(n,m)^2$ pasos de complejidad $O(min(n,m) \times n)$. Es decir, y finalizando con la cota de **Intercambiar Nodos**, que la complejidad es de $O((min(n,m))^3 \times n)$

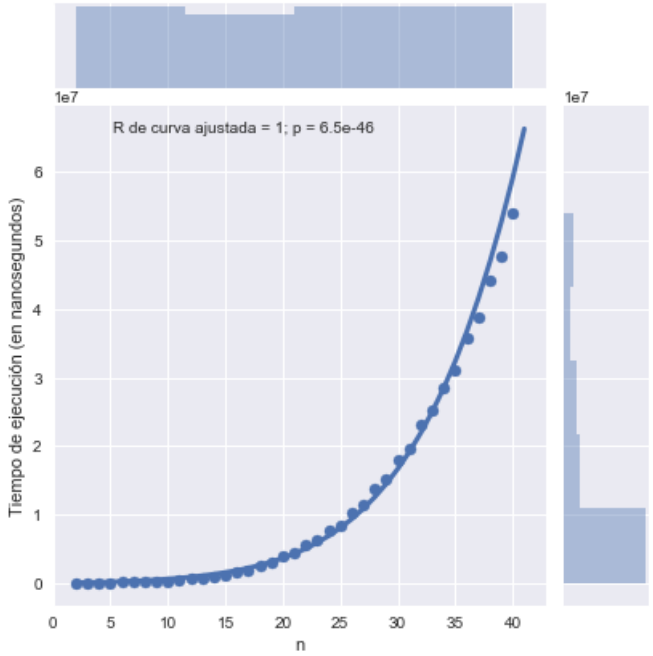
Así, alcanzamos las complejidades de cada una de nuestras vecindades. Por lo tanto, la complejidad de buscar la mejor solución vecina será $O(min(n,m) \times n + min(n,m) \times n + (min(n,m))^3 \times n)$, que por propiedades de O grande se acota finalmente con $O((min(n,m))^3 \times n)$. Puesto que el algoritmo general es, simplemente, tomar una clique y correrle el algoritmo hasta que no mejore, el mismo correrá un máximo de n veces (puede agregar o sacar cada nodo una vez, y no tiene sentido que lo haga más veces, ya que esto implicaría contradecir decisiones anteriores que mejoraban la solución). Por lo tanto, al considerar el peor caso, habremos corrido n veces un algoritmo de costo $O((min(n,m))^3 \times n)$, dejándonos con una complejidad temporal final de $O((min(n,m))^3 \times n^2)$

4.4. Casos Patológicos

Los casos patológicos de la heurística de búsqueda local tienen la complejidad de que partimos de una solución, por lo que si la solución original no es vecina a la mejor solución no vamos a poder encontrarla. Podemos afirmar que la familia de instancias que definimos en la heurística golosa, tambien son malos casos para esta heurística, ya que la solución que nos entrega la golosa, no se encuentra en la vecindad de la la solución óptima, por lo que no lograremos alcanzar esta solución. Hay que tener en cuenta que la heurística depende de la solución inicial, por lo que el caso patológico solamente se va a dar si la solución inicial es la que estamos describiendo.

4.5. Experimentación

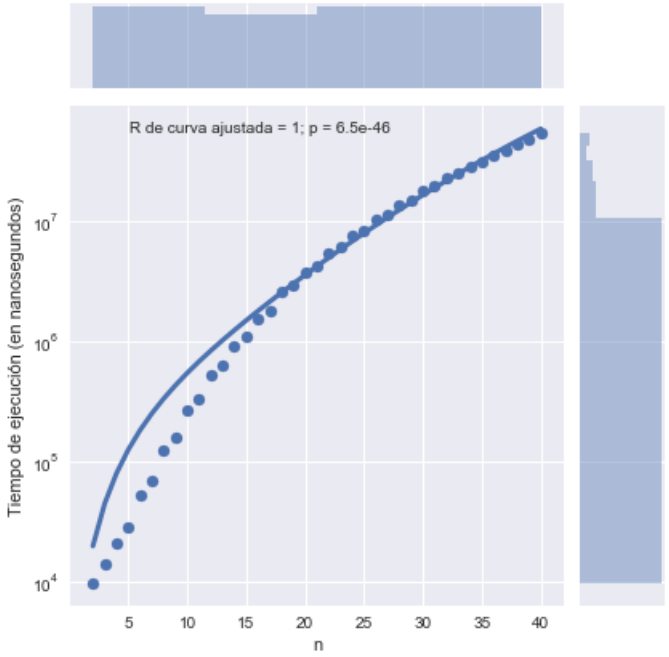
Al igual que con la heurística golosa, el tiempo de ejecución de este algoritmo depende de varios aspectos del grafo, además de n y m. Por esto, decidimos solo realizar un análisis del peor caso para comprobar las cotas propestas.



Datos del gráfico

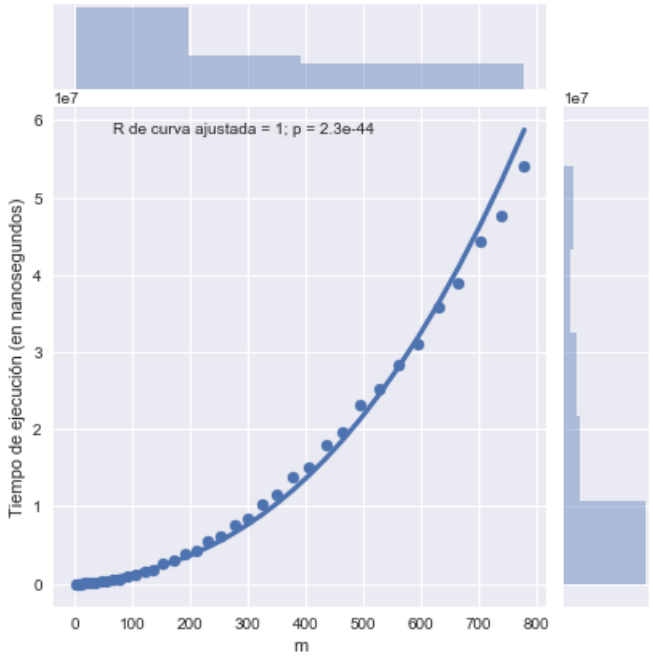
Grafo completo	$m = \frac{n*(n-1)}{2}$
Curva aproximada	$f(x) = (x^5)/2 + 5000 * x^2$

Dado que el exponente que forma parte de nuestra cota de complejidad es bastante elevado, decidimos observar estos mismos datos con 2 enfoques distintos: por un lado, en escala logarítimica, y por el otro, en base a m (que, al ser grafos completos, equivale a n^2).



Datos del gráfico

Grafo completo	$m = \frac{n*(n-1)}{2}$
Curva aproximada	$f(x) = (x^5)/2 + 5000 * x^2$



Datos del gráfico

Grafo completo	$m = \frac{n*(n-1)}{2}$
Curva aproximada	$f(x) = 3 * (x^{2,5}) + 10000 * x$

Podemos notar en estos gráficos que las constantes elegidas para las curvas están un poco sobreajustadas. Sin embargo, utilizando el R de Pearson podemos aseverar que la complejidad en sí es correcta, ya que este coeficiente detecta correlaciones lineales y no depende de las constantes asociadas a cada elemento.

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 la mejor. En otras palabras, quisiéramos ramificar la exploración del espacio de soluciones, para así aumentar nuestras posibilidades de encontrar la clique de máxima frontera.

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 (es decir, que se elija aleatoriamente una opción buena dentro del X por ciento más alto). 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 aleatoriamente uno 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 y, luego, se elige uno aleatoriamente entre los primeros. 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:

- **nodos(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 como lo usamos para ordenar toda la lista de nodos, 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 $O(1)$.

Con estos costos analizados, y considerando que la función **esAdyacenteATodos** fue analizada en casos anteriores, podemos pasar a analizar la complejidad de la función recursiva. Dado que esta función 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, donde por dentro 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 se reduce a considerar las n llamadas recursivas, que cuentan dentro con un ciclo de n nodos y un algoritmo de complejidad $O(n)$. Por lo tanto, nuestra complejidad acaba siendo $O(n^3)$.

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. Por otro lado, hay que considerar que en cada iteración del ciclo se realiza una llamada a localSearch(randomGreedy()). Por lo tanto, lo que acaba ocurriendo es que se suman las complejidades de ambos algoritmos (pues Local Search corre solo una vez sobre la clique generada, por lo que su complejidad es ajena a la de randomGreedy). Por ende, el costo de localSearch(randomGreedy()) es $O(n^3) + O((min(n,m))^3 \times n^2)$, y como no podemos afirmar nada sobre $min(n,m)$, no podemos realizar ninguna cota. Por lo tanto, nuestra complejidad acaba siendo $O(iteraciones \times (n^3 + min(n,m))^3 \times n^2))$.

5.3. Casos Patológicos

Ya que la metaheurística tiene un grado de aleatoriedad, es difícil encontrar casos patológicos. Sin embargo, como la misma utiliza internamente las heurísticas de búsqueda local y constructiva golosa, cuyo casos patológicos son parecidos, tendremos en cuenta este tipo de instancias. De cualquier manera, hay que ganarle a la probabilidad que se seleccione algún nodo de los de la clique de la mejor solución. Para evitarlo, hay que disminuir la probabilidad de

que seleccione uno de estos nodos, por lo que podemos generar muchas componentes como la de mayor grado para que sea muy probable que, en las iteraciones, seleccione estos nodos en vez de los de la solución máxima. Notemos que podemos generar la cantidad que queramos de este tipo de componentes por lo que mientras más generemos, menos probabilidad tendremos de seleccionar la clique.

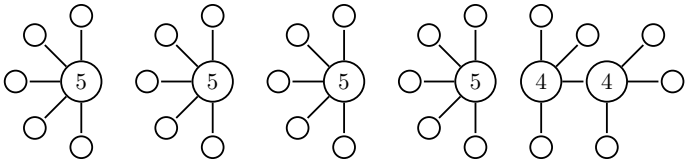
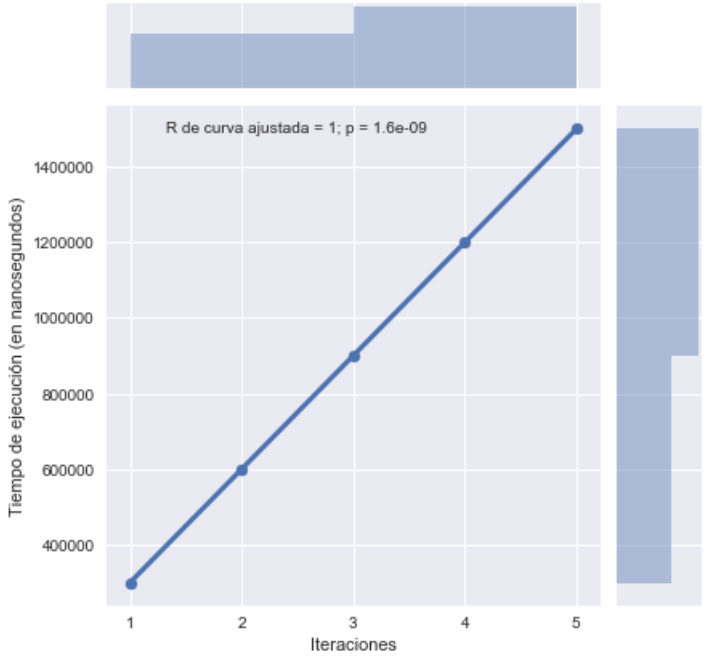


Figura 2: En la imagen podemos apreciar cómo copiamos 4 veces la componente que tiene el nodo de mayor grado, así estas componentes tienen más probabilidad de ser seleccionadas.

5.4. Experimentación

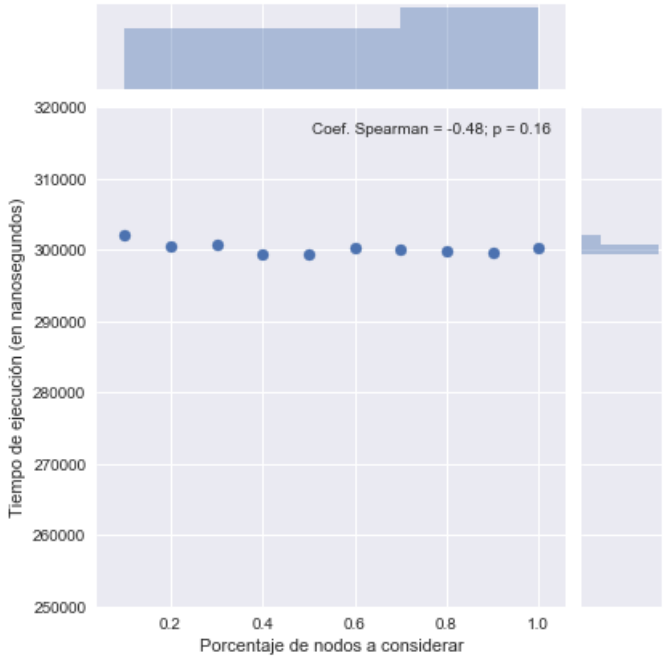
Al experimentar con la metaheurística, primero decidimos analizar el impacto lineal de las iteraciones:



Datos del gráfico

	$n = 10$
	$m = \frac{n*(n-1)}{2} = 45$
Porcentaje de nodos considerados	$p = 0{,}5$
Curva aproximada	$f(x) = 72500 * x * 10000$

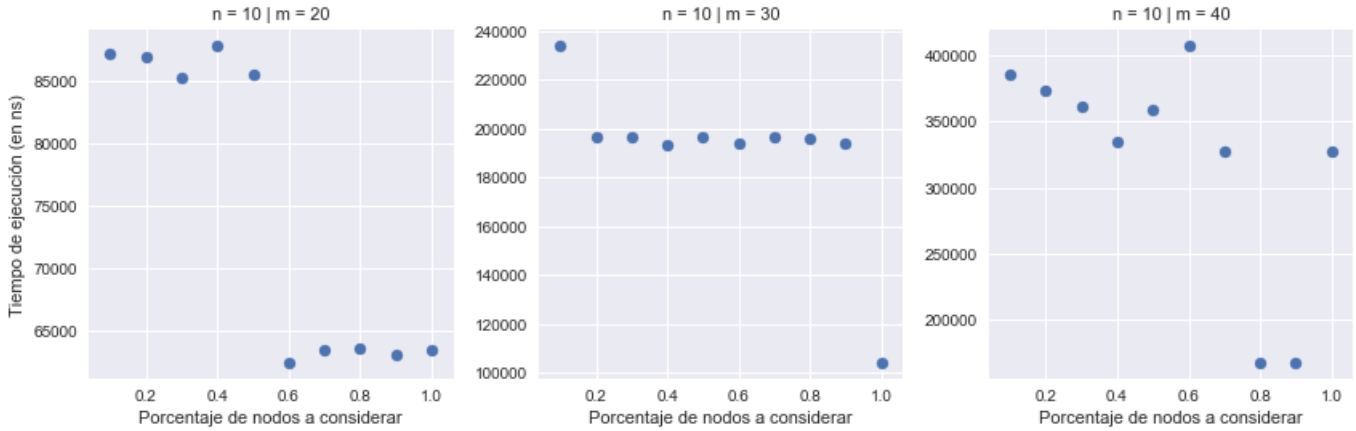
Este resultado era más que esperado, ya que resulta de ejecutar las mismas operaciones una cantidad fija de veces. Teniendo esto en cuenta, podemos analizar los demás factores en el caso de una única iteración.



Datos del gráfico

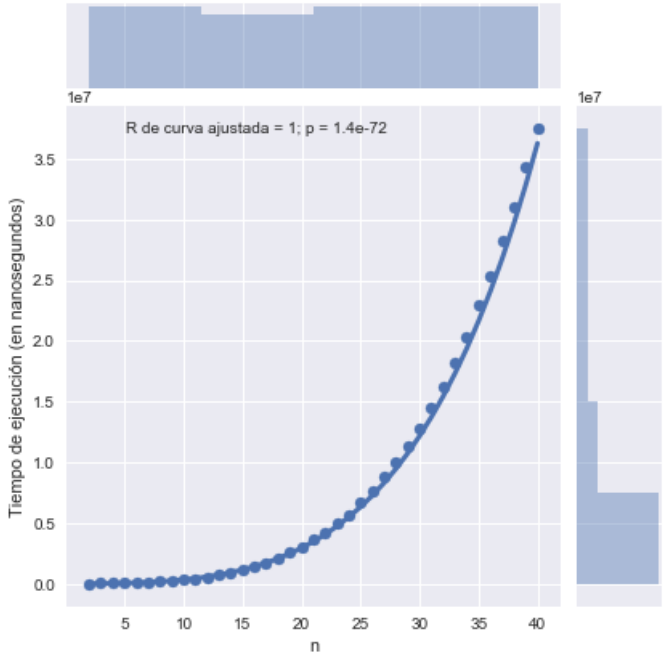
	$n = 10$
	$m = \frac{n*(n-1)}{2} = 45$
Iteraciones	$it = 1$

Dado que los grafos analizados son completos, el porcentaje de nodos a considerar no influye de manera significativa en el tiempo de ejecución. Esto se debe a que la clique máxima es el grafo de entrada en su totalidad. Sin embargo, este porcentaje puede influir en el tiempo de ejecución si el grafo no es uniforme:



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.

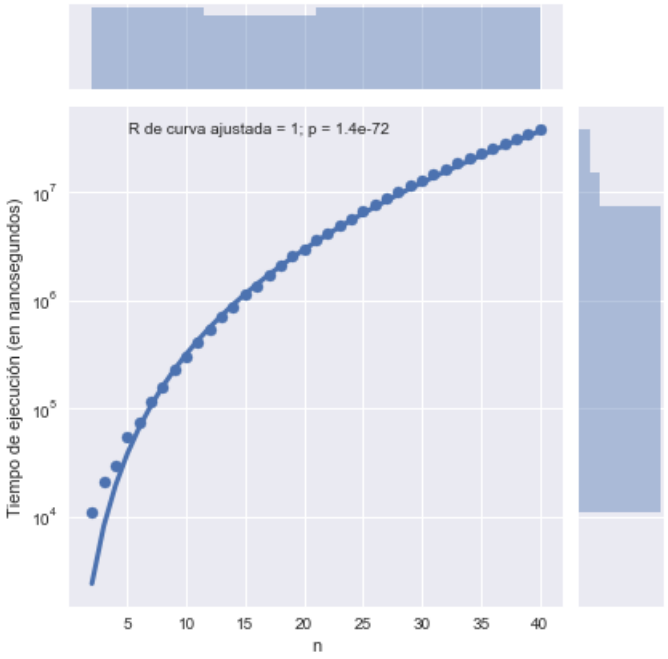
Una vez analisados estos factores y su impacto, nos dispusimos a medir el tiempo de ejecución en base al tamaño del grafo (utilizando grafos completos una vez más para analizar peor caso):



Datos del gráfico

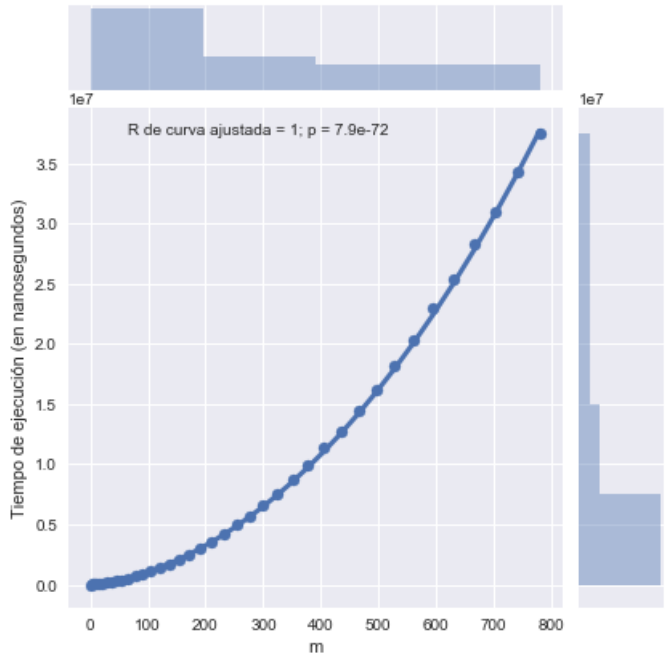
Grafo completo	$m = \frac{n*(n-1)}{2}$
Porcentaje de nodos considerados	$p = 0{,}5$
Iteraciones	$it = 1$
Curva aproximada	$f(x) = (x^5)/6 + 300 * x^3$

Como se mencionó, la cota de complejidad de GRASP es muy similar a la de una búsqueda local. Por ende, también realizamos gráficos similares a estos, con escala logarítmica y en base a m.



Datos del gráfico

Grafo completo	$m = \frac{n*(n-1)}{2}$
Porcentaje de nodos considerados	$p = 0{,}5$
Iteraciones	$it = 1$
Curva aproximada	$f(x) = (x^5)/6 + 300 * x^3$



Datos del gráfico

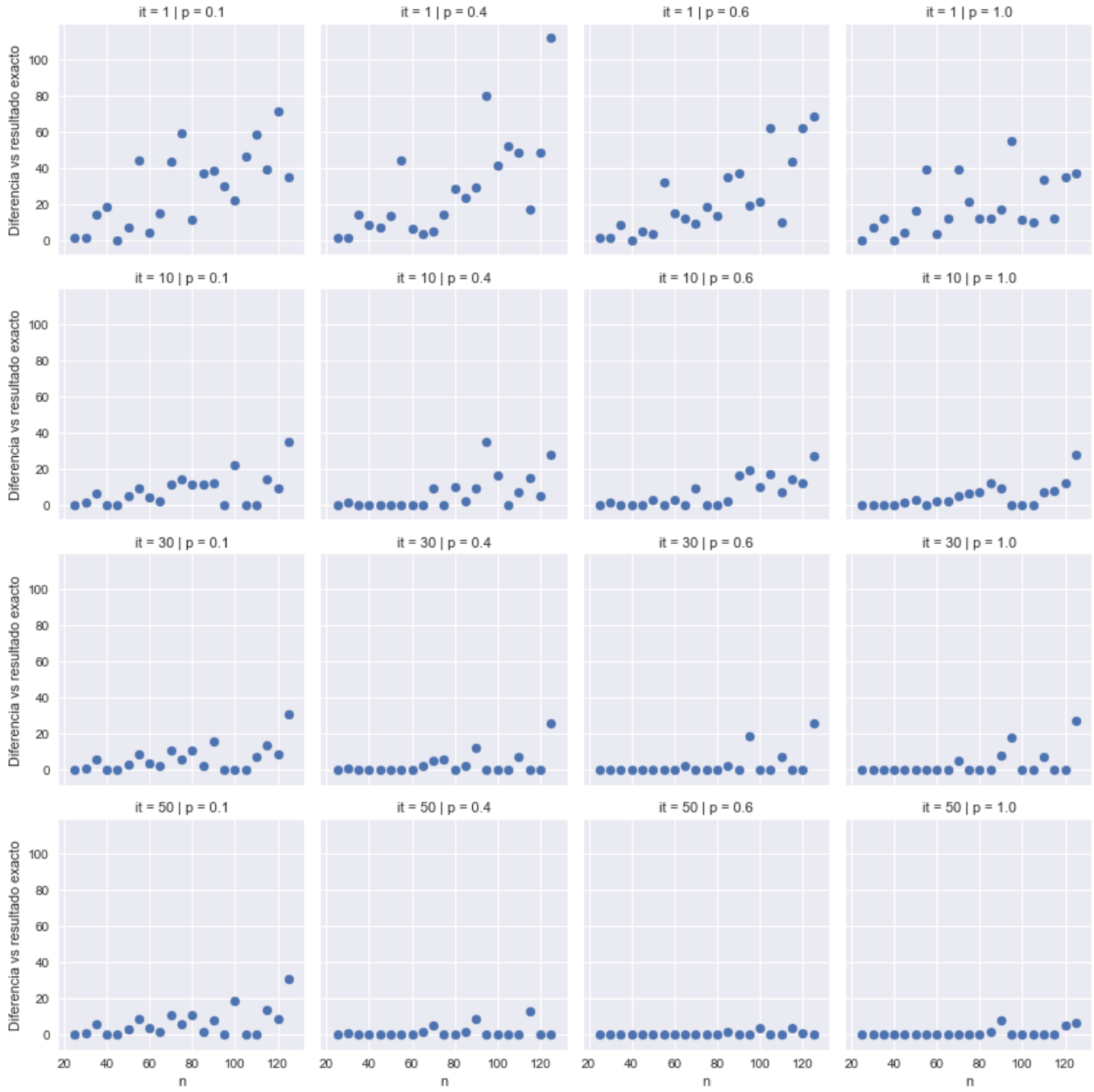
Grafo completo	$m = \frac{n*(n-1)}{2}$
Porcentaje de nodos considerados	$p = 0{,}5$
Iteraciones	$it = 1$
Curva aproximada	$f(x) = x^{2,5} + 950 * x^{1,5}$

En este caso, parece que la curva propuesta es mucho más precisa que la elegida para la búsqueda local, tanto de forma visual como por el p-valor asociado al coeficiente de Pearson

5.5. Entrenamiento de la metaheurística

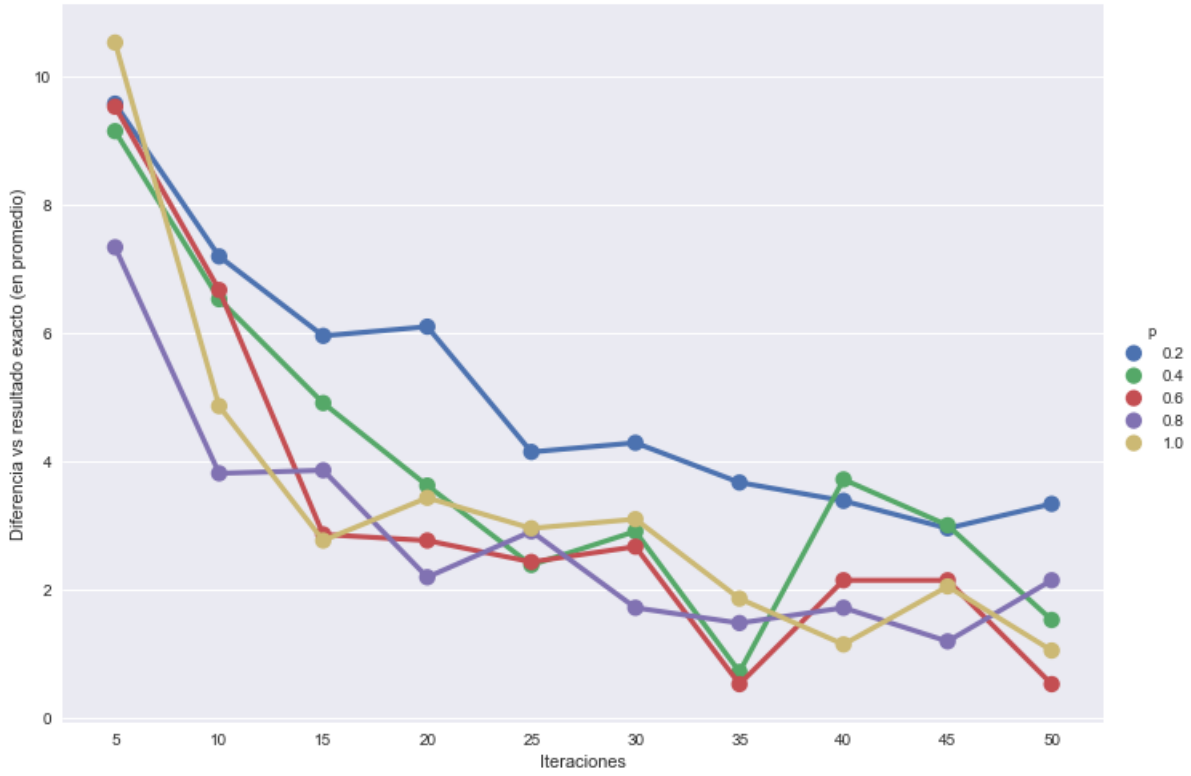
Además de experimentar con los tiempos de ejecución, decidimos analizar la influencia de ambos parámetros de entrada de la metaheurística en la precisión de los resultados obtenidos. Para esto, utilizamos un conjunto de tests de tamaño moderado, cuyas soluciones fueron obtenidas a través del algoritmo 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. Análisis de precisión de heurísticas

Dado que el objetivo de este TP es encontrar la mejor solución posible sin utilizar algoritmos exactos (ya que el problema en sí se resuelve en tiempo exponencial), nos dispusimos a comprar los resultados obtenidos entre todos los algoritmos, poniendo énfasis en la precisión de los resultados y su relación con el tiempo de ejecución de cada uno. La idea es hallar el algoritmo que mantenga la mejor proporción tiempo-calidad.

De cara a esto, sabemos que la heurística golosa es la más veloz. También suponemos que la aplicación de la metaheurística GRASP puede incrementar ampliamente la calidad de la solución (si se seleccionan correctamente los parámetros que la misma utiliza).

Para estos tests, se generaron nuevamente grafos en base a distribuciones uniformes tales que:

- $n \leq 40$
- $m \leq \frac{n*(n-1)}{4}$

Esto se debe a que los resultados deben ser comparados con los del algoritmo exacto, lo cual restringe el tamaño de los grafos a generar (o su solución no podría hallarse en tiempo razonable).

Además de hacer pruebas generales, se hicieron algunas pruebas específicas a los casos patológicos. Estos fueron generados con las siguientes propiedades (detalles adicionales sobre el funcionamiento de estos generadores se encuentran en el apéndice):

- $k \leq 15$
- $q == 10$ (exclusivamente para GRASP)

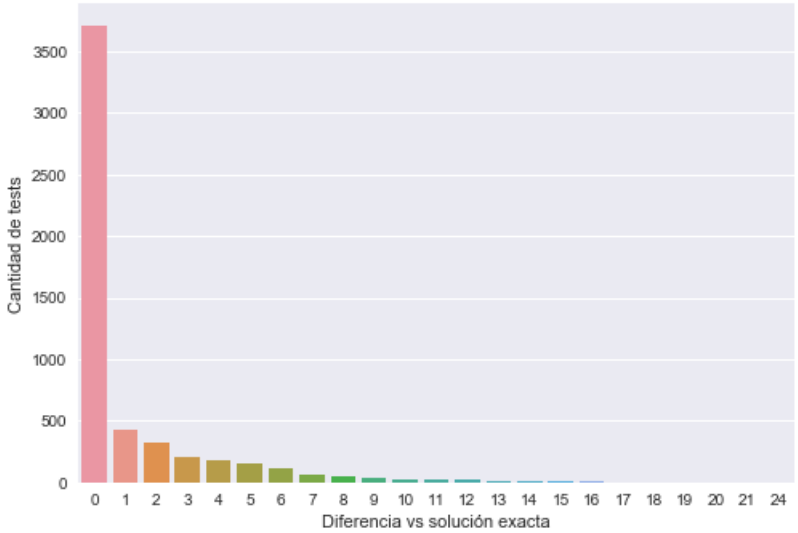
Para cada heurística, analizaremos los siguientes aspectos de las pruebas:

- el porcentaje de resultados que difieren con la solución exacta (independientemente de la diferencia real),
- el error máximo realizado por el algoritmo,
- el error promedio de todos los casos (esto incluye aqueyos sin error),
- la desviación estandar de dicha diferencia, y
- el tiempo de ejecución promedio de las heurísticas

Creemos que con estos indicadores, podemos analizar correctamente el comportamiento de cada algoritmo y realizar comparaciones entre los mismos.

6.1. Heurística constructiva golosa

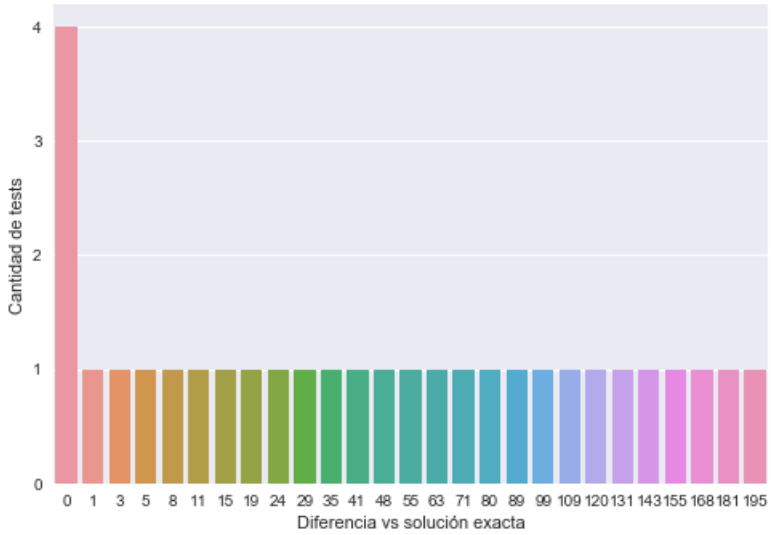
Primero decidimos mirar la heurística golosa, ya que las otras están pensadas para funcionar sobre la misma:



A simple vista, en promedio, la heurística parece bastante precisa. De base podemos ver que el enfoque es efectivo para resolver el algoritmo, aunque no necesariamente suficiente. De estas pruebas pudimos extraer los siguientes indicadores:

Porcentaje de resultados con error	30.83 %
Errores	Δ
- máximo	$max(\Delta) = 24$
- promedio	$\mu(\Delta) = 1{,}18$
- desviacion estandar	$\sigma(\Delta) = 2{,}52$
Tiempo (en ns)	Δ
- máximo	$max(t) = 31079$
- promedio	$\mu(t) = 10122{,}04$
- desviacion estandar	$\sigma(t) = 6429{,}43$

Como podemos ver, la heurística golosa de por sí no es tan exacta, aunque en promedio los errores son menores. Si miramos especificamente los casos patológicos del enfoque goloso:

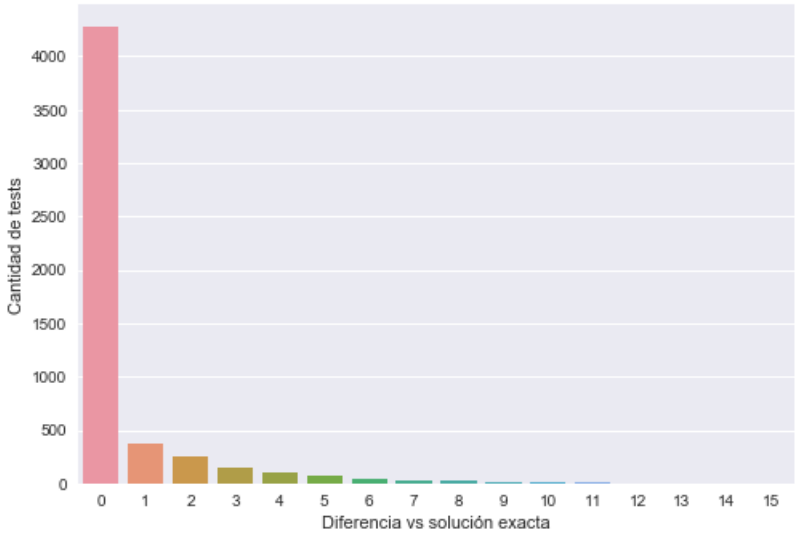


% de resultados con error	86.67 %
Errores	Δ
- máximo	$max(\Delta) = 195$
- promedio	$\mu(\Delta) = 63{,}27$
- desviacion estandar	$\sigma(\Delta) = 61{,}71$
Tiempo (en ns)	t
- máximo	$max(t) = 13200$
- promedio	$\mu(t) = 7102{,}53$
- desviacion estandar	$\sigma(t) = 3629{,}22$

Podemos ver que en estos casos, el algoritmo comete errores enormes, y en general no encuentra el caso ideal. Para grafos de menor tamaño, el caso patológico no es tal, pero a medida que aumenta el tamaño del grafo, la solución es relativamente peor.

Cabe destacar que al analizar los tiempos de ejecución, el caso patológico no es necesariamente más lento que un grafo promedio de tamaño equivalente. Si bien los tiempos de ejecución no son comparables por los distintos tamaños, en general la calidad del resultado no influye (al menos de forma negativa) en los tiempos de ejecución.

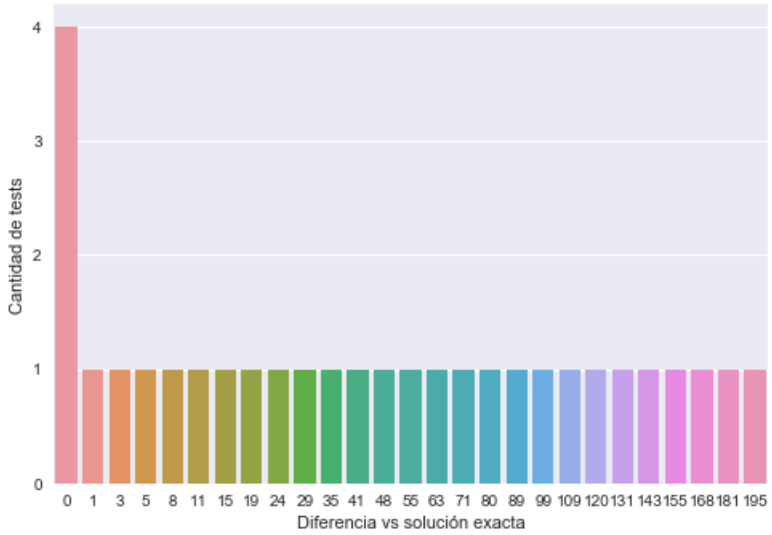
6.2. Heurística de búsqueda local



Al aplicar una búsqueda local sobre la metodología greedy, podemos ver una amplia mejoría de los resultados obtenidos. La misma se evidencia mejor en los indicadores:

Porcentaje de resultados con error	20.21 %
Errores	Δ
- máximo	$max(\Delta) = 15$
- promedio	$\mu(\Delta) = 0{,}59$
- desviacion estandar	$\sigma(\Delta) = 1{,}56$
Tiempo (en ns)	Δ
- máximo	$max(t) = 1886955$
- promedio	$\mu(t) = 189041{,}69$
- desviacion estandar	$\sigma(t) = 202448{,}43$

En promedio, el error cae por debajo de 1, y la cantidad de casos con resultado exacto aumenta significativamente. Sin embargo, sigue habiendo varios errores, incluyendo algunas diferencias grandes.



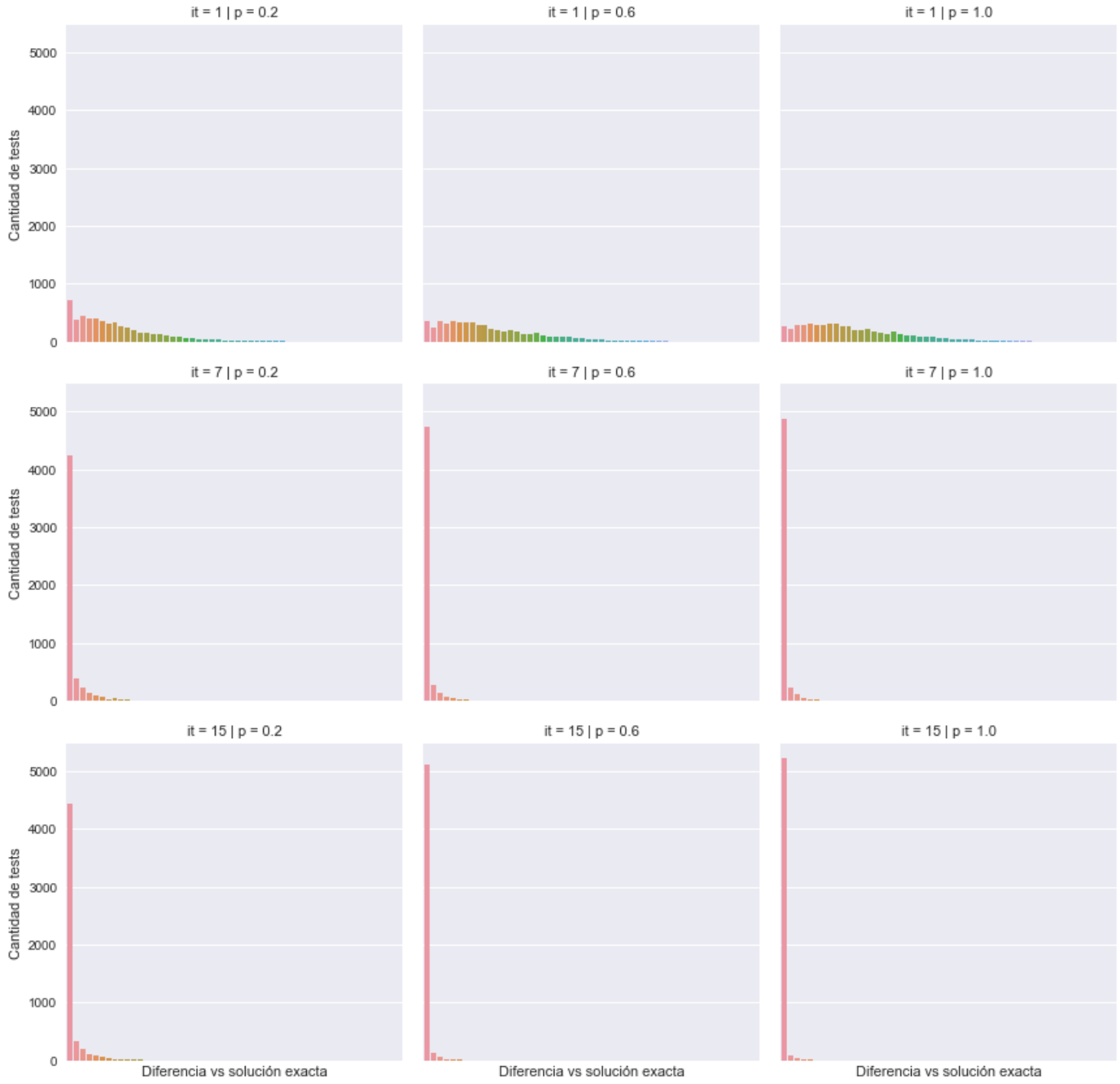
% de resultados con error	86.67 %
Errores	Δ
- máximo	$max(\Delta) = 195$
- promedio	$\mu(\Delta) = 63{,}27$
- desviacion estandar	$\sigma(\Delta) = 61{,}71$
Tiempo (en ns)	t
- máximo	$max(t) = 46304$
- promedio	$\mu(t) = 25391{,}73$
- desviacion estandar	$\sigma(t) = 12556{,}97$

Desafortunadamente, en el caso del grafo patológico, los resultados son exactamente iguales, ya que la forma del grafo no permite intercambiar un único nodo e incrementar la frontera, resultando en un máximo local que no puede mejorar.

No solo eso, dado que el grado máximo es muy elevado, el tiempo de ejecución se triplica en promedio al de la heurística golosa, ya que busca mejoría entre sus nodos adyacentes y se haya con muchos casos idénticamente malos.

6.3. Metaheurística GRASP

Al analizar la precisión de GRASP, tuvimos que tener en cuenta los parámetros de la metaheurística: el porcentaje de los nodos considerados y la cantidad de iteraciones aleatorias. La combinatoria de estos dos parámetros es muy grande, así que decidimos mostrar solo algunos de los valores que consideramos representativos de las tendencias generales:



Si bien este gráfico no permite un análisis muy profundo, muestra una tendencia básica: mientras más iteraciones se realizan, más errores se depuran, y la solución se aproxima más al resultado exacto. No solo eso, realizando una única iteración, los resultados de la metaheurística son paupérrimos, peores que la heurística golosa simple. Esto se debe a que la instancia golosa tomada es aleatoria, por lo que varía con respecto a su contraparte estrictamente golosa, y suele ser peor aunque puede superar.

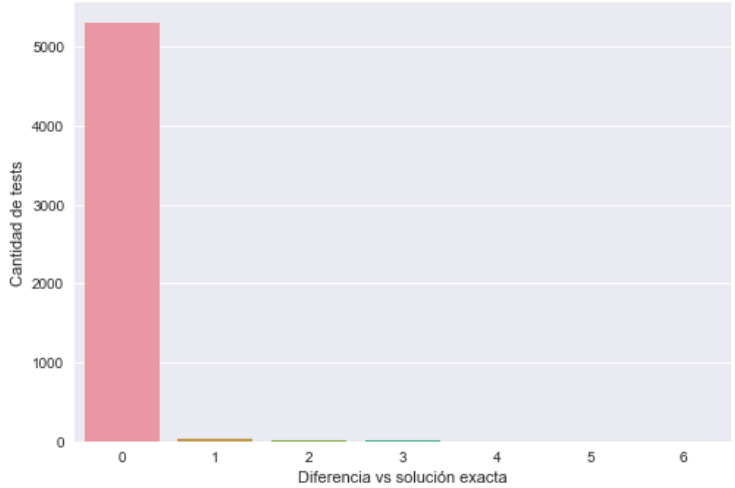
% de nodos considerados	Iteraciones	% de error	$max(\Delta)$	$\mu(\Delta)$	$\sigma(\Delta)$	$\mu(t)$
0.2	1	86.5 %	48	7.134	6.694	384638
	7	21.0 %	27	0.696	1.939	2351098
	15	17.5 %	18	0.530	1.598	5021068
0.6	1	93.4 %	59	9.687	7.898	346049
	7	11.8 %	15	0.307	1.133	2339655
	15	4.7 %	9	0.097	0.562	5009071
1	1	95.1 %	60	10.747	8.241	315578
	7	9.3 %	13	0.227	0.952	2350323
	15	2.7 %	9	0.054	0.401	4995022

Mirando un poco más de cerca los datos obtenidos, se ve de manera mucho más clara en qué influye cada parámetro. En particular, podemos destacar algunas cosas:

1. el factor aleatorio de GRASP puede derivar en resultados mucho menos fiables que usando una heurística golosa simple o con búsqueda local;
2. si bien el porcentaje de nodos considerados influye en la tasa de error, la cantidad de iteraciones realizadas tiene un impacto mucho más visible, y es lo que permite a GRASP obtener mejores resultados que greedy, utilizando el aspecto random de manera ventajosa en lugar de generar casos patológicos;
3. aumentar el porcentaje de nodos considerados puede ser negativo si no se utilizan más iteraciones, ya que permite al aleatorio elegir peores opciones;
4. más allá de lo recién dicho, la cantidad de iteraciones sigue una suerte de ley de retornos decrecientes, y aumentar la cantidad de repeticiones linealmente no genera resultados linealmente mejores, ya que las instancias aleatorias comienzan a repetirse; y
5. por este factor aleatorio, el tiempo de ejecución disminuye al aumentar p ya que se consideran casos subóptimos con cliques más pequeñas, que resultan en menor tiempo de ejecución de la contrucción golosa.

Entre estos casos particulares, los parámetros que mejores resultados brindan son $p = 1$ e $it = 15$. Muy cerca de estos resultados se encuentra $p = 0{,}6$ e $it = 15$, que en promedio dan peores resultados con un costo de ejecución comparable o peor.

A modo de comparación con las pruebas realizadas en la sección de la metaheurística, probamos los parámetros particulares que fueron entregados con el ejecutable de este TP: $p = 0{,}6$ e $it = 50$.

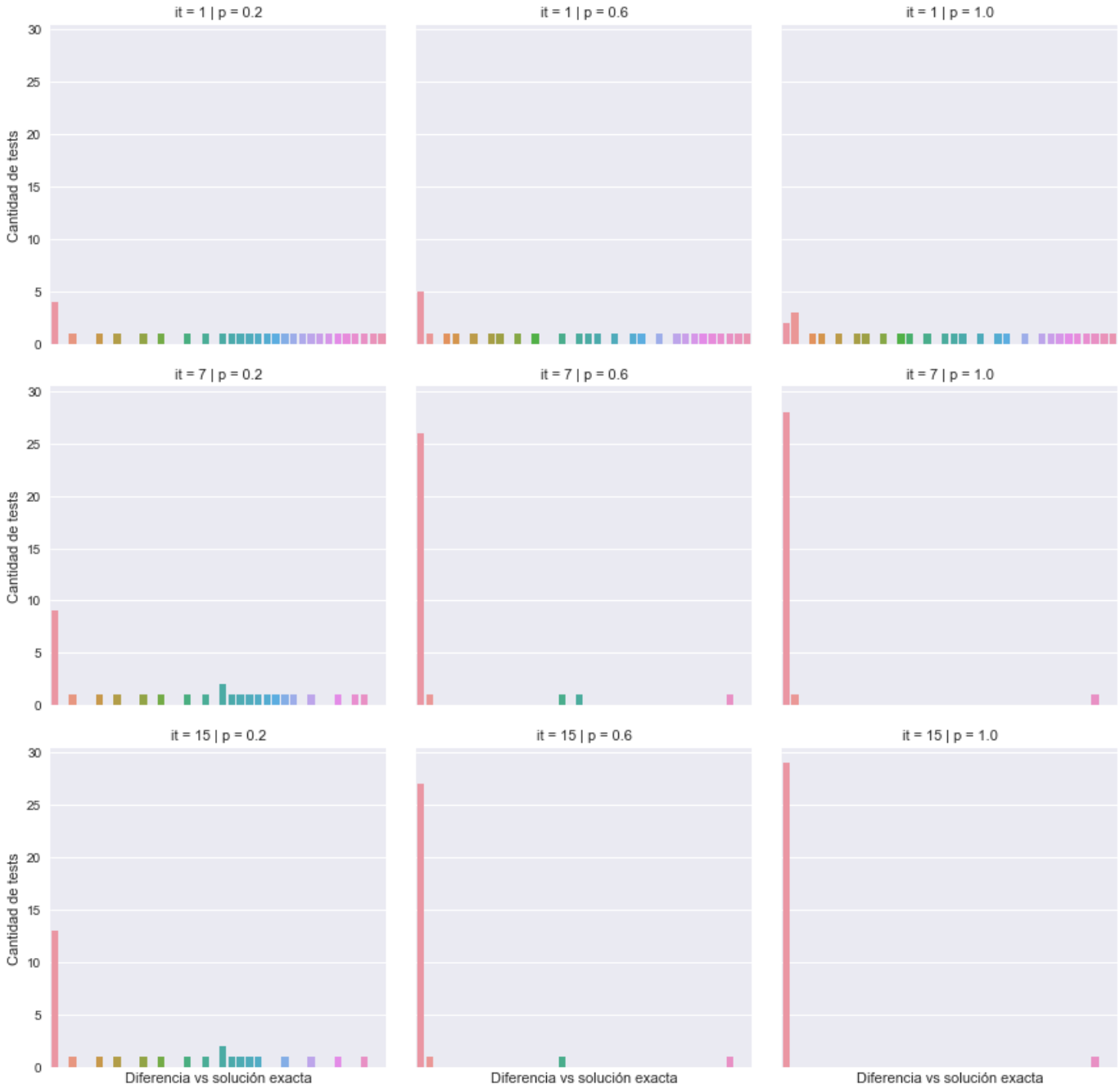


% de resultados con error	1.04 %
Errores	Δ
- máximo	$max(\Delta) = 6$
- promedio	$\mu(\Delta) = 0{,}018$
- desviacion estandar	$\sigma(\Delta) = 0{,}21$
Tiempo (en ns)	t
- máximo	$max(t) = 99235601$
- promedio	$\mu(t) = 15165859$
- desviacion estandar	$\sigma(t) = 14519656$

Como se puede notar, el resultado es mucho más preciso que algunos de los casos mostrados anteriormente. Es más, en este caso, aumentar las iteraciones de 15 a 50 (resultando en al menos el triple del tiempo de ejecución) produjo una diferencia de calidad porcentualmente visible (se redujeron los errores a la mitad, con mucho menor promedio y desviación).

Sin embargo, en términos netos, la diferencia no es tanta, ya que con muchas menos iteraciones la heurística estaba generando buenos resultados. Es difícil cuantificar esta diferencia, ya que depende mucho de los casos de prueba provistos y de la precisión deseada, pero vale la pena mencionar que este aumento en tiempo de ejecución podría no ser valioso.

En cuanto a los casos patológicos, primero decidimos comparar el rendimiento de GRASP frente al caso base que afecta a las heurísticas golosa y de búsqueda local:



% de nodos considerados	Iteraciones	% de error	$max(\Delta)$	$\mu(\Delta)$	$\sigma(\Delta)$	$\mu(t)$
0.2	1	86.7 %	194	35.617	29.447	204206
	7	70.0 %	168	22.350	24.291	1326573
	15	56.7 %	168	15.767	21.808	3081457
0.6	1	83.3 %	194	31.150	31.244	241819
	7	13.3 %	168	4.083	15.847	2228526
	15	10.0 %	168	3.400	15.554	5182525
1	1	93.3 %	194	31.667	30.820	240112
	7	6.7 %	168	2.817	15.333	2612865
	15	3.3 %	168	2.800	15.336	5901750

Estos resultados fueron muy interesantes porque, a diferencia de la tendencia vista en promedio, el tiempo de ejecución aumentó junto con p. Esto se debe a la distribución de los ejes en nuestro grafo: ya que el nodo de mayor grado solo permite generar cliques pequeñas, y cada nodo debe ser comparado contra aquellos que forman la solución, los ciclos del algoritmo goloso son más cortos.

Al probar con los parámetros entregados en el ejecutable del TP ($p = 0{,}6, it = 50$), logramos eliminar todos los

errores, pero debemos recalcar que el tiempo de ejecución es más del triple, lo cual puede ser prohibitivo dependiendo del contexto.

Por último, decidimos generar casos patológicos especificamente diseñados para GRASP. La lógica interna es similar a la de los casos patológicos de las otras heurísticas, pero con el adicional de agregar múltiples falsos positivos (componentes estrella) en un intento de aumentar la probabilidad de que sean seleccionados, generando malos resultados.

% de nodos considerados	Iteraciones	% de error	$max(\Delta)$	$\mu(\Delta)$	$\sigma(\Delta)$	$\mu(t)$
0.2	1	86.7 %	195	63.267	61.710	66185
	7	86.7 %	195	63.267	61.710	975937
	15	86.7 %	195	63.267	61.710	2254688
0.6	1	86.7 %	195	63.267	61.710	193265
	7	86.7 %	195	63.267	61.710	1295279
	15	86.7 %	195	63.267	61.710	2713630
1	1	93.3 %	195	63.333	61.640	456276
	7	40.0 %	195	23.133	48.345	1884723
	15	26.7 %	195	12.033	37.490	4400884

Aquí es claro el rol del parámetro p en la heurística. Si bien existen casos (en particular, casos favorables para la heurística golosa) donde considerar más nodos resulta en una pérdida de tiempo y resultados menos fiables, en estos casos se ve exactamente lo opuesto: considerar la mayor cantidad de nodos resulta en resultados más precisos, ya que la cantidad de falsos positivos es muy grande.

Por supuesto, debe recordarse que GRASP tiene un alto componente aleatorio: $p = 1, it = 1$ resultó peor que otras instancias con menor p. Depurar estos errores luego de sucesivas iteraciones es una parte importante del funcionamiento de la metaheurística.

6.4. Resumen comparativo y conclusiones

A modo de referencia, se muestra aquí una comparación entre todas las heurísticas analizadas:

	Gráfos uniformes				Gráfos patológicos			
Heurística	% error	$\mu(\Delta)$	$\sigma(\Delta)$	$\mu(t)$	% error	$\mu(\Delta)$	$\sigma(\Delta)$	$\mu(t)$
Constructiva Golosa	30.83 %	1.18	2.52	10122	86.67 %	63.27	61.71	7102
Búsqueda Local	20.21 %	0.59	1.56	189041	86.67 %	63.27	61.71	25391
GRASP (p = 1, it = 15)	2.7 %	0.05	0.40	4995022	26.7 %	12.03	37.49	4400884
GRASP (p = 0.6, it = 50)	1.04 %	0.02	0.21	99235601	76.7 %	45.13	49.26	9164302

Como se puede ver, el uso de heurísticas más complejas conlleva un gran aumento del tiempo de ejecución. Algo que nos llamó la atención fue hecho que la búsqueda local, si bien mejora los resultados, tiene un gran costo relativo a los errores resueltos, y sufre los mismos casos patológicos. Por otro lado, la metaheurística reduce ampliamente los errores, pero su uso de búsquedas locales y el factor aleatorio significan que debe ser usado con precaución.

Otro detalle interesante es como el tiempo de ejecución de GRASP puede resultar inversamente proporcional a la calidad del resultado hallado: si bien realizamos más del triple de iteraciones, al utlizar un p menor y no hallar el caso óptimo cada iteración tarda menos porque las cliques a considerar son más pequeñas.

En conclusión, el uso de una metaheurística puede brindar resultados muy cercanos a un algoritmo preciso, pero debe ser optimizada con cuidado en cada uno de sus componentes (incluyendo tanto sus parámetros como sus heurísticas internas) para que el costo de ejecución a pagar por soluciones de mayor calidad no sea muy alto, y para evitar casos patológicos.

7. Apéndices

7.1. 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, al igual que para realizar pruebas comparativas entre las distintas implementaciones.

- Un generador de grafos para casos patológicos

  Para analizar el comportamiento de nuestros algoritmos frente a ciertas instancias particulares, creamos un generador de grafos que no es aleatorio, sino que dados ciertos parámetros genera determinísticamente grafos poco favorables de las heurísticas.

  Dados k y q, k es el mayor grado entre los nodos del grafo, existen q componentes donde hay un nodo de grado k donde sus adyacentes son de grado 1, y además una componente tal que es una clique de $\frac{k}{2}$ nodos y cada uno de ellos tienen $\frac{k}{2}$ nodos adyacentes de grado 1(Como los descritos en la sección de casos patológicos).

  Estos grafos también se encuentran descritos en el análisis de las heurísticas y sus casos patológicos.

- 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 coeficientes de correlación o "R" de Pearson. El mismo denota la correlación lineal entre 2 variables aleatorias. Nostros lo aplicamos entre los datos obtenidos (en particular, el tiempo de ejecución) y la curva graficada que se aproxima a esos datos. De esta manera, podemos determinar si nuestra aproximación es correcto: un R cercano o igual a 1 indica que hay una fuerte correlación positiva entre los valores graficados y la curva, es decir, la curva aproxima correctamente al gráfico. Junto al coeficiente se incluye un p-valor para la hipotesis nula que los valores pueden haber sido generados sin correlación real; un p-valor elevado invalidaría una posible correlación positiva o negativa, lo cual significaría que nuestra aproximación es erronea.

7.2. 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).

8. Informe de cambios

Cambios generales

- Se agregaron nuevas situaciones reales.
- Se hizó un análisis para todas las heurísticas sobre las instancias donde la solución obtenida no es óptima (Casos patológicos).
- El uso previo del R de Pearson, como era provisto por la biblioteca de graficado de Seaborn, no era representativo de las curvas que nosotros aproximamos. Esto se rectificó, y ahora el R se calcula relacionando el gráfico y la curva de complejidad propuesta.
- Se ajustó la cota de la complejidad temporal de los algoritmos

Experimentación general

Se agregaron las experimentaciones generales (de complejidad) de los algoritmos faltantes. Al mismo tiempo, debido a cambios en las complejidades calculadas para los distintos algoritmos, además de un cambio de criterio de medición temporal (que antes incluía una conversión innecesaria entre representaciones de grafos) se midieron nuevamente todas las experimentaciones de las heurísticas.

Heurística de Búsqueda Local

Se eliminó la Vecindad de Local Search que quitaba dos nodos de la clique y agregaba dos nuevos. Si bien en un primer momento creímos que esta vecindad extendía nuestra posibilidad de alcanzar la clique de máxima frontera, al comparar los resultados que se obtenían con y sin la misma notamos que no existía diferencia en nuestros resultados. Sin embargo, sí se veía afectada la cota temporal del algoritmo, por lo que el costo que se pagaba por realizar este intercambio era demasiado para el mínimo beneficio que se obtenía.

De cualquier modo, es importante resaltar que esta decisión es tomada teniendo en cuenta las experimentaciones realizadas y los resultados observados (y, por ende, la repercusión del intercambio sobre nuestro problema). En otro problema, podría ocurrir que esta vecindad tenga una repercusión mayor o bien que su cota temporal sea más chica, justificando su uso.

Análisis de precisión de heurísticas

Se agregó una sección con la finalidad de comparar la calidad de los resultados obtenidos al aplicar cada heurística, y al mismo tiempo evaluar la relación costo/beneficio del uso de heurísticas más complejas.

Esta sección incluye:

- análisis de precisión o calidad de los resultados de cada heurística
- análisis de relación entre calidad y tiempo de ejecución
- experimentación con parámetros de metaheurística (movido de la sección de GRASP)
- experimentación de casos patológicos y su impacto en tiempo de ejecución y calidad de resultados
- comparación entre las distintas heurísticas en términos de tiempo de ejecución, calidad de resultados y casos patológicos