DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Reentrega del Trabajo Práctico III

Grupo 5

17 de diciembre de 2013 — Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Aleman, Damian Eliel	377/10	damianealeman@gmail.com
Amil, Diego Alejandro	68/09	amildie@gmail.com
Barabas, Ariel	775/11	ariel.baras@gmail.com
Fernández, Gonzalo Pablo	836/10	ralo4155@hotmail.com

Corrector: JPD

Entrega	I	II	III	IV	V	VI	Promedio
Primera	8	R	8.6	6.7	6.5	6	6.6 ®
Segunda	-		-	-	-	-	
				Nota Final			

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Algoritmo Exacto

1.1. Implementación del Algoritmo

Para realizar el algoritmo exacto, utilizamos la técnica de programación de backtracking. Para ver cuál es el clique con una frontera de mayor cardinalidad, calculamos la frontera de cada clique que hay en el grafo. Partimos de una solución en la que el conjunto de nodos es vacío[1]. Luego utlizamos recursión para agregar cada posible nodo al conjunto. La función recursiva se llama a sí misma dos veces: una agregando el i-ésimo nodo a la solución actual y otra sin agregarlo.

A la función recursiva se le pasan como parámetros la solución actual y la solución óptima hallada hasta el momento (ambas inicializadas con la solución vacía). Cada vez que se entra a la función recursiva se evalúa el tamaño de la frontera de la solución actual y si es mayor a la frontera de la solución óptima, la solución actual reemplaza a la óptima. Una vez finalizado el algoritmo se retorna la solución almacenada como óptima. La solución vacía almacenada como óptima será reemplazada en cuanto se evalúe cualquier solución con tamaño de frontera mayor a cero. Como cada nodo es una clique, si el algoritmo retorna la solución vacía significa que el grafo no tiene aristas.

Con el objetivo de reducir la complejidad del algoritmo exacto se ha implementado una importante poda. Cuando la función va a llamarse recursivamente, primero se evalúa si tiene sentido agregar el nodo a la solución actual, es decir si con el nodo que se agrega, la solución actual sigue representando una clique. Si con ese nodo no se forma un clique, luego ese subconjunto de nodos sumado a otros nodos, tampoco va a ser clique. Por esta razón, se poda toda solución que agregue el nodo a la solución actual, es decir todo subconjunto de nodos, que no forman una clique.

Como esta primera poda resultó ser muy poco inteligente, se implementó una segunda poda basada en la cantidad de nodos de la clique. Proponemos que ninguna clique máxima puede tener más de $n/2$ nodos o, en otras palabras, que dada cualquier clique de k nodos ($k > n/2$) existe una clique con menos nodos y mayor frontera. La demostración de tal enunciado surge de ver cómo se altera la frontera de una clique al quitar un nodo v. Las aristas incidentes a v pueden ser de dos tipos. Pueden conectar a v con un nodo perteneciente a la clique o pueden conectar a v con un nodo que no pertenezca a la clique.

Las aristas del primer tipo son aristas que no pertenecían a la frontera de la clique (ya que unían dos nodos pertenecientes a la clique) pero al quitar un nodo v de la clique, estas pasan a pertencer a la frontera de la clique, ya que conectan un nodo fuera de la clique (el nodo v que se acaba de quitar) con uno que sí está en ella (el otro nodo que sigue perteneciendo a la clique). La cantidad de aristas de este tipo es exactamente la cantidad de nodos de la clique menos uno, es decir k-1, porque para que sea una clique el nodo v que fue quitado debe haber estado conectado con todos los otros k-1 nodos de la clique. Entonces, al quitar un nodo cualquiera la cantidad de aristas de frontera de la clique aumenta en exactamente k-1 aristas.

Las aristas del segundo tipo son aristas que son parte de la frontera de la clique pero al quitar un nodo v dejan de serlo, ya que conectan dos nodos de los cuales ninguno pertenece a la clique (v deja de pertenecer y el otro nodo no pertenecía). La cantidad de aristas de este tipo es la cantidad de aristas incidentes a v menos las correspondientes a unir a v con el resto de los nodos de la clique, es decir d(v) - (k-1). Entonces, al quitar un nodo cualquiera la cantidad de aristas de frontera de la clique disminuye en el grado del nodo menos k-1, una cantidad que no puede ser mayor a

[1] Es debatible si esta solución es válida o no, pero al tener frontera igual a cero, cualquier solución válida con al menos uno de frontera será mejor. En todo caso, si el grafo no tiene aristas, se devolverá esta solución vacía.

n-1-(k-1).

En definitiva, al quitar un nodo de una clique la frontera de la clique aumenta en k-1 y disminuye en n-1-k+1 = n-k. Si la cantidad de nodos de la clique es $k > n/2$, entonces la frontera aumenta en al menos $n/2$ y disminuye en a los sumo $n/2$. En conclusión, para toda clique con $k > n/2$ nodos, existe una clique de mayor frontera que se obtiene quitándole nodos a la clique. Habiendo demostrado esto, podemos asegurar que la clique máxima de un grafo no puede tener más de $n/2$ nodos. En no evaluar dichas soluciones consiste la segunda poda implementada. A continuación se muestra un pseudocódigo del algoritmo exacto basado en backtracking recursivo.

Algoritmo 1: Pseudocódigo del algoritmo exacto

```
   Data: Grafo G(V, E)
1  Solucion solucion_optima ← solucion_vacia
2  Solucion solucion_actual ← solucion_vacia
3  Recursion(0, solucion_optima, solucion_actual, 0)
4  return solucion_optima
```

Algoritmo 2: Llamada recursiva

```
   Data: int i, Solucion solucion_optima, Solucion solucion_actual, int size_solucion
1  solucion_optima ← Max(solucion_optima, solucion_actual)
2  if i = cantNodos then
3  |  Terminar recursion
4  if size_solucion + 1 > cantNodos/2 then
5  |  Terminar recursion
6  if se_conecta_con_clique(i, solucion_actual) then
7  |  Agregar i a solucion_actual
8  |  Recursion(i + 1, solucion_optima, solucion_actual, size_solucion + 1)
9  |  Quitar i de solucion_actual
10 |  Recursion(i + 1, solucion_optima, solucion_actual, size_solucion)
11 else
12 |  Recursion(i + 1, solucion_optima, solucion_actual, size_solucion)
```

Como se mencionó en la sección *Pautas de Implementación*, Una solución se representa con un vector de booleanos y un entero. El vector de booleanos tiene un elemento por cada nodo del grafo. El valor del *i*-ésimo booleano indica si el *i*-ésimo nodo del grafo pertenece al conjunto de nodos representado por la solución. El entero representa el tamaño de frontera del conjunto representado. Si es negativo, El conjunto no es una clique.

Como el conjunto de nodos en la solución se representa con un vector de booleanos, agregar y quitar nodos (líneas 7 y 9 del pseudocódigo) debería consistir únicamente en alternar el correspondiente elemento del vector. Sin embargo, esas funciones también se encargan de recalcular la frontera en función del nodo agregado o quitado, para lo cual deben recorrer los nodos adyacentes al él.

La función Max compara dos soluciones y retorna la de mayor frontera. **Como lo único que hace esta función es comparar dos enteros (el segundo elemento de la dupla de cada solución) no es necesario mostrar un pseudocódigo para ella.** $cantNodos$ es la cantidad de nodos del grafo. Cuando el índice del nodo que se va a agregar i es igual a dicha cantidad, significa que ya no hay más nodos para agregar y la recursión debe terminar. En otras palabras, se llegó a una hoja del árbol de backtracking. La función *se_conecta_con_clique* toma un nodo i y una solución S y retorna si la solución S seguiría representando una clique tras agregarle el nodo

i. Para ello evalúa si existe una arista entre el nodo i y cada nodo de la clique representada por S. **A continuación se muestra el pseudocódigo de la función *se_conecta_con_clique*.**

Algoritmo 3: se_conecta_con_clique

```
   Data: int nodo, Solucion solucion
1  for int i ∈ [ 0..cantNodos-1 ] do
2  |  if i ∈ solucion ∧ ¬∃ (i, nodo) ∈ E_G then
3  |  |  return FALSE
4  return TRUE
```

Como la solución se representa con un arreglo de booleanos, ver si el nodo *i* pertenece a la solución (línea 2: $i \in solucion$) consiste en obtener el i-ésimo booleano del arreglo, lo cual se realiza en tiempo constante. Como el grafo se representa con matrices de adyacencia, saber si determinada arista (i, $nodo$) existe (línea 2: $\neg\exists$ (i, $nodo$) $\in E_G$) consiste en obtener el elemento (i, $nodo$) de dicha matriz, lo cual también se realiza en tiempo constante. Como esta función pertenece a la clase *grafo*, el grafo en el cual se verifica la existencia de las aristas no se pasa como parámetro de la función.

La implementación de la segunda poda consiste simplemente en evaluar la cantidad de nodos de la solución que se recorre (líneas 4 y 5 del pseudocódigo). Si esta es mayor a la mitad de la cantidad de nodos se retorna en lugar de seguir agregando nodos a la solución. Como la solución es un vector de booleanos, la cantidad de nodos de la solución se pasa como un parámetro más del algoritmo. Cuando se agrega un nodo a la solución (línea 8) se incrementa en uno el valor de dicho parámetro.

1.2. Orden de complejidad

El algoritmo exacto recorre todas las posibles soluciones. Si bien hemos implementado una poda, en el peor caso (un grafo completo K_n) todas las soluciones serán válidas y todas ellas deberán ser evaluadas. Dado que en una solución cada nodo puede estar o no estar, la cantidad de soluciones posibles (la cantidad de subconjuntos del conjunto de nodos) es 2^n con n la cantidad de nodos. Las operaciones realizadas para cada solución son polinomiales y, por lo tanto no aportan **significativamente** a la complejidad exponencial.

Como se mencionó en la sección *Pautas de Implementación*, el grafo se representa con listas de adyacencia y con matriz de adyacencia simultáneamente. La función *se_conecta_con_clique* **recorre los n nodos del grafo. Como ya se vió en el pseudocódigo de esta función, todas las operaciones sobre cada nodo son constantes (Como el grafo está representado con matriz de adyacencia, saber si hay una arista entre un par de nodos es constante)**, por lo que la complejidad de esta función es la cantidad de nodos n del grafo. Cuando se agregan o se quitan nodos de una solución, la frontera de dicha solución debe recalcularse.

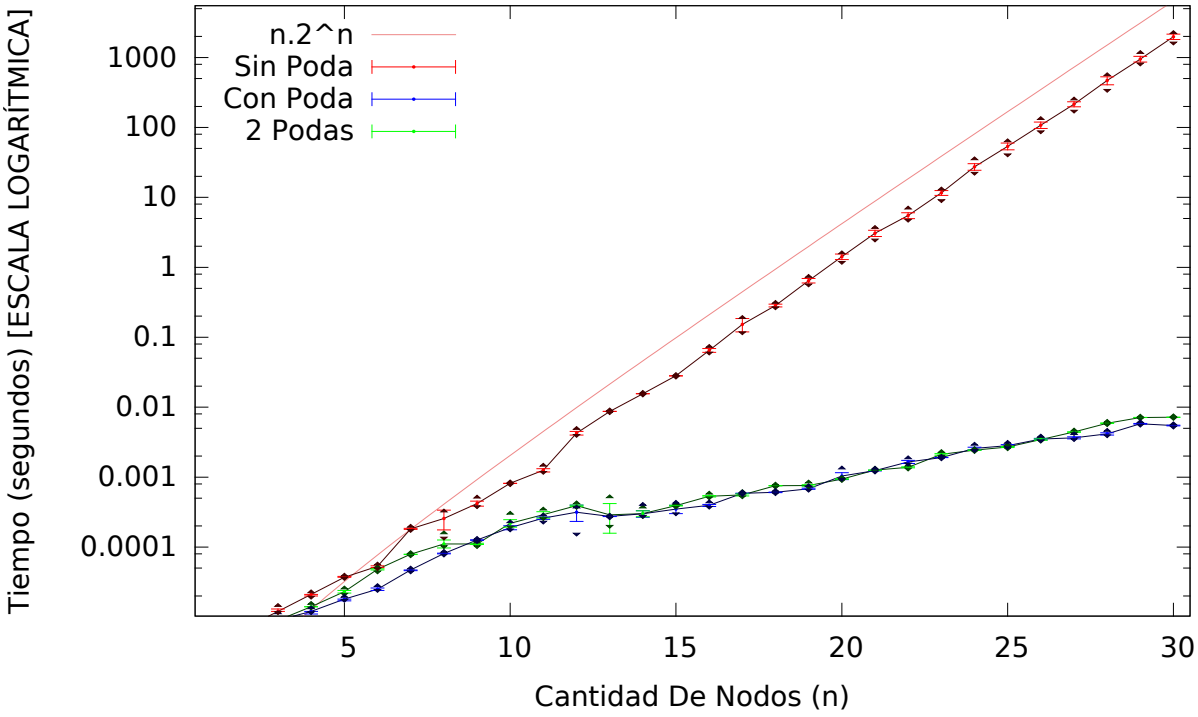
Como ya se mencionó, no se cuenta con una función que calcule, dado un conjunto de nodos, la frontera de dicho conjunto en el grafo. En lugar de eso, la frontera de una solución se calcula cuando a una solución anterior se le agrega o quita un nodo, ya que todas las soluciones que utilizamos se construyen agregándole o quitándole un nodo a una solución ya usada. La complejidad de agr**e**gar o quitar un nodo i es $d_G(i)$ (el grado de dicho nodo, que también puede acotarse superiormente por n), ya que deben evaluarse todas sus aristas. En definitiva, la complejidad del algoritmo exacto es $2^n * (n + n + n) \in$ **O(n.2^n).**

1.3. Experimentación

1.3.1. Performance

El archivo *test_exacto.cpp* en la carpeta *codigo/exacto* contiene la implementación del código que mide los tiempos de ejecución del algoritmo exacto. Se ejecutó este programa con $N = \mathbf{30}$, $s = 1$ y $k = 10$. Es decir, para cada n entero entre 1 y **30**, se generaron 10 grafos aleatorios (con la función *generar_aristas_aleatorias*). Para cada instancia generada se corrieron **3** verisones del algoritmo exacto. Uno con la **primera** poda**, otro con ambas podas** y otro sin ella**s**. En el siguiente gráfico se muestran los resultados obtenidos.

Gráfico de performance del algoritmo exacto en función de la cantidad de nodos del grafo



Cada punto • en el gráfico representa el promedio de los tiempos medidos para cada una de las 10 ejecuciones de una determinada cantidad de nodos del grafo. El tamaño del segmento vertical sobre cada punto • representa su varianza asociada. Además, para cada cantidad de nodos n se graficaron la máxima medición con ▲ y la mínima medición con ▼.

La función graficada con una curva sin puntos es **la** función $\mathbf{2.10^{-7}.x.2^x}$. Al estar utilizando escala logarítmica en el eje de ordenadas, esta curva se ve como una recta. Como se puede observar, la curva definida por la versión sin poda del algoritmo (la curva resultante de unir los puntos •) se asemeja mucho a tal curva. Mientras que la versión con **una** poda tiene menores tiempos promedio de ejecución, si bien tiene la misma complejidad.

La versión con dos podas no se ve mejor a la versión con una poda. Esto podría deberse a la forma en que los grafos aleatorios son generados. Por cada posible arista, se tiene 0.5 probabilidad de agregarla al grafo. En promedio, la cantidad de aristas de un grafo aleatorio generado de esta forma debería estar al rededor de $(n.n\text{-}1)/4$ (la mitad de la máxima cantidad de aristas posibles. Es probable entonces que se generen pocas cliques grandes y, por lo tanto la segunda poda no elimine más soluciones que

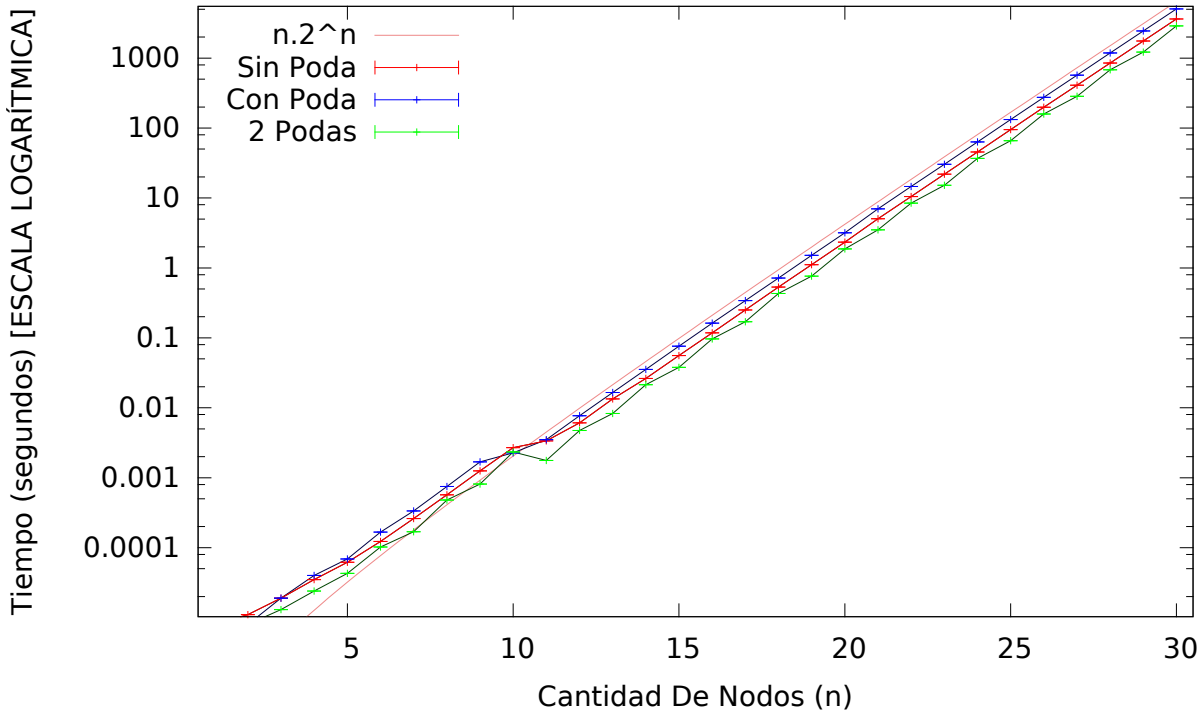
la primera. Lo que sí resulta raro es que en determinados casos, la versión con dos podas tenga mayores tiempos de ejecución que la versión con una.

En la mayo**r**ía de los casos, la varianza de **todas las** versiones es tan pequeña que apenas puede verse en el gráfico. Esto nos dice que **las tres** versiones del algoritmo son estables**, en cuanto a que, sin importar la estructura específica de cada instancia, el tiempo de ejecución para instancias del mismo tamaño, es muy similar.** Era de esperarse de la versión sin poda, ya que dado cualquier grafo, la cantidad de soluciones que deben recorrerse es siempre la misma. Sin embargo, de la**s versiones con podas** se esperaba una varianza bastante superior debido a que la cantidad de soluciones que est**os deben** recorrer depende en gran medida de **la estructura específica del** grafo entrada.

Como se mencionó en la sección *Implementación del Algoritmo*, la primera poda no sirve en grafos completos, ya que todo conjunto de nodos es una clique y, por lo tanto, ninguna solución se poda. Para solucionar esto, se implementó la segunda poda que, sin importar la estructura del grafo (sin importar la densidad de aristas) la cantidad de soluciones que evalúa es exactmente la mitad que la versión sin podas.

El siguiente experimento busca mostrar tanto la similitud entre las versiones con y sin poda, como la mejora de la segunda poda respecto a estas dos, en instancias con grafos completos. Se ejecutó el archivo *test_peor_caso.cpp* en la carpeta *codigo/exacto*. Funciona igual que el test anterior pero en lugar de generar k grafos aleatorios genera un único grafo completo para cada valor de n. Se repitieron los parámetros $N = 30$ y $s = 1$. En el siguiente gráfico se muestran los resultados obtenidos.

Gráfico de performance del algoritmo exacto con grafos completos



Cada punto en el gráfico representa la medición obtenida para un determinado tamaño de entrada. Como se realizó una medición por tamaño, no hay varianza, mínimos ni máximos. El grafo entrada en todos los casos es un grafo completo generado

con la función *generar_completo* de la clase grafo. Lo primero que salta a la vista es que los tiempos de ejecución de la versión con una poda son superiores a los de la versión sin podas. Debido a la escala logarítmica, la versión con dos podas no parece tener tiempos de ejecución mucho mejores que las otras dos versiones, sin embargo son de al rededor de la mitad, tal como se esperaba.

2. Apéndice

2.1. Grafo

Sólo se incluyen en esta sección las funciones nuevas o modificadas para la reentrega

```
//Construye un grafo completo
// (llamar despues del constructor basico)
void generar_completo() {
  for(int i = 0; i < cantNodos; ++i)
    for(int j = i; j < cantNodos; ++j)
      asociar(i, j);
}
```

2.2. Algoritmo Exacto

```
Solucion exacto_2p(grafo &g) {
  Solucion optima = make_pair(vector<bool>(g.cantNodos,false), 0);
  Solucion actual = make_pair(vector<bool>(g.cantNodos,false), 0);
  exacto_recur_2p(g, 0, optima, actual, 0);
  return optima;
}

Solucion exacto(grafo &g) {
  Solucion optima = make_pair(vector<bool>(g.cantNodos,false), 0);
  Solucion actual = make_pair(vector<bool>(g.cantNodos,false), 0);
  exacto_recur(g, 0, optima, actual);
  return optima;
}

Solucion exacto_sin_poda(grafo &g) {
  Solucion optima = make_pair(vector<bool>(g.cantNodos,false), 0);
  Solucion actual = make_pair(vector<bool>(g.cantNodos,false), 0);
  exacto_recur_sp(g, 0, optima, actual);
  return optima;
}

void exacto_recur_2p(grafo &g, int i, Solucion &optima, Solucion &actual, int siz
  if (actual.second > optima.second) {
    optima = actual;
  }
  if (i==g.cantNodos) {
    return;
  }
  // Segunda Poda: si al agregar nodos, la clique es demasiado
  // grande, entonces ya no sirve seguir agregando
  if (size_solucion+1 > (g.cantNodos/2)) {
    return;
  }

  if ( g.se_conecta_con_clique(i, actual.first) ) {
    g.alternar(actual, i);
    exacto_recur_2p(g, i+1, optima, actual, size_solucion + 1);
    g.alternar(actual, i);
```

```
    exacto_recur_2p(g, i+1, optima, actual, size_solucion);
  } else {
    exacto_recur_2p(g, i+1, optima, actual, size_solucion);
  }
}

void exacto_recur(grafo &g, int i, Solucion &optima, Solucion &actual) {
  if (actual.second > optima.second) {
    optima = actual;
  }
  if (i==g.cantNodos) {
    return;
  }

  // si se_conecta_con_clique sigo la recursion con y sin ese nodo
  if ( g.se_conecta_con_clique(i, actual.first) ) {
    g.alternar(actual, i);
    exacto_recur(g, i+1, optima, actual);
    g.alternar(actual, i);
    exacto_recur(g, i+1, optima, actual);
  // si no se conecta con mas nodos tampoco va a ser una clique,
  // luego hago la recursion sin ese nodo
  } else {
    exacto_recur(g, i+1, optima, actual);
  }
}

void exacto_recur_sp(grafo &g, int i, Solucion &optima, Solucion &actual) {
  if (actual.second > optima.second) {
    optima = actual;
  }
  if (i==g.cantNodos) {
    return;
  }
  g.alternar(actual, i);
  exacto_recur_sp(g, i+1, optima, actual);
  g.alternar(actual, i);
  exacto_recur_sp(g, i+1, optima, actual);
}
```