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, Heursticas, Metaheursticas, Algoritmo Goloso, Bsqueda 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

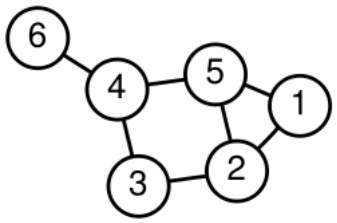

Figura 1: Grafo Ejemplo

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").

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

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
```

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 $O(n)$. 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. Este costo se puede acotar por n ya que 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 $O(2^n + n * 2^n) = O(n * 2^n)$, donde n es la cantidad de nodos del grfico original.

3.5. Resultados

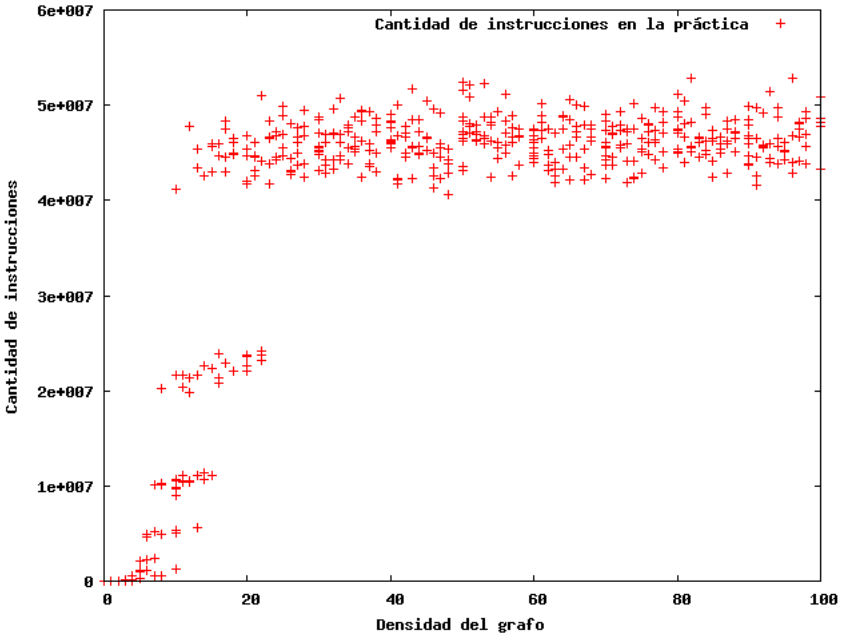

Figura 2: Cantidad de instrucciones en funcion a la densidad del grafo

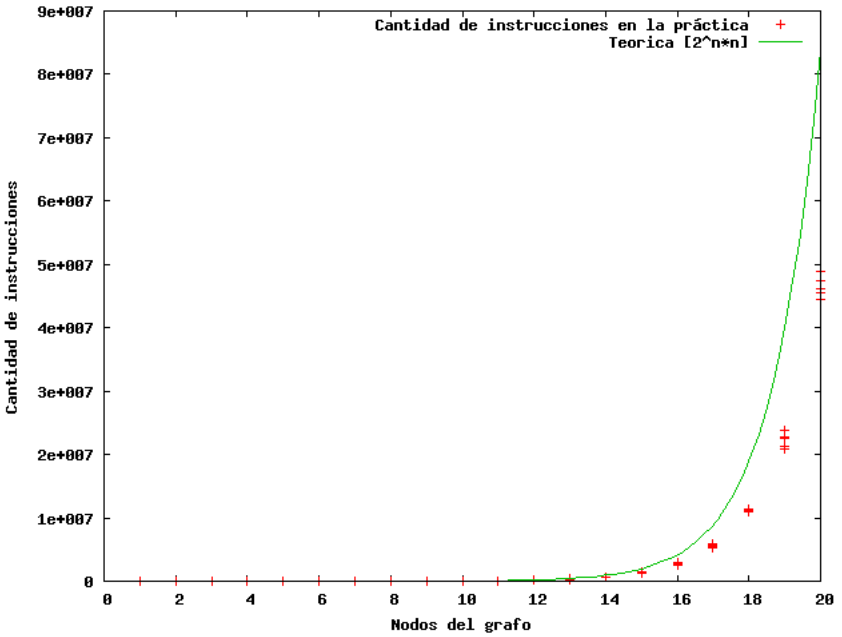

Figura 3: Cantidad de instrucciones en funcion de la cantidad de nodos

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 %.

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 $O(n(n+m)+n) \rightarrow O(n^2 + n*m+n) \rightarrow O(n^2)$

4.5. Resultados

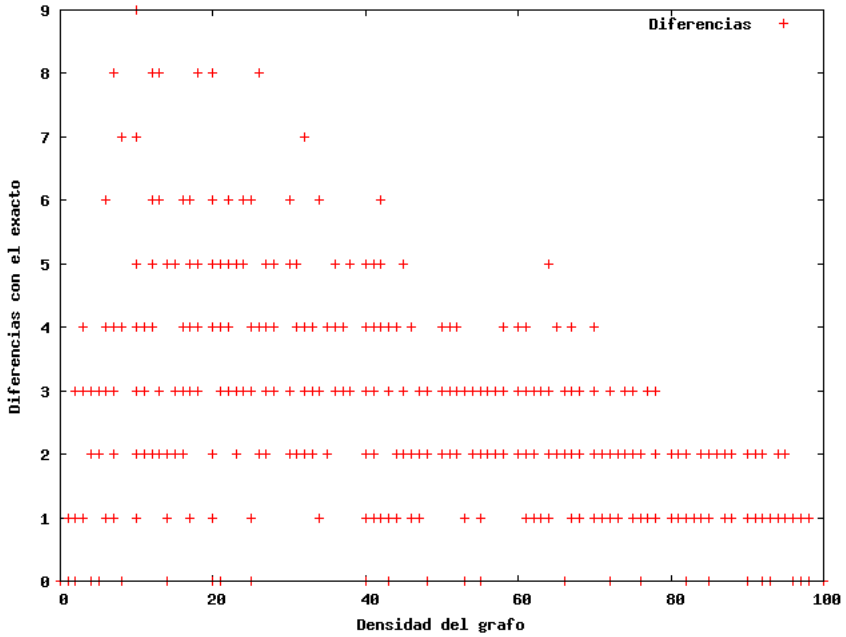

Figura 4: Diferencias con el exacto, en la densidad del grafo

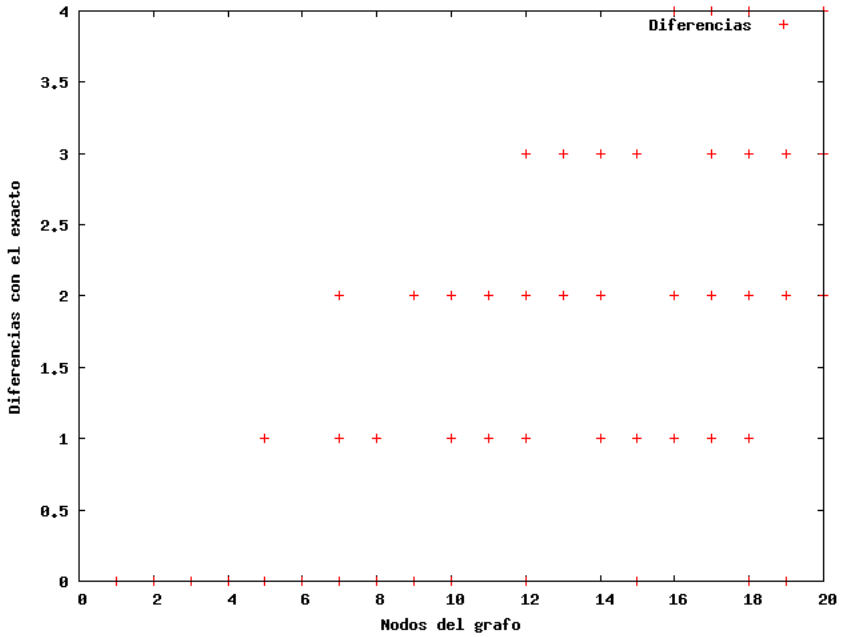

Figura 5: Diferencias con el exacto, en la cantidad de nodos

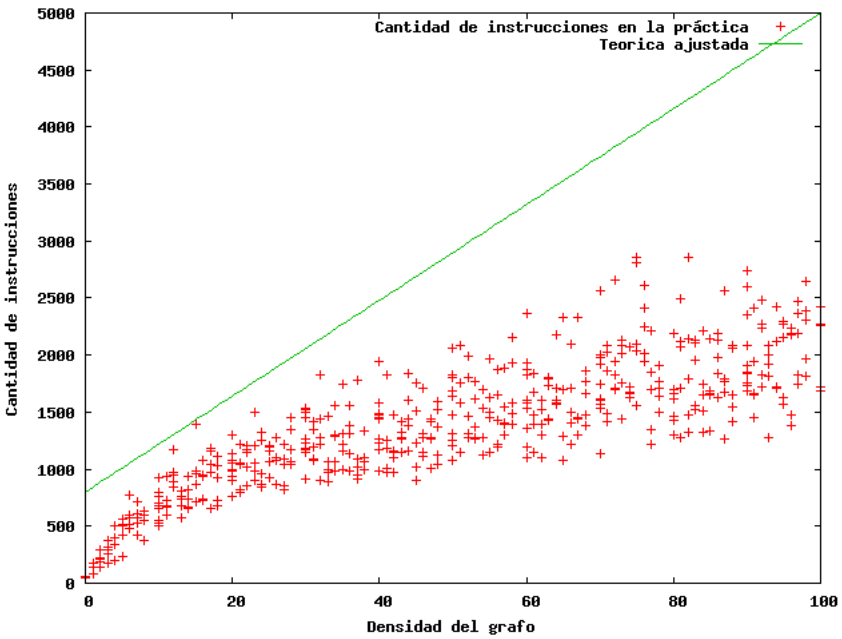

Figura 6: Cantidad de instrucciones, en la densidad del grafo

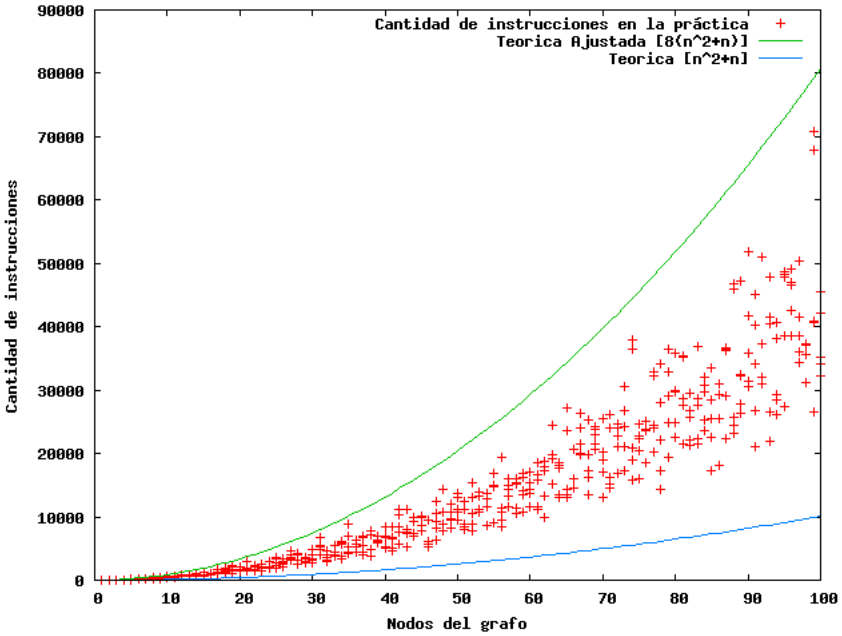

Figura 7: Cantidad de instrucciones, en la cantidad de nodos

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. 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 $n^2 + n$, y ajustandola, multiplicandole una constante de 8, se ve que la cota es buena y cumple la complejidad practica.

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

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). Luego se quitan los primeros cs de la lista de nodos del recubrimiento de s y se corrobora si es un recubrimiento. De serlo, se devuelve como mejor vecino (ya que, como la función objetivo es proporcional a la cantidad de nodos del recubrimiento, al sacar una cantidad positiva de éstos el resultado será siempre mejor). Pero si éste no es recubrimiento entonces se agregan ca nodos (nuevos, no se incluyen los que se acaban de sacar) para volver a realizar la verificación. Como se supone que la cantidad de nodos que saco es mayor a la que agrego, de ser un recubrimiento al agregar se está obteniendo mejor objetvo, por lo que se devuelve ésta nueva solución. De no serlo, se regresa a s y se quitan ésta vez cs nodos pero *desfazados* en la lista del recubrimiento por 1 nodo (es decir, el primero ya no se toma más pero sí el último). Si no se consigue ningún vecino mejor después de realizar éste barrido, entonces 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.

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.

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

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.

5.3. Pseudocódigos

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

```
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 $\longrightarrow O((t-cs)(cs+m+(n-t)(ca+m)))$

```
1: s es la solucion actual
2: t ← logitud de s
3: cs ← cuantos nodos saco
4: ca ← cuantos nodos agrego
5: for cada sublista contigua de longitud cs en s do
6:     if Sacando esta sublista de s sigue siendo recubrimiento then
7:         return s - sublista
8:     else
9:         for Cada sublista contigua de longitud ca en la lista de nodos que no
           estaban en la solucion do
10:            if Agregando ca nodos a s - sublista sigue siendo un recubrimiento
               then
11:                return s - sublista + nuevos nodos
12:            end if
13:        end for
14:    end if
15: end for
16: return No hay mejor vecino
```

5.4. Análisis de complejidad

5.4.1. 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, el cual va recorriendo las posibles sublistas a sacar de s. Por lo que podemos ver, el primer elemento de ésta sublista puede ir desde 1 hasta $t - cs$, ya que de estar en algun momento en una posición mayor, el último elemento de la sublista no tendría posición. Por lo tanto se ejecuta $t - cs$ veces.

Dentro se quitan los nodos, lo cual es del orden de cs y luego se pregunta si es o no recubrimiento. Dado que éste método recorre todos los ejes y para cada uno de ellos recorre todos los nodos para ver si está o no alguno de sus extremos en el posible recubrimiento, la complejidad es del orden de $m(t - cs)$ (ya que $t - cs$ es la longitud del posible recubrimiento).

En el peor caso la respuesta anterior es negativa asique debo seguir calculando. A continuación está el otro bucle, el que agrega nodos. Análogamente al bucle anterior, aquí también se calculan sublistas pero ésta vez sobre la lista de nodos que no están en la solución inicial. Dado que ésta es de longitud $n - t$ y por el mismo argumento que para el primer bucle, se repite $(n - t) - ca$ veces.

Por último, dentro de ése bucle debo agregar todos los nodos (ca) y luego consultar si es un recubrimiento (dado que la nueva posible solución tendrá longitud $t - cs + ca$, la complejidad será del orden de $m(t - cs - ca)$).

Juntando toda la información aislada mostrada anteriormente se obtiene que la complejidad del algoritmo es

$$O(n + (t - cs)(cs + m(t - cs) + (n - t - ca)(ca + m(t - cs - ca)))) = O()$$

5.5. Resultados

5.6. Discusión

5.7. Conclusiones

6. GRASP

6.1. Introducción

6.2. Desarrollo

6.3. Pseudocódigos

6.4. Análisis de complejidad

6.5. Resultados

6.6. Discusión

6.7. Conclusiones

7. Resultados

Los casos de prueba fueron generados automáticamente con valores azarosos:

- hola

8. Discusión

9. Conclusiones

10. Referencias

- hola