Algoritmos y Estructuras de Datos III

Primer Cuatrimestre de 2007

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

Trabajo Práctico 3

Integrante	LU	Correo electrónico
Blanco, Matias	508/05	matiasblanco18@gmail.com
Freijo, Diego	4/05	giga.freijo@gmail.com

Palabras Clave

Vertex Cover, Heurísticas, Metaheurísticas, Algoritmo Goloso, Búsqueda Local, GRASP

Índice

1. Introducción

El problema planteado para resolver es el llamado "Vertex Cover". Este consiste en encontrar un subconjunto de nodos tal que para todos los ejes del grafo, por lo menos uno de los nodos del subconjunto sea extremo de este. Para dar un ejemplo, en un grafo asi

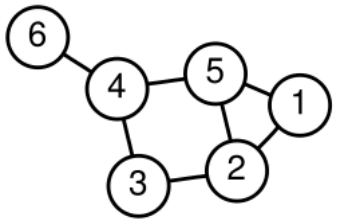

Figura 1: Grafo Ejemplo

los nodos 1,3,5,6 son un vertex cover, pero tambien lo es el 2,4,5.

Nuestro objetivo es, por medio de heuristicas (descriptas mas adelante), tratar de llegar al minimo vertex cover. Este es el minimo subconjunto que cumpla con la condicion.

Este problema pertenece a la familia de los problemas NP-Completos en la teoria de complejidades y es uno de los 21 problemas NP-Completos de Karp. Los problemas NP-Completos son los mas dificiles de la familia de los NP ("non-deterministic polynomial time").

Algunos ejemplos en donde se ve la aplicacion de esta resolucion de problema, serian:

- Se desean poner camaras de seguridad en el coqueto barrio Parque, tal que sea la minima cantidad de camaras y que estas vigilen todas las cuadras. Para esto se contrata a un grupo de computadores, que modelan el problema poniendo a las esquinas como nodos y a las cuadras como ejes entre esos nodos. Al aplicarle la solucion, se les podra dar una respuesta satisfactoria a los vecinos, y asi garantizar su seguridad, tanto de sus casas como de sus bolsillos.
- Un sistema de audio moderno, tiene, para no escatimar, muchos parlantes. Cada persona, por su poco oido, necesita que el sonido llegue desde uno solo de ellos, no es necesario mas. Se necesita llevarse algunos parlantes para otra sala. Al aplicarle la solucion, se utilizan a las personas como nodos y al sonido de cada parlante como eje. Se quitan aquellos parlantes tal que todas sus ondas de sonido lleguen a personas que ya estan escuchando por otro.

Al no tener solucion en tiempo polinomico (es decir de la forma n^k), en un grafo medianamente chico, el tiempo de ejecucion puede llevar mucho tiempo. Por ejemplo, un grafo de 50 nodos, va a realizar, en el caso del algoritmo exacto mas conocido (o sea de complejidad $O(2^n)$), 2^{50} operaciones, es decir 1.125.899.906.842.624 operaciones. Aproximando una cantidad de operaciones por segundo de 1.000.000, tardaria mas de 357020 siglos en dar una solucion exacta. Por mas que se tenga una maquina 100 veces mas rapida, igual tardaria mucho, o sea mas de 3570 siglos.

Por este motivo, se implementaron diferentes heuristicas (algoritmo que provee una combinacion de buena solucion y rapidez de ejecucion, pero sin garantizar, que la respuesta sea correcta).

La primera de ellas fue la heuristica golosa. Un algoritmo goloso es aquel que, para resolver el problema, hace una seleccion local optima en cada iteracion del mismo, con el objetivo de buscar un optimo global. En el caso del vertex cover, la funcion de seleccion optima local del goloso es buscar aquel nodo que tenga mayor cantidad de vecinos, con lo que se espera que poniendo estos nodos, se llegue mas rapido a recubrir todo el grafo. En el caso de un grafo rueda, por ejemplo, este algoritmo cumple con ser el minimo vertex cover, pero en los algoritmos aleatorios no siempre cumple con ese requisito (ver mas adelante en los graficos).

La segunda heuristica implementada fue busqueda local. Esta consiste en, a partir de una solucion inicial, moverse a traves de diferentes soluciones hasta que no aparezca una solucion mejor en ninguno de sus vecinos o hasta que se cumpla una condicion de parada (puede ser por tiempo o por cantidad de iteraciones realizadas). Graficando el problema como una funcion, esta heuristica busca un maximo o minimo local (dependiendo lo que se quiera realizar), que se espera que sea tambien el global, aunque no siempre coincide.

La tercera heuristica usada fue GRASP (Greedy randomized adaptive search procedure). Esta heuristica consiste en iteraciones hechas a partir de construcciones de una solucion inicial tomada de un algoritmo goloso random y una consiguiente secuencia de mejoras a partir de de la busqueda local.

A continuacion se explica con mayor detalle cada uno de los algoritmos implementados, asi como tambien su pseudocodigo, complejidad y conclusiones. Para finalizar, se evaluaran las 3 alternativas y se comparara con el exacto para observar cual fue la que mas se acerco a la solucion optima.

2. Planificación

En esta seccion se va a explicar las decisiones tomadas antes de arrancar a realizar el trabajo. Primero se decidio hacer las pruebas sobre 4 tipos de grafos:

- Completos
- Rueda
- Bipartito
- Aleatorio

Los .in de prueba fueron implementados en python para poder realizar mayor cantidad de los mismos. En los Completos varia la cantidad de nodos, asi como tambien en los Rueda. En los Bipartito varia la densidad del grafo y la cantidad de nodos. En el aleatorio tambien varian la densidad y la cantidad. Se consideraron estos a priori, porque se estimo que eran casos diferentes entre si y podrian arrojar algunas conclusiones al finalizar el trabajo. El aleatorio fue el mas usado para las pruebas, ya que permite tener una mejor muestra de comportamiento ante diferentes densidades y cantidades de nodos.

Para mejor implementacion, se desarrollaron las siguientes clases:

- Recubrimiento

  Para poder manejar mejor las listas de nodos, teniendo el metodo EsRecubrimiento y un comparador. Tambien guarda el objeto Estadistica, para ir aumentando el contador de instrucciones.

- Estadisticas

  Simplemente una clase con un entero que funciona como acumulador de instrucciones para cada ejecucion.

- Grafico

  El encargado de sacar al .dat las estadisticas resultantes de la ejecucion

- Parser

  Se encarga de leer los grafos y escribir la respuesta.

3. Exacto

3.1. Introducción

El algoritmo exacto que resuelve Vertex Cover es de los llamados algoritmos "malos", ya que no resuelve el problema en tiempo polinomico. Esto se debe a que este problema esta dentro de la familia de los NP-Completos, para los cuales todavia no se les encontro una solucion polinomica, y ni siquiera se sabe si va a ser posible esto. En investigaciones realizadas (ver la carpeta bib del cd), se encontro una solucion en tiempo menor a 2^n, aunque sigue siendo exponencial. Este algoritmo se comporta asi en grafos chicos con nodos de grados chicos. Este algoritmo requiere, por ejemplo en nuestras pruebas, mucho tiempo para resolver un grafo. Para citar un caso, en un grafo de 20 nodos con una densidad del 50 % el algoritmo tardo 25 minutos en una Intel Core 2 Duo 1.86Ghz con 2GB RAM DDR2 1000mhz.

3.2. Desarrollo

Para realizar esta implementacion se partio con una idea, aunque se termino implementando otra. La idea inicial era hacer un algoritmo que realice, primero, un conjunto con todos los subconjuntos posibles de nodos y a partir de ahi ir analizando si es recubrimiento y tomar el menor de todos. Esto realiza siempre todas las comparaciones y no tiene poda, lo que haria que el tiempo de ejecucion sea muy costoso. Para mejorar esto, se decidio implementar un algoritmo recursivo, que, si bien en un peor caso puede semejarse al algoritmo anteriormente mencionado, en el caso promedio se porta mejor en cuestion de tiempo.

El algoritmo lo que hace basicamente es, primero una verificacion. Esta consiste en ver si la solucion de entrada es recubrimiento. Si lo es, devuelve esa solucion. Si no lo es, ejecuta recursivamente el algoritmo agregando el siguiente nodo, y lo ejecuta nuevamente sin agregar el nodo, devolviendo el menor de los dos resultados. No tendria sentido volver a realizar la ejecucion si ya es recubrimiento, ya que de esa manera daria una solucion tambien valida, pero con un nodo mas, o identica a la realizada. Ahi se podan muchos casos y reduce el tiempo de ejecucion considerablemente.

Como tambien se realiza en las heuristicas siguientes, antes de ejecutar el algoritmo recursivo por primera vez, se le pasa una lista de nodos donde fueron excluidos los aislados, optimizacion que tambien reduce la cantidad de iteraciones realizadas.

3.3. Pseudocódigos

Exacto: Algoritmo Exacto para resolver Vertex Cover $\longrightarrow O(2^n)$

```
1: Sacar nodos aislados de g
2: Ejecutar exactoRecursivo con nodos, solucion vacia y solucion minima igual a
   los nodos
```

ExactoRecursivo: Algoritmo ExactoRecursivo $\longrightarrow O(2^n)$

```
1: if Es recubrimiento la solucion? then
2:     if La solucion es menor a la minima? then
3:         Devolver la solucion
4:     else
5:         Devolver la minima
6:     end if
7: else if Quedan nodos? then
8:     Devolver la menor solucion de ejecutar el ExactoRecursivo agregandole un
       nodo o sin agregar
9: else
10:    Devolver minima
11: end if
```

Es Recubrimiento?: Devuelve si la solucion ingresada es recubrimiento del grafo o no $\longrightarrow O(m)$

```
1: Para cada eje en G
2: if si la solucion no tiene a ninguno de los dos ejes then
3:     devuelvo FALSE
4: end if
5: devuelvo TRUE
```

3.4. Análisis de complejidad

La complejidad del exacto es la siguiente. Al comienzo del algoritmo, chequea que la solucion que le entra si es recubrimiento. Esto es $O(m)$. Luego, realiza 2^n recursiones, devolviendo cada vez el menor recubrimiento logrado. Este se consigue realizando dos veces la recursion, una vez con un nodo mas y otro sin ese nodo, sacando el mismo de la lista de nodos. Tomando un rbol de decisin, cada nodo es un nivel del rbol, y sus dos hijos son la llamada recursiva a la funcin agregando este nodo y la llamada recursiva sin agregar el nodo. Por lo tanto, tenemos un rbol binario completo de n niveles, teniendo 2^n nodos, donde cada nodo es una llamada a la funcin ExactoRecursivo. Con n tamao del problema, la frmula para la complejidad recursiva queda de la siguiente manera:

$T(n) = 2 * T(n-1) + c = 4 * T(n-2) + 3 * c = ... = 2^{n-1} * T(1) + (2^{n-1} - 1) * c = O(2^n + 2^n * c)$

c es el costo que tiene cada llamada recursiva, el cual es m. Las llamadas son a lo sumo $O(d)$, siendo d el grado de un nodo. Esto se puede acotar por n, ya que el grado de un nodo no puede ser mayor a la cantidad de nodos del grafo.

En conclusin, la complejidad de la funcin es $O(2^n + m * 2^n) = O(m * 2^n)$, donde n es la cantidad de nodos del grfico original.

3.5. Resultados

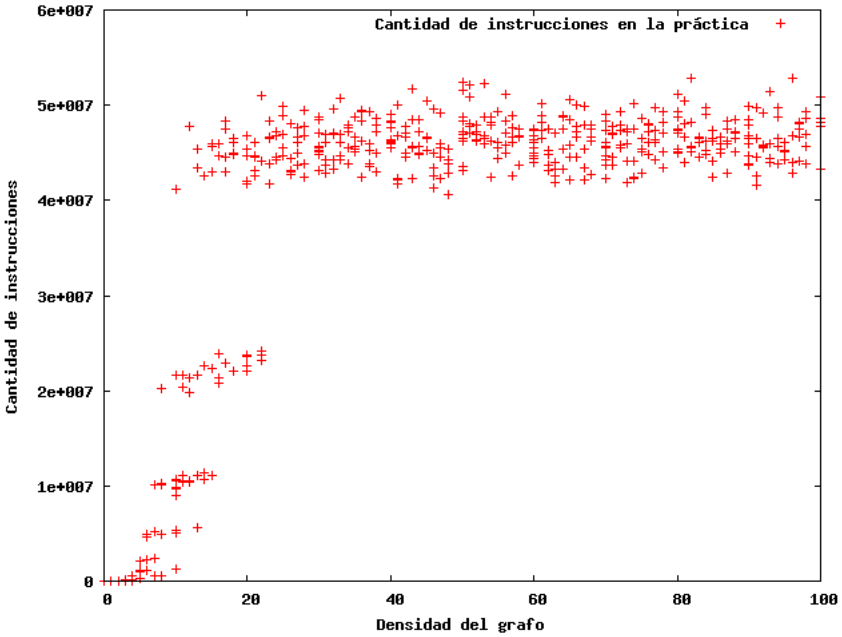

Figura 2: Cantidad de instrucciones en funcion a la densidad del grafo

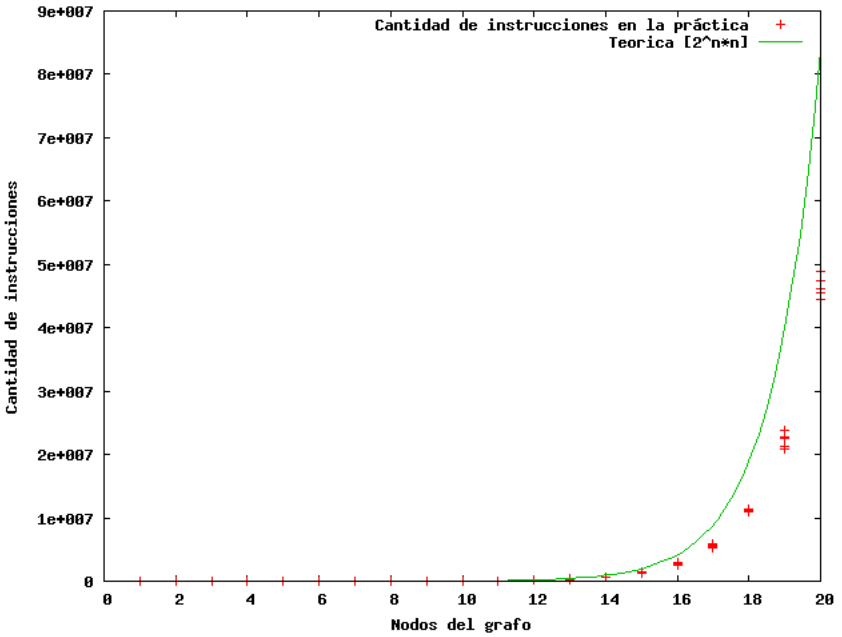

Figura 3: Cantidad de instrucciones en funcion de la cantidad de nodos

3.6. Discusión

En la figura 1 se ve la cantidad de instrucciones realizadas por el algoritmo en funcion de la densidad del grafo. Las pruebas realizadas fueron sobre grafos de 20 nodos y variando la densidad entre 0 y 100. Para comentar, esta prueba tardo 1 hora 10 minutos en una maquina con un Intel Core 2 Duo 1.86Ghz con 2 GB RAM DDR2 1000Mhz. Se tomaron 5 muestras por cada variacion de densidad para tener un resultado un poco mas acertado. A primera vista se observa como la cantidad de instrucciones es muy grande y tiende a seguir subiendo, recien estabilizandose cuando la densidad va llegando a 100 %. Un comentario sobre este grafico: Si bien se ve que en el caso que son completos los grafos, y se tomaron 5 muestras, estos son iguales observacionalmente, no estructuralmente. O sea que la lista de ejes que llega desde el parametro de entrada no siempre esta en el mismo orden y eso varia la cantidad de instrucciones a realizar.

En la figura 2, el grafico muestra la cantidad de instrucciones realizadas, pero en funcion de los nodos del grafo. Aca se ve claramente que la complejidad se cumple, ya que es exponencial y crece rapidamente. Las pruebas fueron realizadas fijando la densidad en 50 % y variando los nodos entre 0 y 20. No se utilizaron mas nodos, ya que la prueba tardaria mas de 3 horas.

3.7. Conclusiones

Las conclusiones son bastante claras. El algoritmo se comporto como se esperaba, tardando mucho en grafos medianamente chico para ejecutar. Se ve claramente como el algoritmo no es recomendable para resolver este problema, aunque se quiera una solucion precisa, ya que demoraria meses, aos o siglos en tener la misma. En la carpeta bib se adjunta un pdf con una solucion que baja la complejidad a $O(1{,}87^n)$ aproximadamente, aunque en grafos muy especificos.

4. Goloso

4.1. Introducción

Un algoritmo goloso es aquel que soluciona un problema mediante la busqueda del optimo local en cada etapa de ejecucion esperando tener al optimo global del problema.

El algoritmo goloso consta de varias partes:

- Un conjunto de candidatos, desde donde se va a formar la solucion
- Una funcion de seleccion, que elige al mejor candidato para agregar a la solucion
- Una funcion de fiabilidad, que dice si el candidato contribuye a la solucion
- Una funcion objetivo, que le asigna un valor a la solucion o a la parcial
- Una funcion solucion, que dice si se llego a una solucion completa

Los problemas para los cuales el algoritmo goloso funciona mejor son aquellos que cumplen con las siguientes propiedades:

- Greedy Choice Property

  Esto quiere decir que el problema permite que podamos buscar siempre una solucion optima local, por su naturaleza. Este algoritmo iterativamente crea una solucion golosa despues de la otra, haciendo que el problema se reduzca. Un algoritmo goloso nunca reconsidera sus elecciones.

- Optimal Substructure

  Una subestructura optima existe si una optima solucion al problema contiene optimas soluciones a sus sub-problemas.

4.2. Desarrollo

Esta heuristica fue implementada de la siguiente manera:

Primero se definio la funcion de seleccion. Esta selecciona al nodo que tenga mayor cantidad de vecinos en el grafo. Para realizar esto, la funcion NodoMayorGrado, itera sobre la lista de nodos, preguntando en cada paso si la lista de vecinos es mas grande que la de la iteracion anterior. Luego, devuelve el nodo elegido.

La funcion de objetivo es EsRecubrimiento. Esta verifica que para cada eje, por lo menos uno de los extremos este en la solucion.

Antes de iniciar el ciclo, se sacan los aislados del grafo, ya que estos no modifican la solucion final, debido a que no tienen vecinos, y esto haria que las iteraciones sean mas, influyendo en la performance del algoritmo. Los aislados se sacan en la funcion SacarAislados, que recibe como parametro el grafo y la cantidad de nodos y guarda en una lista de Nodos, aquellos que si tienen vecinos.

Ya en el ciclo, luego de elegir el nodo de mayor grado, este se agrega a la solucion, y se saca de la lista de nodos.

Una vez cumplida la funcion objetivo, se devuelve la solucion como un objeto de la clase Recubrimiento, implementada para mejor manejo de los datos para las pruebas.

Una optimizacion que se encontro investigando sobre el tema, es tambien, en la funcion de seleccion, hacer una resta entre la cantidad de vecinos del nodo y los ejes que este nodo cubriria que ya estan en la solucion. Esto reduciria la cantidad de iteraciones y mejoraria la solucion, pero se decidio no implementarlo ya que se habian realizado ya todas las pruebas y tomaria mucho tiempo realizar todos los graficos y corridas otra vez, mas que nada las que comparan con el algoritmo Exacto.

4.3. Pseudocódigos

Goloso: Algoritmo Goloso para resolver Vertex Cover $\longrightarrow O(n^2)$

```
1: Sacar nodos aislados de g
2: while EsRecubrimiento(solucion) do
3:     Buscar nodo con mayor cantidad de vecinos
4:     Agregar el nodo a la solucion
5:     Sacar el nodo de la lista de nodos
6: end while
```

4.4. Análisis de complejidad

La complejidad del algoritmo goloso es bastante simple. Este algoritmo lo primero que realiza es sacar los aislados del grafo, estos son los nodos que no tienen vecinos. Por consiguiente, estos nodos no modifican al vertex cover ya que no recubren ningun eje. Esto lo realiza en $O(n)$, recorriendo toda la lista de nodos y sacando los que no se van a usar. Luego, lo que realiza el algoritmo es un ciclo en donde se le va agregando a la solucion el nodo que mayor cantidad de vecinos tenga. Este ciclo se repite hasta que la solucion ya es un recubrimiento. Este ciclo se repite, en peor caso, n veces. Dentro del ciclo, la funcion que busca al nodo con mas vecino, tiene complejidad $O(n)$ y la funcion que verifica si es recubrimiento es $O(m)$, ya que itera sobre la lista de ejes. Las demas operaciones son en $O(1)$. La complejidad resultante es de $O(n(n+m)+n) \rightarrow O(n^2 + n*m + n) \rightarrow O(n^3)$ en peor caso, acotando a m por n^2.

4.5. Resultados

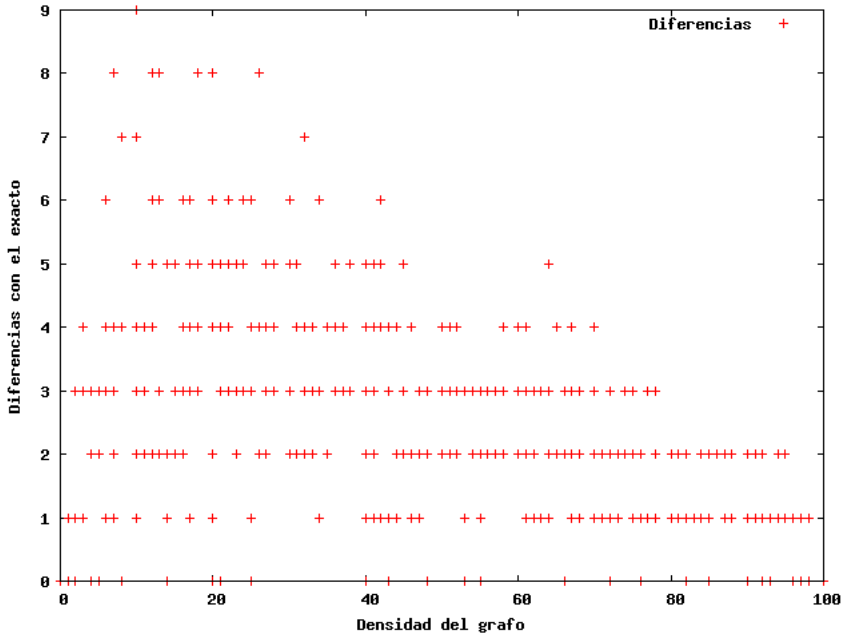

Figura 4: Diferencias con el exacto, en la densidad del grafo

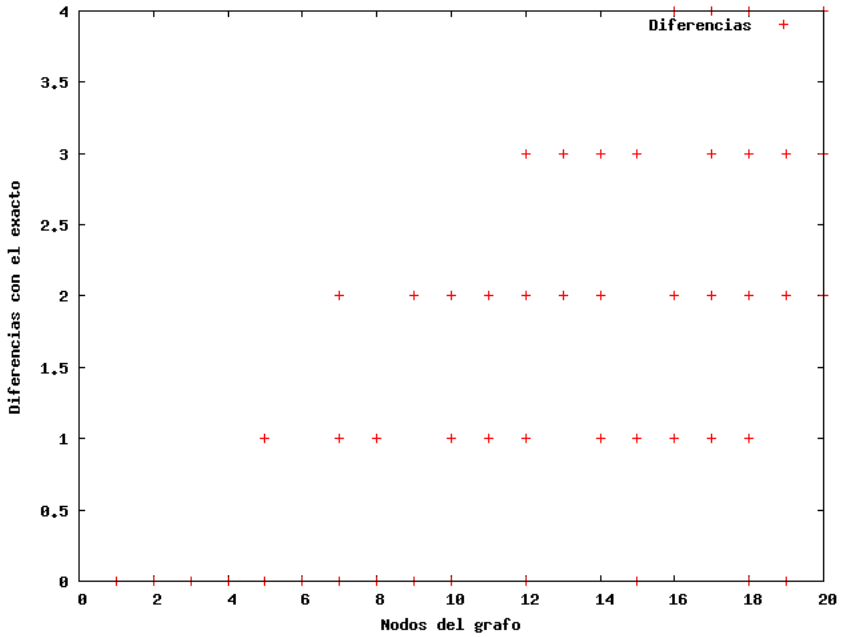

Figura 5: Diferencias con el exacto, en la cantidad de nodos

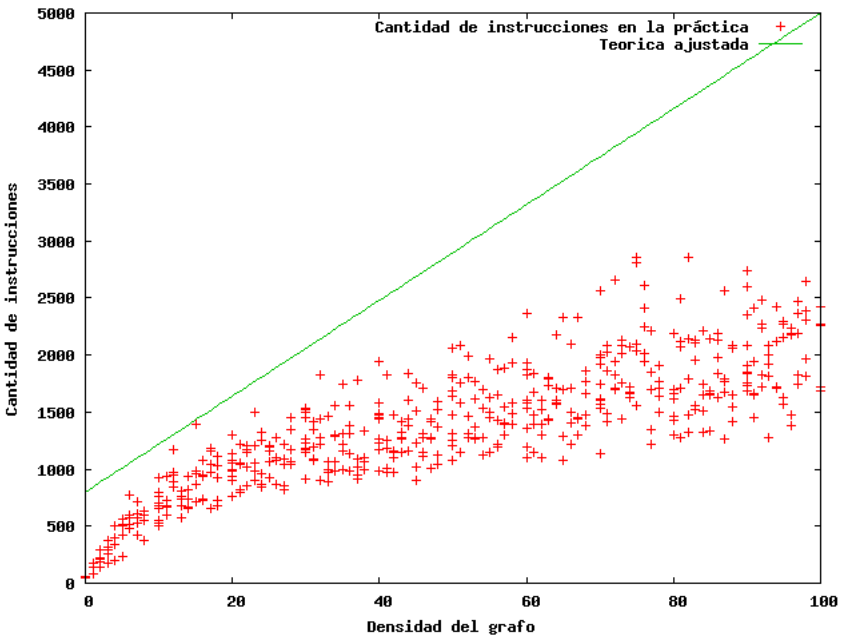

Figura 6: Cantidad de instrucciones, en la densidad del grafo

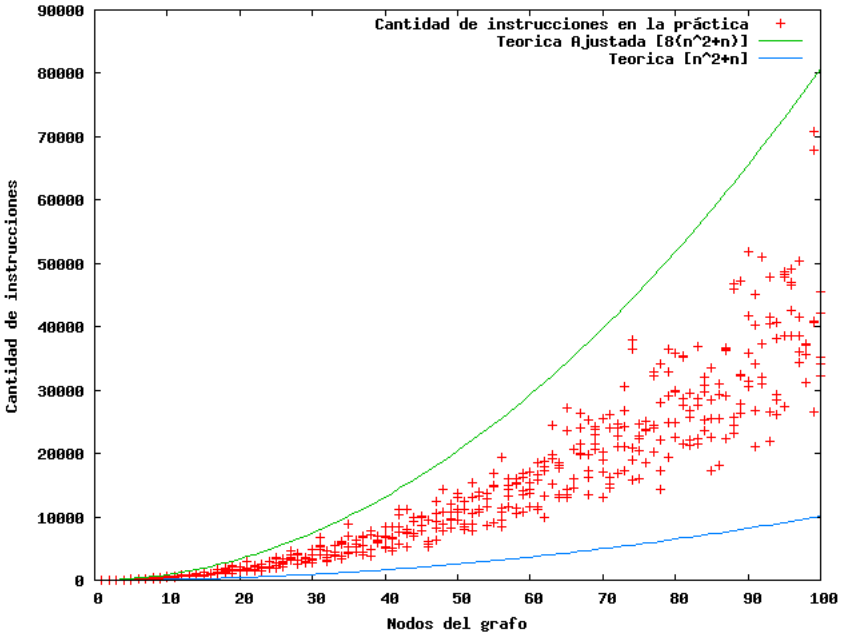

Figura 7: Cantidad de instrucciones, en la cantidad de nodos

4.6. Discusión

En la figura 1, estan graficadas las diferencias entre el algoritmo exacto y el goloso. Se observa un patron que, mientras mayor es la densidad del grafo, menor es la diferencia entre ambos. Esto se debe, por ejemplo en el grafo K_n, el goloso va agarrando al nodo de mayor grado, como es completo, todos tienen el mismo grado, o sea que va agarrando de a uno. Al llegar al ultimo, ya le queda recubierto, porque los k ejes de ese ultimo nodo estaban conectados con los otros n-1 nodos. Esto es lo mismo que sucede con el exacto, por eso la diferencia es minima.

Esta prueba se corrio para 500 grafos de 20 nodos con densidades del 1 % al 100 %. Tambien se observa que la media esta entre 2 y 4 diferencias, y mientras mas se acerca al 100 %, las diferencias son menores.

En la figura 2, estan graficadas las diferencias, nuevamente, entre el exacto y el goloso, pero en funcion de la cantidad de nodos del grafo. Se grafico fijando la densidad en 50 % y aumentando los nodos de 1 a 20. En este se observa, que mientras mas chica es la cantidad de nodos, mejores resultados obtiene el goloso. Las diferencias van entre 1 y 4 nodos entre los resultados.

Entre las dos primeras figuras, se puede tomar que, con pocos nodos y una densidad grande, se obtienen mejores resultados con la heuristica Golosa.

En la figura 3, estan graficadas las instrucciones realizadas por el algoritmo en funcion de la densidad del grafo. Se observa que los resultados se esparsen cuando la densidad aumenta. La complejidad en este caso, fijando el valor de n en 20, seria $20^2 + 20 * m + m$. Se ve que cumple con la cota de complejidad en peor caso.

En la figura 4 esta graficada la cantidad de instrucciones realizadas por el algoritmo en funcion a la cantidad de nodos del grafo. La complejidad teorica calculada es de n^3, se ve que la cota es buena y cumple la complejidad practica (los titulos del grafico estan equivocados, el de la teorica tendria que decir n^3, pero en la generacion del grafico si se uso ese valor).

Aclaracion: Se tomo densidad 50 % porque se hicieron mediciones con otras densidades y no variaba el grafico. Se dejo en 50 % por un tema de performance de la corrida, para que no se haga muy largo el tiempo de ejecucion.

4.7. Conclusiones

Como conclusiones finales, se puede analizar que:

El algoritmo, si bien responde en un tiempo razonable, no siempre da una solucion optima. Esto se cumple en un pequeo porcentaje de los casos. Estas pruebas fueron hechas con grafos aleatorios, tanto variando la cantidad de los nodos, como la densidad del mismo. Como conclusion final, se puede observar que, se comporta bien en casos donde los nodos sean pocos y la densidad sea grande, dando los mejores resultados en esa combinacion.

5. Búsqueda Local

5.1. Introducción

Los algoritmos de busqueda local se caracterizan por definir, dada una posible solución del problema, soluciones *vecinas*. Dos soluciones se consideran vecinas si ellas son *parecidas* bajo algún critero. Definida la función de vecindad, el algoritmo lo que hace es generar una solucion inicial (ya sea utilizando otras heurísticas, técnicas algorítmicas o soluciones *ad-hoc*) y proseguir a enumerar sus vecinos. De allí, y bajo algún criterio, se selecciona alguna que se considere *mejor* a fines del problema a resolver. Para ello se define la función objetivo. Ésta asigna un valor numérico a cada posible solución de forma tal que, si una solución s es mejor que otra t bajo el criterio que se está utilizando, entonces s deberá tener menor función objetivo (el resultado de aplicar dicha función a s sera menor que áquel donde se aplique a t). Por lo tanto, el algoritmo podrá elegir fácilmente un vecino mejor simplemente evaluando ésta función. Notar que el criterio de decisión también deberá incluir alguna forma de elegir una sola solución si es que dos o más soluciones vecinas poseen la menor función objetivo. Luego se reinicia el procedimiento pero tomando ésta vez como solución inicial a la elegida anteriormente como mejor vecina. El algoritmo finaliza cuando no existe una mejor solución que la actual.

Notar que la heurística (tal y como por definición de heurística debería ser) no es exacta, es decir que no brinda (necesariamente) la mejor solución al problema. Ésto sucede cuando el algoritmo encuentra un mínimo local de la función objetivo en lugar de encontrar el mínimo global. Por ello es que es importante elegir una buena solución inicial (una que se encuentre alejada de mínimos locales) y un buen criterio de selección de mejores vecinos. Pero, obviamente, ésto es difícil de conseguir en la práctica.

A pesar de la inseguridad de la solución que brinda, la heurística puede resultar muy útil en comparación con un algoritmo exacto si se utiliza en problemas cuya solución más rápida conocida hasta el momento es exponencial. Y se debe principalmente a la complejidad polinomial que posee y a la existencia de parámetros, lo cual permite ajustar el algoritmo a las necesidades de cada contexto de uso.

5.2. Desarrollo

5.2.1. El algoritmo

Con lo primero que nos cruzamos a la hora de hacer resolver el problema del recubrimieto de ejes mediante ésta heurística fue con la función de vecindad. Y luego de pensarlo bien, notamos que un algoritmo de búsqueda local es principalmente eso, definir una función de vecindad. Ésto se debe a que la función objetivo del problema es muy sencilla: dadas dos soluciones (recubrimientos de ejes), la mejor será aquella con menor cantidad de nodos. Pero ésta tenía que ser elaborada con ciertos cuidados:

- Debía permitir que cualquier solución pueda ser alcanzada mediante cualquier otra al aplicar finitas veces la función de vecindad, ya que de no ser así podíamos caer en no conseguir cierta solución si partíamos de cierta otra inicial. Y como ésta podria haber sido la óptima, la heurística no hubiese sido exitosa.
- No debería devolver una cantidad enorme de vecinas. Y con enorme queremos referirnos a cantidades exponenciales en función de la cantidad de nodos/ejes del grafo. De haber sido así, el aplicar la función objetivo a cada una hubiese costado tiempo exponencial, con lo que la heurística hubiese perdido su razón de existencia que es poseer complejidad polinomial.

Con ésto en mente, la solución que se optó fue la siguiente[1]:

Se recibe la solución inicial/anterior s junto con parámetros de cuantos nodos se sacan de la solución y cuántos se agregan (cs y ca respectivamente). Primero se prosigue a quitar nodos del recubrimiento actual. Para ello se itera nodo por nodo y, para cada uno, se prosigue de la siguiente forma (supongo que el actual es el i-ésimo de la lista):

- La primer solución vecina será la de tomar los nodos restantes consecutivos a éste nodo (es decir, los de las posiciones $i, i+1, i+2, \ldots$).
- La segunda solución vecina será aquella que incluya al nodo actual, a aquel en 2 posiciones siguientes, 4 posiciones, 6 posiciones, etc. Es decir, se tomarán los índices $i, i+2, i+4, \ldots$. Notar que se saltea 1 nodo en la elección, por eso llamamos que aquí se eligió con un $offset = 1$.
- La tercer solución será $i, i+3, i+6, \ldots$, es decir con un $offset = 2$.

El procedimiento se continúa hasta el $offset$ más grande posible. Cada posible solución hallada de ésta forma es quitada y verificada si es un recubrimiento válido. De serlo, se toma como mejor vecino y se devuelve. De no ser asi, se prosigue a insertarle nuevos nodos.

La inserción sigue el mismo proceso de selección pero ésta vez para los nodos restantes que no estaban en el recubrimiento pasado. Se los agrega y se verifica que el resultado sea efectivamente un recubrimiento. De serlo, y como $cs < ca$, entonces se toma como mejor recubrimiento al actual y se devuelve.

[1] Para una mayor comprensión del lector, dirigirse a la sección de pseudocódigos

Si no se consigue ningún vecino mejor después de verificar todas las vecindades, se toma a s como la mejor solución alcanzada, y por ende como la solución de la heurística.

Notar que ambas funciones, la que lista vecinos y la que elige alguno de ellos, están implementadas juntas. Ésto es para evitar tener que listar todos los vecinos e ir calculando un vecino simplemente cuando se lo necesite.

Existe una aclaración que nos gustaría mencionar. Debido a la representación de un grafo elegida por nosotros, dos grafos pueden ser observacionalmente iguales (o mejor dicho, isomorfos) pero aún así obtener diferentes resultados al correr éste algoritmo. Ésto ocurre porque dos grafos isomorfos no necesariamente guardan los valores de sus listas en el mismo orden, pero éste orden es el que rige que vecinos se recorrerán y cuales no. Asique puede suceder que para dos grafos isomorfos las vecindades obtenidas sean diferentes y por lo tanto se obtengan recubrimientos diferentes y cantidad de instrucciones ejecutadas distintas.

Como solución inicial se eligió una bastante sencilla, con poca *inteligencia* y que devuelve un recubrimiento en la mayoría de as veces grande. Consistió en recorrer todos los ejes y verificar si en la solucion a devolver no existía ya alguno de sus nodos. De no ser así, se agregaba cualquiera. Notar que ésta implementación se adapta bien al funcionamiento del algoritmo siguiente ya que al generar una solución con muchos nodos dá mayor libertad para luego sacar algunos y agregar nuevos. Ésta fue la solución menos costosa que se nos ocurrió y la dejamos justamente porque eso es lo que queríamos, preferimos dejarle el calculo más árduo a la búsqueda local.

Más allá de los parámetros que toma la heurística ya mencionados anteriormente, cuantos nodos se sacan y cuantos se agregan para el cálculo de vecinos, se nos ocurrieron otros más de menor importancia:

- Mezclar la solución antes de buscarle vecinos? Un posible problema con el que nos encontramos fue que una solución esta condenada a elegir siempre los mismos vecinos, todo dependiendo del orden en el cual le fueron agregados los nodos a su lista. Por eso se podría haber implementado este bit que indica si se debía mezclar la lista o no para ofrecer un mayor rango de alcance. Pero ésto podía generar que a algunas soluciones sea difícil de llegar debido a que siempre la mezcla dá de cierta forma. Además, se implementó ésta mezcla y en la práctica no causaba mejoras sino que encima, a veces, empeoraba. Por ello se decidió no agregar éste parámetro.

- Utilizar solución golosa? La idea de utilizar la solución ad-hoc descripta más arriba nos pareció desde el principio, como ya se dijo, bastante adecuada para la mecánica del algoritmo. Pero también se nos ocurrió el utilizar la heurística golosa ya desarrollada anteriormente. Se realizaron algunas pruebas pero no fueron satisfactorias. Principalmente, se ejecutaron mayor cantidad de operaciones (lo cual comparando las complejidades teóricas es de esperar) y los recubrimientos obtenidos no fueron mejores. Por lo tanto, se dejó como solución inicial la ya mencionada anteriormente[2].

[2]Para mas información, dirigirse a la sección de resultados.

5.2.2. Las pruebas

Dado que habían varias variables a evaluar en las pruebas, diseñamos un sistema de puntajes que nos permita elegir fácilmente el mejor par de parámetros (porcentaje de nodos que agrego y que saco) para el algoritmo así luego se utilizaban éstos para las demás pruebas. Decidimos correr todas las posibilidades de parámetros posibles (cuantos saco de 1 a 100 y cuantos agrego de 0 a cuantos saco) y dar un puntaje a cada parámetro según el tamaño del recubrimiento que devolvió. A los que sacan el más chico, 5 puntos. A los segundos 3 y a los terceros 1. Preferimos que sea así y no simplemente contando cuantas veces salio primero cada uno porque podría suceder que, por ejemplo, un par salga siempre segundo cuando los demás pelean parejo el primer puesto, y es muy probable que en un caso así sea mejor el comportamiento del segundo. Además quisimos que los puntajes no estén tan cerca como asigando 3, 2 y 1 punto a los tres puesto ya que preferimos darle más importancia al que llega a ser primero. Por eso distanciamos en 2 puntos los premios.

Éstas pruebas decidimos hacerlas sobre familias de grafos aleatorios, puesto que no queríamos ningún comportamiento particular que pueda favorecer a algun par sin que nosotros pudiésemos darnos cuenta. Además, puesto que la calidad de la solución es regida principalmente por la cantidad de ejes (ya que cuantos hallan definirán el tamaño de la solución) más que la cantidad de nodos (influyen más en la complejidad), decidimos que las pruebas para elegir el mejor parámetro sea con grafos con una cantidad fija de nodos y la cantidad de ejes, variable.

Una vez obtenido el mejor par de parámetros, lo consideramos como el mejor para cualquier instancia (a modo de simplificación de las pruebas) dado que se basó en grafos aleatorios, por lo que proseguimos a calcular la cantidad de instrucciones que ejecuta y a comparar las diferencias en los recubrimientos que genera en comparación con los del algoritmo exacto (en ambos casos, quisimos estar seguros y generar las pruebas con grafos que varíen su densidad y luego grafos que varíen su cantidad de nodos).

5.3. Pseudocódigos

BusquedaLocal: Algoritmo de Búsqueda Local para resolver Vertex Cover $\longrightarrow$ $O(n^3m)$

```
1: s ← Generar solución inicial
2: while Halla mejor vecino de s do
3:     s ← Mejor vecino
4: end while
5: return s
```

SolucionInicial: Construye una solución incial *naive* $\longrightarrow$ $O(mn)$

```
1: sol ← φ
2: for Cada eje e del grafo do
3:     if No hay ningún extremo de e en sol then
4:         Agregar el primer extremo a sol
5:     end if
6: end for
7: return sol
```

MejorVecino: Devuelve un mejor vecino a la solución dada o indica que no existe mejor de ser así $\longrightarrow$ $O(n^2m)$

```
1: s es la solucion actual
2: t ← logitud de s
3: cs ← cuantos nodos saco
4: ca ← cuantos nodos agrego
5: for cada nodo de s do
6:     for cada offset posible que genere una sublista de longitud cs de s do
7:         if sacando esta sublista de s sigue siendo recubrimiento then
8:             return s - sublista
9:         else
10:            for cada nodo en la lista de nodos que no estaban en la solucion
               do
11:                for cada offset posible que genere una sublista de longitud ca
                   de nodos que no estaban en la solucion do
12:                    if agregando ca nodos a s - sublista sigue siendo un re-
                       cubrimiento then
13:                        return s - sublista + nuevos nodos
14:                    end if
15:                end for
16:            end for
17:        end if
18:    end for
19: end for
20: return No hay mejor vecino
```

5.4. Análisis de complejidad

5.4.1. SolucionInicial

Éste algoritmo recorre todos los ejes (m veces) y para cada uno verifica si existe alguno de sus extremos en la posible solución (como puede llegar a ser todos los nodos, n). Por lo tanto la complejidad es $O(nm)$.

5.4.2. MejorVecino

La primer acción realizada por el algoritmo que no tiene complejidad constante es la selección de los nodos a agregar. Ésta es del orden de n (cantidad de nodos del grafo) en el peor caso ya que debe ir comparando para cada uno de los nodos si está o no en la solución.

Luego viene el bucle exterior, que va recorriendo cada nodo de la solución. Por lo que se ejecuta t veces, lo cual equivale a $\mathrm{O}(n)$ ya que n es el máximo valor posible para t.

El segundo bucle se ejecuta tantas veces como *offsets* posibles hallan. Ésta cantidad esta dominada por la ecuación

$$(cs - 1) * o + cs \leq t - i$$

considerando que o es el offset e i es la posición del nodo actual. Por lo tanto el offset máximo om será

$$om = (t - i - cs)/(cs - 1)$$

redondeado para abajo. Notar que éste valor será mayor cuando se esté al comienzo de la lista ($i = 0$), y cuando la cantidad de nodos que saco también sea mínima ($cs = 2$, porque de ser 1 no existiría el offset ya que se toma solo a i) y cuando el tamaño de la solución de entrada sea máxima ($t = n$). Por lo que se ejecutará $\mathrm{O}(n)$ veces.

El tercer y cuarto bucle son similares a los anteriores, y ya que la lista de nodos a poner puede también ser n, las complejidades son las mismas.

Las verificaciones de si es o no un recubrimiento son $\mathrm{O}(m)$ y devolverlos se realizan en tiempo constantes, por lo que se considera a todas las operaciones de los bucles en $\mathrm{O}(m)$.

Por lo tanto, la complejidad total del algoritmo es $\mathrm{O}(n^4 m)$

5.5. BusquedaLocal

El algoritmo lo primero que realiza es generar la solución *naive*, lo cual le cuesta $O(nm)$. Luego ejecuta MejorVecino hasta encontrar una mejor solución. Dado que el bucle se ejecuta si hay una mejora en la solución, y que ésta no puede tener mas

de n nodos, el caso en donde más veces iterará será cuando la solución naive arroje un resultado de longitud n y en cada iteración se disminuya la respuesta en uno, hasta alcanzar el mínimo en un conjunto con un sólo nodo. En éste caso se harían n iteraciones, por lo que la complejidad sería

$$O(nm + n(n^4m)) = O(n^5m)$$

5.6. Resultados

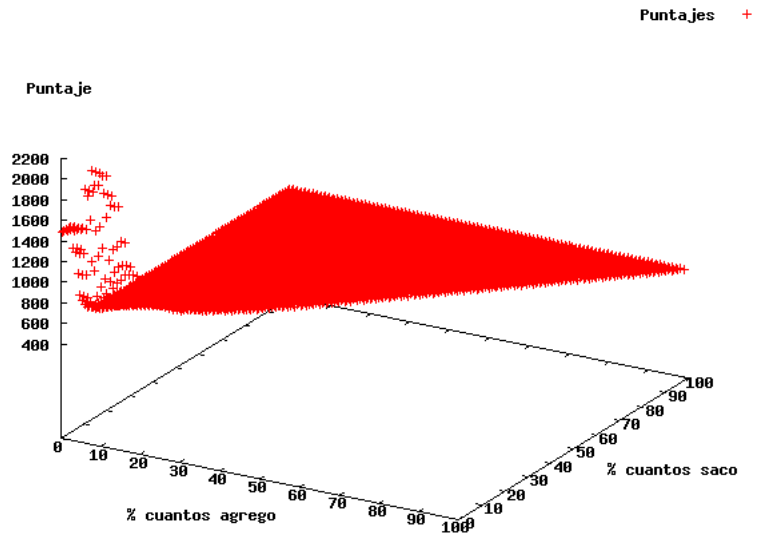

Figura 8: Puntaje de cada par de parámetros en función de ambos valores. Evaluado sobre grafos aleatorios de 40 nodos y densidad variable de 0 % a 100 %

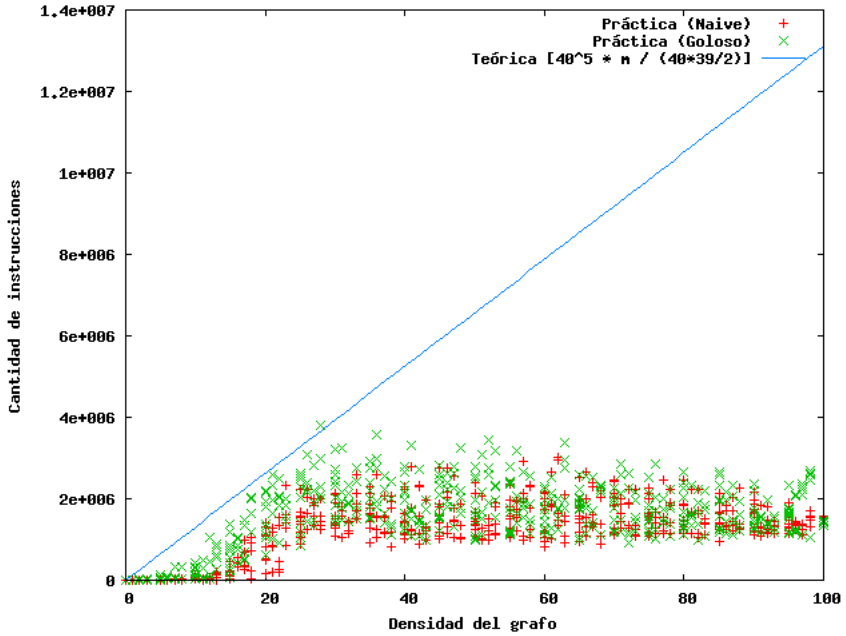

Figura 9: Cantidad de instrucciones con el mejor parámetro encontrado en función de la densidad de los grafos. Evaluado sobre grafos aleatorios de 40 nodos y densidad variable de 0 % a 100 %.

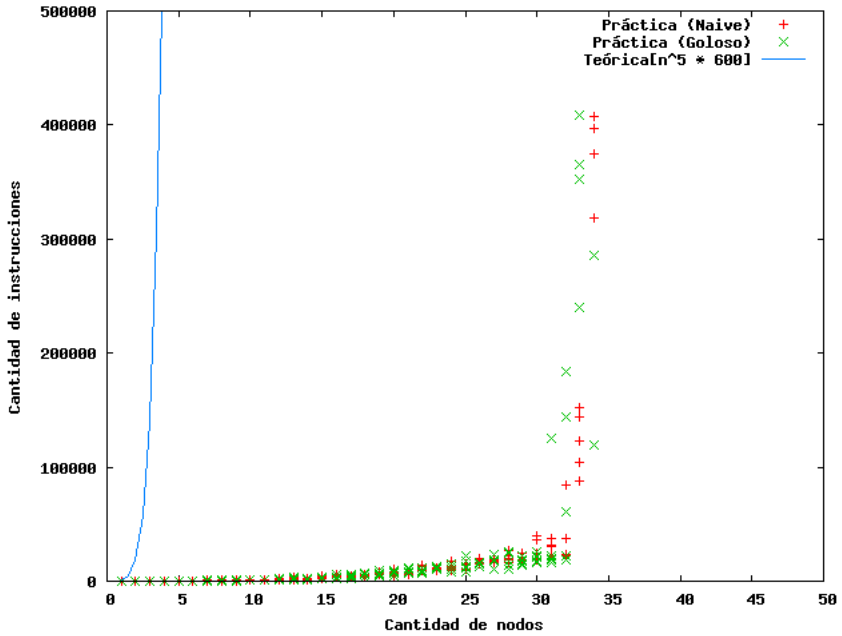

Figura 10: Cantidad de instrucciones con el mejor parámetro encontrado en función de la cantidad de nodos de los grafos. Evaluado sobre grafos aleatorios con 50 % de densidad y cantidad de nodos variable entre 1 y 50

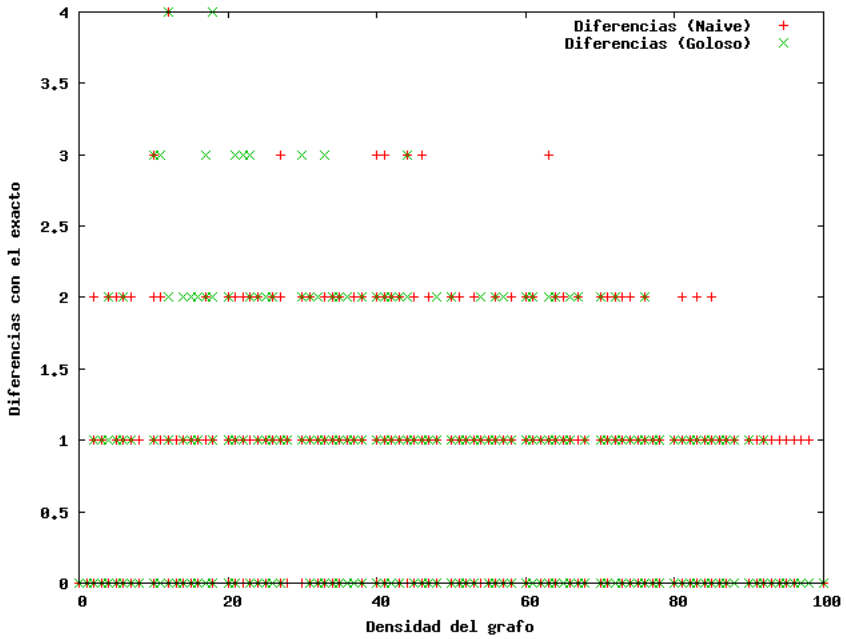

Figura 11: Cantidad de diferencias en comparación con el algoritmo exacto en función de la densidad de los grafos. Evaluado sobre grafos aleatorios de 20 nodos y densidad variable de 0 % a 100 %

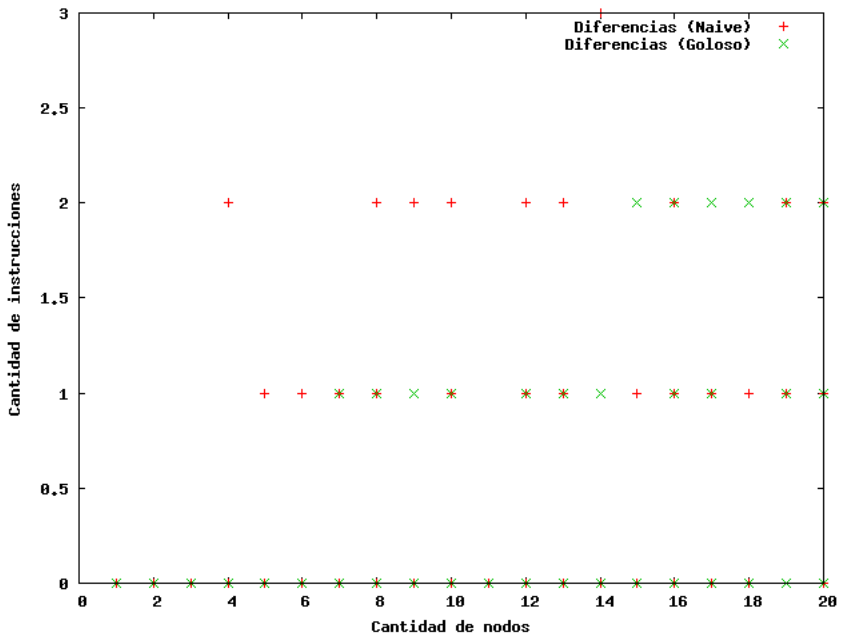

Figura 12: Cantidad de diferencias en comparación con el algoritmo exacto en función de la cantidad de nodos de los grafos. Evaluado sobre grafos aleatorios con 50 % de densidad y cantidad de nodos variable entre 1 y 20

5.7. Discusión

Como se puede ver en el primer gráfico, a partir de cierto punto, mientras ambos parámetros aumentan el puntaje disminuye drásticamente. Pero ésta acumulación de la importancia cerca de los parámetros no nos sorprendió. Es de esperar que, al sacar y agregar pequeñas cantidades de nodos, se obtenga mayor flexibilidad para generar vecinos y de esa forma conseguir un mayor espectro de posibles vecinos con los cuales probar. Lo que sí nos sorprendió fue que las mismas pruebas fueron corridas con las pruebas utilizando al algoritmo goloso como solución inicial y el resultado fue el mismo: se tendía a mejorar el puntaje con parámetros cercanos al (4,7). Creemos que se debe a que a nuestro algoritmo goloso no lo quisimos hacer minimal, por lo que en cierta medida puede generar soluciones *similares* al algoritmo *naive*, ya que el último brinda recubrimientos de gran tamaño.

Los mejores parámetros fueron[3]: (4,7), (5,7) y (6,7), leyéndose éstas tuplas como (porcentaje cuantos agrego, porcentaje cuantos saco). Preferimos tomar al menor de ellos como el mejor, y la decisión se basó principalmente en lo dicho anteriormente: nos parece que menores valores brindan flexibilidad a las soluciones, por lo que podrían comportarse mejor. Además, considerando el rango posible de valores, podemos apreciar que valores pequeños son realmente los ganadores y los mayores generan una meseta en el gráfico de puntajes. Por ende, el (4,7) es nuestro mejor par.

En el segundo y tercer gráfico ya utilizamos el parámetro calculado anteriormente para averiguar la cantidad de instrucciones que cuesta cada ejecución utilizando como solución inicial la *naive* y la golosa.

El primero de los dos utiliza grafos todos con la misma cantidad de nodos pero densidad (porcentaje de la cantidad total de posibles ejes) variable. Notar que m fue dividido por la cantidad máxima de ejes posibles $\left(\frac{n(n-1)}{2}\right)$ siendo en éste caso $n = 40$) del grafo para obtener la densidad y así ser posible plasmar los valores en un gráfico que se encuentra en función de la densidad. Como se puede apreciar, la cota de la complejidad es válida. Pero además se puede apreciar otro hecho importante: la cantidad de instrucciones totales aumenta considerablemente cuando se elige como solución inicial a la golosa por sobre la *naive*. Y era de esperar debido a que la complejidad del primero es mayor que la del segundo.

El segundo de los gráficos de instrucciones fija la densidad de los grafos (50 %) y varía la cantidad de nodos. La curva teórica dibujada se baso en que m sea el mayor valor posible en el grafo (es decir, cuando la cantidad de nodos es máxima: $50 * 49/2 \approx 600$) y es por eso que nos pareció, en principio, tan exagerada la cota. Por otro lado, la complejidad teórica está basada en casos que son prácticamente imposibles en la realidad. Por ejemplo, las n llamadas a MejorVecino explicado en la complejidad del algoritmo principal, BusquedaLocal (entre otros). Igualmente no nos preocupa porque en el peor caso es verdad que puede llegar a suceder, asique no se la puede considerar una "mala cota". Con respecto a la comparacion entre las soluciones iniciales, aquí se refleja nuevamente el costo superior de la solución golosa por sobre el de la *naive*.

Los dos últimos gráficos intentan mostrar la bondad de las soluciones de la

[3]Para mayor información sobre los mejores puntajes, dirigirse al apéndice correspondiente.

búsqueda local con ambas soluciones iniciales, basándonos nuevamente en el mejor par obtenido anteriormente. La primer comparación nos pareció que arrojó resultados favorables ya que, en promedio, la mayor diferencia en el recubrimiento es de uno o nungún nodo. Notar que a medida que aumenta la densidad del grafo, los errores disminuyen. Sin embargo, en el que realiza comparaciones en función de la cantidad de nodos no se encuentra patrón alguno (por lo menos para la cantidad de muestras que pudimos computar). Ésto nos parece extraño pero la única explicación disponible es que trabajamos con grafos aleatorios, y si quisiésemos encontrar detalles a éste nivel deberíamos analizar grafo por grafo.

Tampoco fueron encontrados patrones respecto a la comparación entre ambas soluciones iniciales ya que mantienen valores similares. Se podría decir que el goloso tiene una cantidad un tanto menor de diferencias con respecto a la *naive*, pero los resultados obtenidos no arrojan pruebas contundentes que aseguren que la primera es mejor que la segunda.

5.8. Conclusiones

Para finalizar el análisis de la heurística, hay dos puntos que son los que más nos sorprendieron:

- La mejora en las soluciones que pueden causar las vecindades

  En comparación a otra implementación realizada con anterioridad de la función de vecindad que recorría menos vecinos, los mejores parámetros pasaron de (0,1) a (4,7). Y los puntajes fueron mejores. Creemos que, aunque la complejidad del algoritmo aumentó a raíz de ésta nueva implementación, no deja de ser polinomial y arroja mejores resultados.

- Las pocas diferencias con el algoritmo exacto

  Es verdad que éste es un punto que depende netamente del contexto de uso que se le dará al algoritmo y a la solución que produzca, pero que ande en errores de la décima parte del total nos parece un punto importante a favor de la heurística. Es posible que sólo se dé éste fenómeno en grafos pequeños pero, repitiendo, de haber podido hubiésemos hecho muchas pruebas más.

- La superioridad de la solución inicial *naive*

  En un principio creímos que agregar una solución con mayor inteligencia podría llevarnos a mejores resultados, pero no fue así. El utilizar la solución inicial golosa lo único que causó fue que la cantidad de instrucciones aumente considerablemente y las mejoras no son convincentes. Notar además[4] la cercanía de los mejores parámetros obtenidos en ambos casos. Pero el mejor puntaje del goloso es muy inferior al mejor del *naive*. Por eso creemos que la solución *naive* funciona mejor para nuestro algoritmo.

[4] Ver apéndice de puntajes.

6. GRASP

6.1. Introducción

La heurística GRASP se basa en utilizar en cierta manera las dos utilizadas anteriormente en el presente trabajo. El algoritmo consiste en utilizar Búsqueda Local pero con una solución inicial brindada por un algoritmo goloso con un factor de azar. La solución que brinda se guarda para futuras comparaciones y será actualizado en aquellos casos donde la nueva solución sea mejor que la anterior.

El motivo por el cual se parte de una solución golosa azaroza es porque, al ser GRASP un algoritmo que solamente termina cuando llega a su límite de iteraciones maximas o durante unas iteraciones predeterminadas no hubo cambios. Si la solucion fuese la golosa tradicional, siempre partiria de la misma, y siempre llegaria a lo mismo, terminando el algoritmo en los pasos dichos por la cantidad de iteraciones sin cambios.

Es por esto que se agrega un parametro que determina el porcentaje de permisividad para el goloso. Este lo que hace es, tomando el grado del nodo con mas cantidad de vecinos, utiliza solo ese porcentaje del grado. Por ejemplo, si el grado es 10 y el porcentaje de permisividad es del 40 %, entrarían a la lista de candidatos todos los que tienen 4 o mas vecinos.

Una vez que tenemos esta solución, se le aplica búsqueda local, para mejorar la solución, a partir de parámetros que dicen cuantos nodos agregar y sacar en cada iteracion (mejor explicado en Busqueda Local). Éstos parámetros fueron seleccionados luego de varias pruebas donde se le asignó puntaje a cada par de parámetros, dependiendo del tamaño de la solución. A menor tamaño, más puntaje. Se corrió varias veces con muchos algoritmos y los parámetros utilizados son los que lograron mayor puntaje en las pruebas.

6.2. Desarrollo

6.2.1. GRASP

El desarrollo de GRASP no tuvo demasiados inconvenientes ya que fue muy parecido al de la búsqueda local. Los parámetros que se agregaron fueron el porcentaje que utiliza el el GolosoRandom y la cantidad máxima de iteraciones a ejecutar y la cantidad de iteraciones que deben pasar sin mejoras para detenerse.

Los parámetros de la búsqueda local ya fueron calculados anteriormente (el mejor había sido (4,7)) por lo que decidimos seguir considerándolo el mejor par de parámetros para la búsqueda local. Por supuesto, son todas suposiciones evaluadas luego de analizar algunos casos pequeños, pero de haber sido posible hubiésemos realizado pruebas de todos los valores contra todos hasta encontrar a los mejores (claro que nos haría falta un cluster para poder terminarlo en el tiempo de entrega del presente trabajo).

Para seleccionar el mejor porcentaje se realizaron pruebas similares a las de la búsqueda local para hallar el porcentaje de cuantos nodos se sacan y cuantos se agregan. El más chico de entre los mejores fue 50, asique éste fue el elegido.

La cantidad de iteraciones sin cambios la dedujimos en función de varias corridas del algoritmo ya que vimos que, por lo general, cuando esta pocas veces sin cambios es porque nunca más encuentra una solución mejor. Por eso tomamos un valor pequeño de 5. Es obvio que depende mucho éste tipo de parámetro si es que el algoritmo será corrido en una instalación que está prendida 24 hs al dia 365 dias al año y se usa para obtener la mejor respuesta o en una aplicación de tiempo real donde la respuesta deberá ser devuelta en milisegundos, sin importar que tan bien esté pero que por lo menos tenga una aproximación.

Al último parámetro no le dimos importancia porque, como se dijo en el párrafo anterior, importaría mucho cuando la respuesta se necesita al instante. Allí éste parámetro podría ser usado como un contador de *time out*; es decir que marque cuando el usuario/cliente podría comenzar a notar la falta de fluidez en el sistema. Como nos interesaba que termine cuando ya no encontraba mejoras, le dimos un valor exagerado (20.000.000) así no interfería.

6.2.2. GolosoRandom

El desarrollo de este algoritmo es muy parecido al explicado ya en el goloso standard. La diferencia radica en la funcion que elige el nodo a ingresar. En ésta, se obtiene el nodo de mayor grado, luego de ordenar la lista de nodos por cantidad de vecinos. Una vez hecho esto, se calcula el grado minimo que se permite. Ésto surge de multiplicar el grado minimo permitido por el porcentaje ingresado por parametro. Una vez hecho esto, se genera una lista, con todos los nodos que cumplen ese grado mínimo y por último se elige un nodo random de esa lista.

6.3. Pseudocódigos

GRASP: Algoritmo GRASP para resolver Vertex Cover $\longrightarrow O(im*n^3m)$

```
1: while No se cumplan las condiciones de parada do
2:     Generar solucion actual golosa azarosa y aplicarle busqueda local
3:     if La solucion actual es mejor que la mejor solucion then
4:         Mejor solucion ← solucion actual
5:     end if
6: end while
7: return Mejor solucion
```

GolosoRandom: Devuelve un random de una lista de candidatos mediante la funcion de seleccion del goloso $\longrightarrow O(n^2)$

```
1: Sacar Aislados del grafo
2: Solucion ← φ
3: while Solucion no es recubrimiento do
4:     Agregar el nodo devuelto por NodoAgregar
5:     Sacar el nodo de la lista de nodos
6: end while
7: Devolver solucion
```

NodoAgregar: Devuelve un nodo para agregar a la solucion GolosaRandom $\longrightarrow O(n)$

```
1: Obtengo el nodo de mayor grado
2: Calculo el minimo grado permitido segun el porcentaje ingresado por parametro
3: Armo una lista con los que cumplen con ese minimo
4: Devuelvo un Random de esa lista
```

6.4. Análisis de complejidad

6.4.1. GolosoGreedy

La complejidad tambien es similiar al goloso, en esta se modifica que se agregan n iteraciones al ordenar la lista de nodos y otras n cuando se arma la lista para despues elegir el random.

Las demas operaciones son en $O(1)$, por lo que la complejidad quedaria:

$$O(n^2 + n * m + m) \leftarrow O(n^3)$$

6.4.2. GRASP

En el peor caso, el algoritmo deberá iterar hasta alcanzar la cantidad máxima de iteraciones (ya que se supone que es mucho mayor a la cantidad de iteraciones que deben pasar sin cambios para parar). Dado que en cada iteración se ejecutan el GolosoGreedy y la BusquedaLocal, la complejidad será

$$O(im * (n^2 + n^5 m)) = O(im * n^5 m)$$

aunque, por supuesto, no se espera que llegue nunca a ejecutar casos de ése estilo.

6.5. Resultados

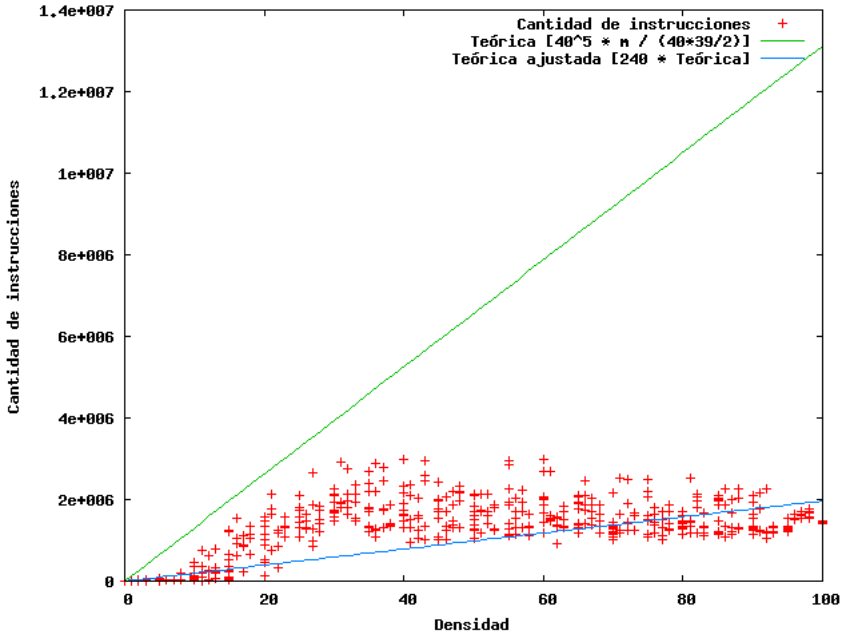

Figura 13: Cantidad de instrucciones con el mejor parámetro encontrado en función de la densidad de los grafos. Evaluado sobre grafos aleatorios de 40 nodos y densidad variable de 0 % a 100 %.

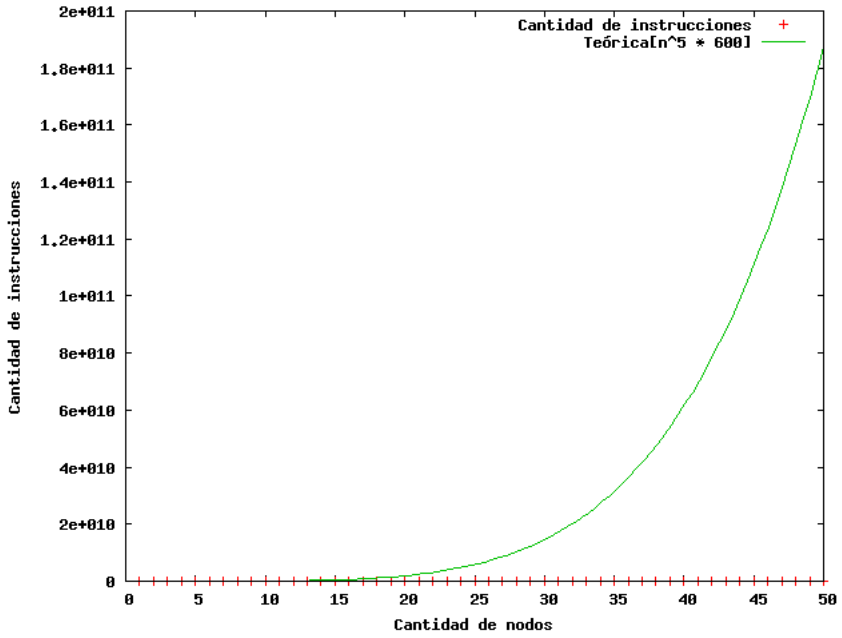

Figura 14: Cantidad de instrucciones con el mejor parámetro encontrado en función de la cantidad de nodos de los grafos. Evaluado sobre grafos aleatorios con 50 % de densidad y cantidad de nodos variable entre 1 y 50

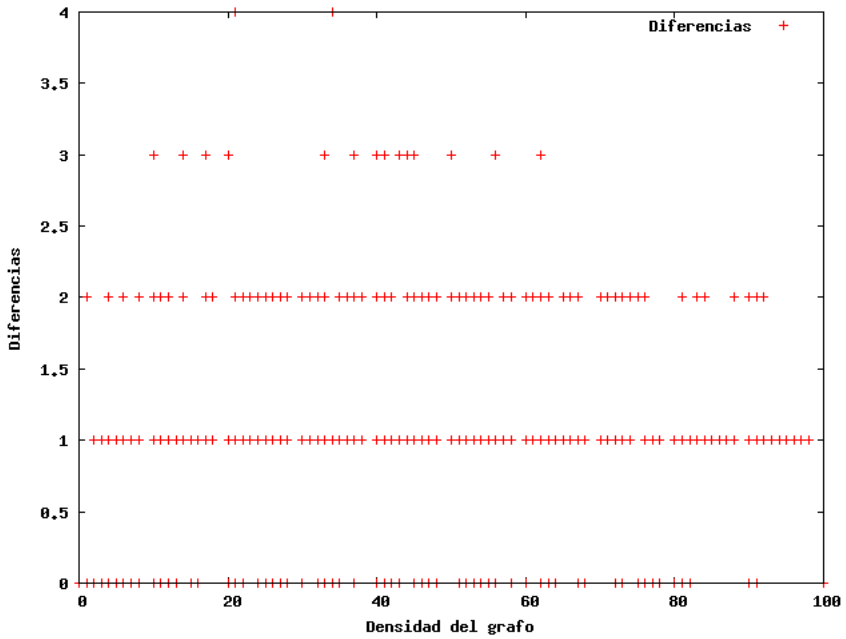

Figura 15: Cantidad de diferencias en comparación con el algoritmo exacto en función de la densidad de los grafos. Evaluado sobre grafos aleatorios de 20 nodos y densidad variable de 0 % a 100 %

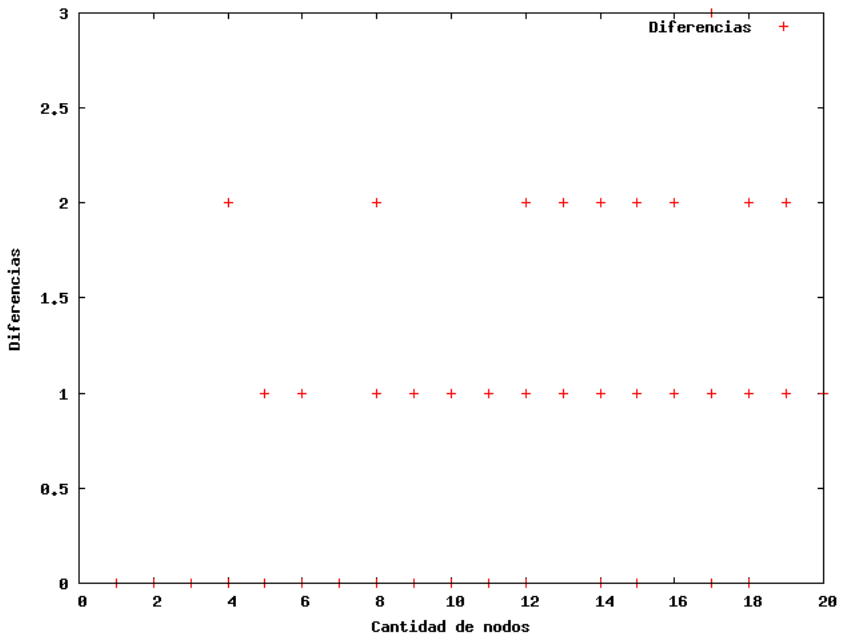

Figura 16: Cantidad de diferencias en comparación con el algoritmo exacto en función de la cantidad de nodos de los grafos. Evaluado sobre grafos aleatorios con 50 % de densidad y cantidad de nodos variable entre 1 y 20

6.6. Discusión

Los gráficos calculados fueron similiares a los de búsqueda local. Notar que aquí se utilizaron los mismos grafos y se realizaron los mismos ajustes a los parámetros (por ej, a m para que valla acorde a la densidad) por las mismas causas descriptas en la búsqueda local.

Notar que la cantidad de instrucciones coinciden bastante con los resultados de búsqueda local en ambos gráficos. Y tiene bastante sentido ya que la complejidad de GRASP es casi idéntica a la de la búsqueda local (prácticamente nunca se alcanzan las iteraciones máximas).

Las diferencias con el exacto dieron patrones parecidos tambien a la búsqueda local, siquiendo un patrón de disminuir los errores cuando aumenta la densidad y ninguno cuando aumentan los nodos. La cantidad de diferencias son bastante pequeñas, al igual que en la búsqueda local.

6.7. Conclusiones

Las conclusiones que tenemos para GRASP es que la nueva lógica que se agregó generó cambios útiles. Aunque se incrementó drásticamente la cantidad de instrucciones, las diferencias con el exacto fueron menores. No creemos que ésto marque al algoritmo mejor que los anteriores, ya que en aplicaciones de tiempo real éste incremento el las instrucciones puede conllevar a acciones indeseadas (como la pérdida en la continuidad de la ejecución, conllevando a una insatisfacción del usuario).

Pero ésto también nos indica que la búsqueda local puede llegar a alcanzar mejores soluciones que con los vecinos que recorre, porque en GRASP - gracias a que se ejecuta el mismo algoritmo varias veces pero cambiando la solución inicial - se verifican en total mas cantidad de vecinos. Y si la ejecución repetitiva de la búsqueda local lleva a cada vez mejores algoritmos, es porque la busqueda local efectivamente no recorre todos los vecinos posibles (cosa que es deseada ya que de no ser así estaríamos hablando de un algoritmo exacto mas que una heurística).

7. Resultados

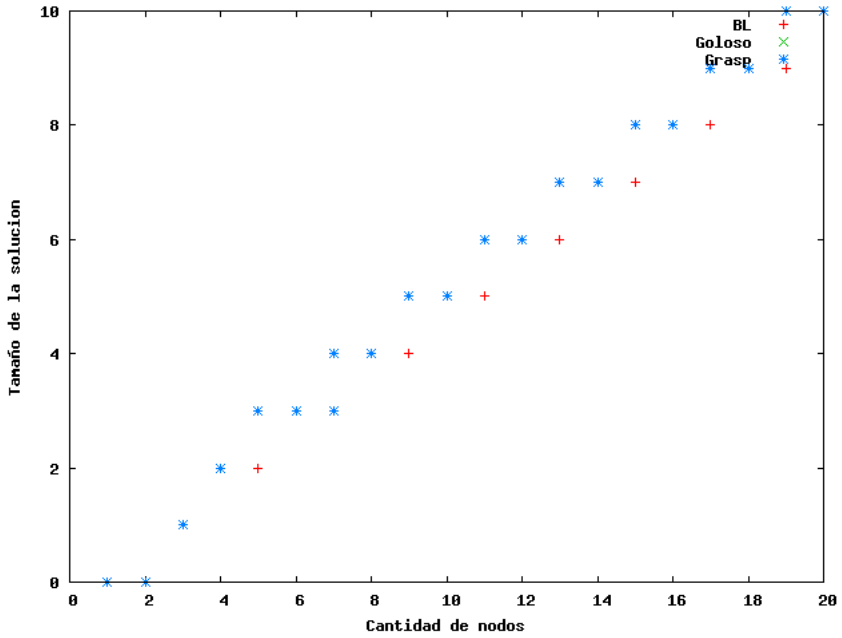

Figura 17: Grafico del tamao de la solucion, en los 3 algoritmos, en grafo Rueda

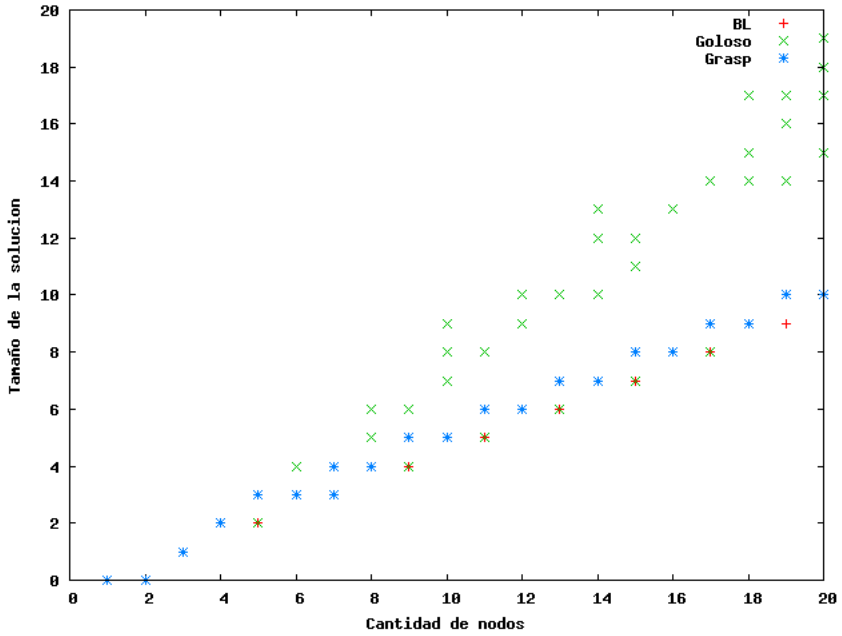

Figura 18: Grafico del tamao de la solucion, en los 3 algoritmos, en grafo Aleatorio

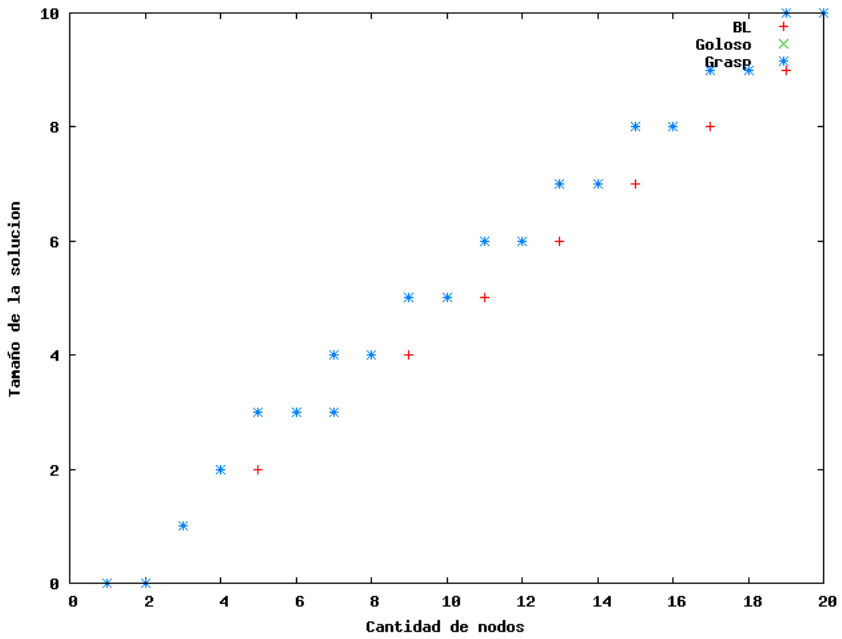

Figura 19: Grafico del tamao de la solucion, en los 3 algoritmos, grafo Completo

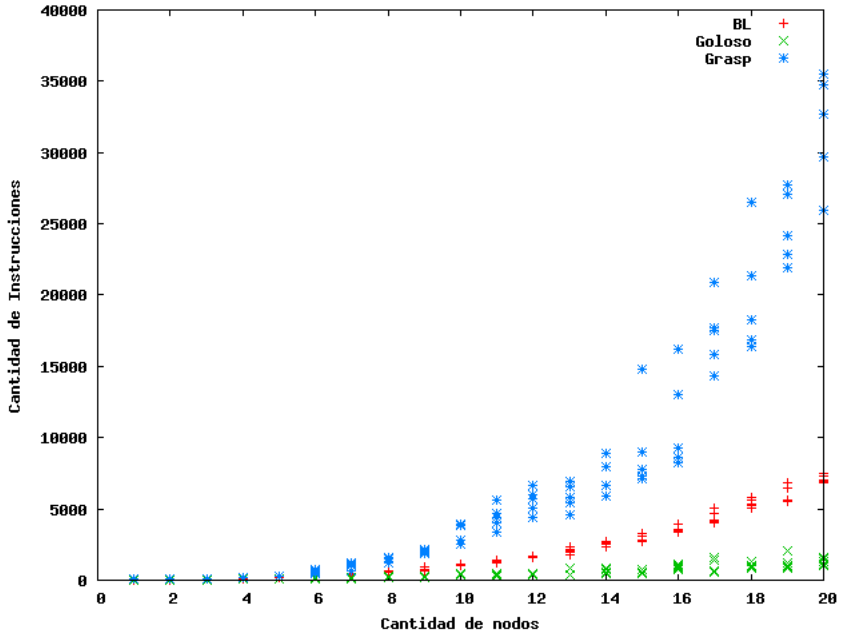

Figura 20: Grafico de la cantidad de instrucciones, en los 3 algoritmos, grafo Rueda

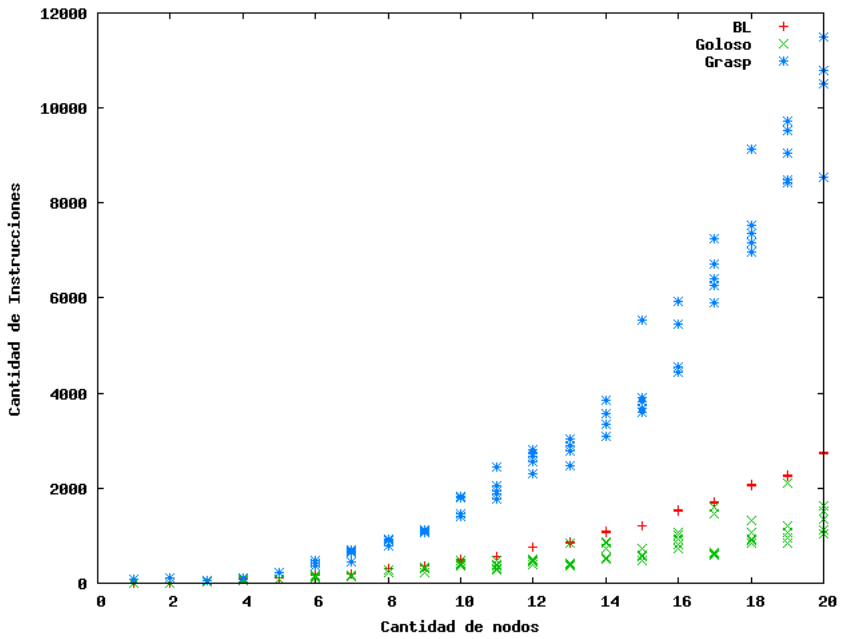

Figura 21: Grafico de la cantidad de instrucciones, en los 3 algoritmos, grafo Aleatorio

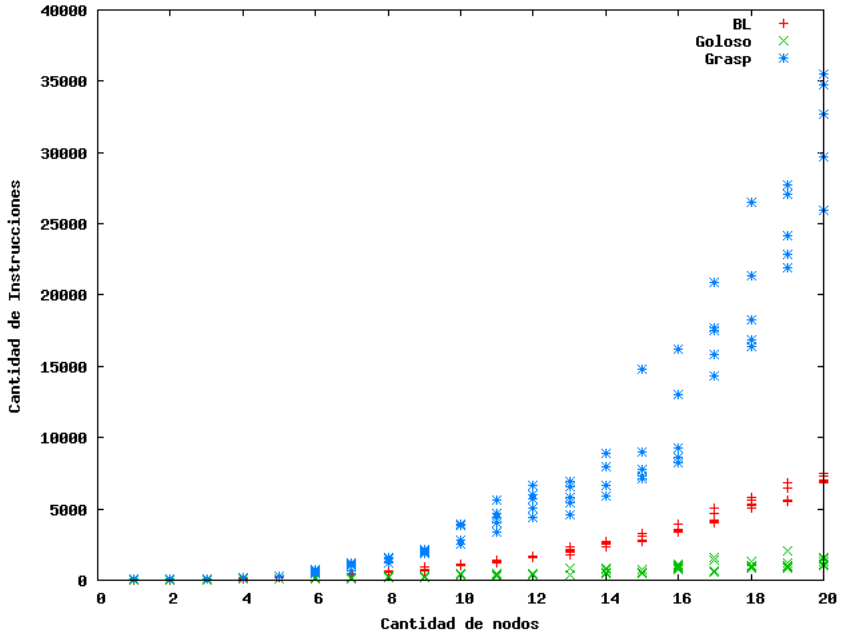

Figura 22: Grafico de la cantidad de instrucciones, en los 3 algoritmos, grafo Completo

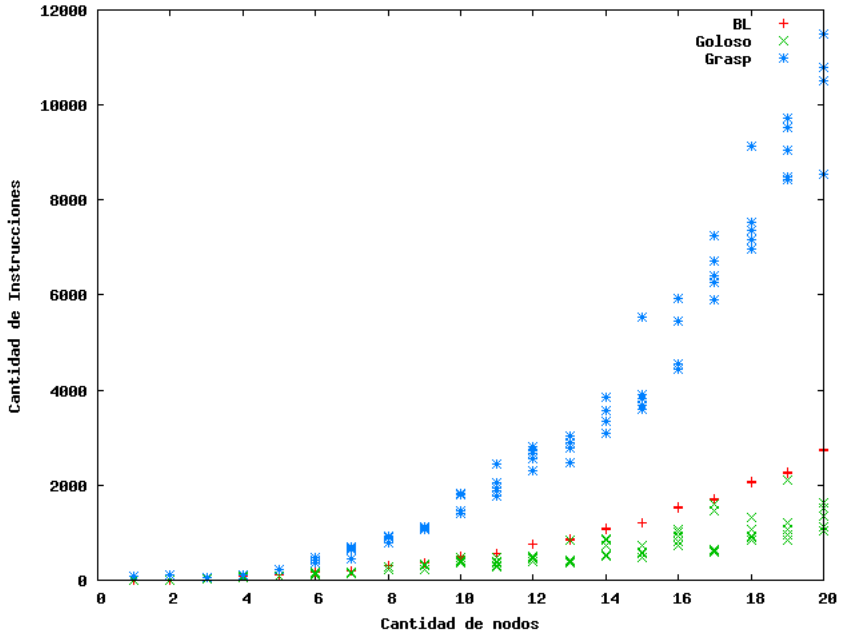

Figura 23: Grafico de la cantidad de instrucciones, en los 3 algoritmos, grafo Bipartito

8. Discusión

En las primeras cuatro figuras, esta graficado el tamao de la solucion, para cada tipo de grafo, en los 3 algoritmos.

En el grafo Rueda, tienen todos un rendimiento similar, ya que es un grafo altamente denso, y son necesarios muchos nodos para recubrirlo totalmente.

En el grafo Aleatorio, GRASP y Busqueda Local se comportan similarmente, aunque hay casos en los que el goloso los supera. En este caso es dificil de analizar, ya que son random los casos generados, pero se puede observar que cuanto mas grande es la cantidad de nodos del grafo, mas grande es la solucion golosa, creciendo casi linealmente. En el grafo Completo, se comportan similar a la rueda, por la misma razon que el anterior, con una leve disminucion en el tamao de la solucion.

En el grafo Bipartito, sucede lo mismo, crece casi linealmente, con una leve ventaja del BL con respecto a GRASP.

En las siguiente figuras, se grafica la cantidad de instrucciones por tipo de grafo, comparando los 3 algoritmo. En Rueda, se puede observar la gran diferencia que le saca GRASP a BL y a Goloso, siendo este ultimo, el que menos instrucciones realiza.

En las siguientes familias, se cumple lo explicado en rueda, con mayor o menor margen para GRASP por encima de Goloso.

9. Conclusiones

Luego de haber analizado a cada algoritmo por separado y luego analizarlo juntos, se llega a las siguientes conclusiones:

- El goloso es rapido, mas comparado al GRASP, pero sus soluciones siempre estan dentro de las peores comparando el tamao.
- La BL, tiene un tiempo de ejecucion medio entre Goloso y GRASP, y sus soluciones son aproximadas al GRASP.
- GRASP tiene los mejores resultados, pero es costoso si se mira el tiempo de ejecucion.

Con estas conclusiones se puede advertir que:

- Para grafos chicos, se recomienda GRASP o BL, ya que su solucion es aproximada a la exacta, y al ser poca la cantidad de nodos, el tiempo de ejecucion no influye demasiado.
- Para grafos grandes, Goloso o BL, el primero en los casos en que se quiera rapidez y menos eficacia y viceversa para el segundo.

10. Apéndice A: uso de la interfaz de usuario

La interfaz del programa fue realizada en Python 2.5 para mayor agilidad en la implementación. Para ejecutarlo, correr el comando **python vertexcover.py** en la consola, previa instalacion del interprete python.

Una vez dentro, se le pide elegir el algoritmo a correr, luego los archivos de entrada, salida y datos, los cuales tienen que ser ingresados con el PATH COMPLETO. Luego pide los parametros, dependiendo del algoritmo elegido. El programa va a devolver la cantidad de instrucciones, el tamao de la solucion, el tiempo de ejecucion, asi como tambien el grafo ingresado.

11. Apéndice B: mejores parámetros de la Búsqueda Local y GRASP

11.1. Búsqueda Local

El sistema de pruebas de la Búsqueda Local con solución inicial *naive* devolvió a éstos 5 mejores puntajes y sus respectivos ganadores

```
Mejores puntajes:
  2011:
    (4,7)
    (5,7)
    (6,7)
  1968:
    (6,8)
    (7,8)
  1864:
    (4,8)
    (5,8)
  1837:
    (3,6)
    (4,6)
    (5,6)
  1780:
    (6,9)
    (7,9)
    (8,9)
Me quede con: (4,7)
```

El mismo algoritmo pero con solución inicial golosa arrojó los siguientes punajes

```
Mejores puntajes:
  1789:
    (4,7)
    (5,7)
    (6,7)
  1776:
    (6,8)
    (7,8)
  1750:
    (4,8)
    (5,8)
  1659:
    (3,6)
    (4,6)
    (5,6)
  1615:
    (3,7)
Me quede con: (4,7)
```

11.2. GRASP

Éstos son los resultados del sistema para el mejor porcentaje para el GolosoRandom

```
Mejores puntajes:
  301:
    (50)
    (51)
    (52)
    (53)
    (59)
  300:
    (54)
  297:
    (45)
    (47)
    (48)
    (49)
  295:
    (46)
    (58)
  291:
    (55)
    (70)
    (71)
Me quede con: (50)
```