Práctica 3, Parte 1:

Algoritmos Voraces (Greedy)

Índice:

GRUPO 1

Fc.Javier Azpeitia Muñoz
Christian Andrades Molina
Guillermo Muriel Sánchez lafuente
Miguel Keane Cañizares
Mercedes Alba Moyano

- Recubrimiento de un grafo no dirigido

Consideremos un grafo no dirigido G = (V, E). Un conjunto U se dice que es un recubrimiento de G si U ⊆ V y cada arista en E incide en, al menos, un vértice o nodo de U , es decir ∀(x, y) ∈ E, bien x ∈ U y ∈ U . Un conjunto de nodos es un recubrimiento minimal de G si es un recubrimiento con el menor número posible de nodos.

1. El problema de la cobertura de vértices (vertex cover)

Perteneciente a la clase de complejidad NP-completos y se encuentra junto con otros 20 problemas computacionales en una lista elaborada por Richard Karp en 1972. Los problemas se caracterizan por que la solución óptima, es decir, la que menos vértices incluye, no puede ser hallada mediante técnicas greedy. Pero si se obtienen aproximaciones mediante heurísticas válidas.

Dentro de estos problemas, encontramos el algorítmo de aproximación 2, el cual cogerá todas las soluciones que se aproximen por menos de 2 a la solución óptima. Este algortimo funciona con la siguiente fórmula:

Solución óptima: s
Recubrimiento obtenido: r

Máximo: r/s=2 → El resultado obtenido por esta división no puede ser mayor de dos.

1.1 Recubrimiento minimal de un grafo por greedy:

Un recubrimiento, se produce cuando se encuentra un conjunto de vértices de un grafo, los cuales están conectados a la totalidad de aristas del grafo. Evidentemente, un recubrimiento se puede hacer mejor, o peor, dependiendo del número de vértices seleccionados. Por ejemplo: Si en un grafo cogemos todos sus vértices, será un recubrimiento correcto, pero muy poco óptimo. Nosotros pretendemos seleccionar el conjunto óptimo o un conjunto que se le aproxime. Siendo el conjunto con menor número de vértices el minimal, el óptimo.

→ Pseudo-código para encontrar un recubrimiento minimal:

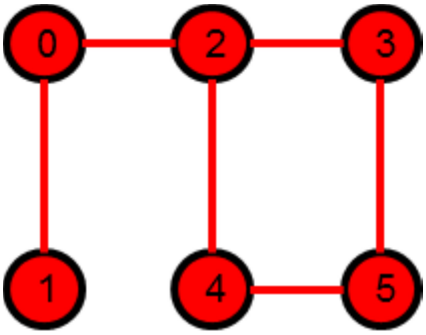
```
INICIO
PARA i=0 HASTA N CON INCREMENTO +1  // Recorrer matriz
   PARA j=0 HASTA N CON INCREMENTO +1
        SI MATRIZ[I][J] == 1 ENTONCES // Primera arista encontrada
                INTRODUCE VERTICES (I,J) EN EL CONJUNTO SOLUCION
                PARA x = 0 HASTA N CON INCREMENTO +1
                BORRAR FILAS DE I y J
        FIN_PARA
                PARA y = 0 HASTA N CON INCREMENTO +1
                BORRAR COLUMNAS DE I y J
        FIN_PARA
        FIN_PARA
FIN_PARA
FIN
```

Usamos una matriz de adyacencia, la cual nos indica qué vértices están conectados a qué aristas e introducimos en un conjunto solución los vértices unidos por la arista en cuestión, recorremos todas las aristas una por una.

→ Elementos del problema:

- Candidatos a seleccionar:
Aristas que unen a dos vértices **- Candidatos seleccionados**
- Función Solución:
Conjunto de nodos que recubren el grafo.
- Función de Factibilidad:
Matriz de adyacencia a cero.
- Función Selección:
Primera arista encontrada en la matriz de adyacencia
- Función Objetivo:
Usar el mínimo número de vértices posibles para cubrir el grafo

Imaginemos el siguiente grafo:



- Esta sería su matriz de adyacencia:

	0	1	2	3	4	5
0	0	1	1	0	0	0
1	1	0	0	0	0	0
2	1	0	0	1	1	0
3	0	0	1	0	0	1
4	0	0	1	0	0	1
5	0	0	0	1	1	0

Como podemos observar en la matriz, si dos vértices están unidos hay un 1, y si no lo están hay un 0. Por ejemplo los vértices 1-0 tienen un 1 porque están unidos entre sí. El programa lo que hace es coger esos 1 y una vez coge uno, pone toda la fila y toda la columna respectiva de ese vértice a 0, ya que todas las uniones que tuviese ese vértice han sido cubiertas.

2. Recubrimiento de un arbol (Greedy)

→ Pseudo-código:

```
while( ! arbolvacio ) {   // Mientras no esté vacío el árbol

        nodo = BuscarHojaMasProfunda() ;      // 1º Buscamos la hoja con mayor profundidad del  árbol
        npadre = BuscarPadre (nodo) ;         // 2º Buscamos el padre del nodo hoja anterior
```

```
        sol.push_back (npadre) ;                    // 3º Nodo padre obtenido al conjunto solución
        BorrarConexionesHijos (npadre);             // 4º Eliminar conexiones con sus nodos hijos
        BorrarConexionNodoPadre (npadre);   // 5º Eliminar conexión del nodo solución anterior con su padre.

}
```

→ Elementos del problema:

- Candidatos a seleccionar:
Hojas de mayor profundidad- **Candidatos seleccionados**
- Solución:
Conjunto de nodos que cumplen el recubrimiento minimal.
- Función de Factibilidad:
Que queden aristas en el árbol
- Función Selección:
Nodo hoja del último nivel del árbol
- Función Objetivo:
Usar el mínimo número de nodos para cubrir el árbol

→ Explicación del algoritmo propuesto como solución al problema del Arbol:

1º Busca una hoja cualquiera del árbol que esté en el último nivel del mismo (NodoHoja)
2º Incluye en el conjunto solución al nodo padre de 'NodoHoja' (NodoPadre)
3º Elimina todas las aristas que van hacia los nodos hijos del NodoPadre
4º Elimina la arista que va de NodoPadre a su propio nodo padre
5º Repetir el proceso hasta finalizar el árbol

Tiene una eficiencia de orden cuadrática, es decir, O(n^2).
Las estructuras de datos utilizadas han sido una Matriz dinámica de adyacencia (como en los grafos) para representar el árbol y una cola FIFO para almacenar el conjunto solución.

3. Diferencias entre un árbol y un grafo:

→ Grafos:
Mediante un algoritmo voraz, es decir, directo y eficiente. Alcanzar la solución óptima no está asegurado, nos tenemos que conformar con una solución que, aunque será aproximada, puede o puede no ser la minimal.

→ Árboles:

A diferencia de los grafos, con los árboles la obtención de la solución minimal si está asegurada. Puesto que los árboles son estructuras jerarquizadas y podemos intuir de mejor manera como hacer el recubrimiento óptimo.

4. Códigos utilizados:

→ Para recubrimiento de grafos:

```
#include <iostream>
using namespace std;
#include <ctime>
#include <cstdlib>
#include <chrono>
#include <cassert>
#include <list>

using namespace std::chrono;

high_resolution_clock::time_point tantes, tdespues;
duration<double> transcurrido;

int main(int argc, char * argv[]){

if (argc != 2)
   {
    cerr << "Formato " << argv[0] << " <tamaño_grafo>" << endl;
    return -1;
   }

int n = atoi(argv[1]);
int u=0; //etiqueta del nodo
int matriz[n][n];
int v1, v2;
list <int> vertice;//Subconjunto solución

/*-------------------------------------------------------*/
//Rellenamos matriz a ceros

for(int i=0; i<n; i++){
   for(int j=0; j<n; j++){
           matriz[i][j] = 0;
   }
}

while (v1!=-1){
    cin >> v1 >> v2;
    matriz[v1][v2]=1;
}

/*-------------------------------------------------------*/
//Matriz rellenada

cout << "\nMATRIZ RELLENADA: \n";
```

```
for (int a=0;a<n;a++){
       for (int b=0;b<n;b++){
          cout << " " << matriz[a][b] << " ";
       }
       cout << "\n";
   }
   cout << "\n\n";

/*-----------------------------------------------------*/

/* Cogemos un arista y sus dos vértices. Posteriormente los quitaremos y volveremos a elegir una arista de manera aleatoria */

tantes = high_resolution_clock::now();

for(int x=0; x<n; x++){
   for(int y=0; y<n; y++){
      if(matriz[x][y]==1){ //Si encuentra la arista

         //Mete los dos vértices subconjunto solución
      vertice.push_back(x);
      vertice.push_back(y);

         for(int d=0; d<n; d++){ //Borrar las conexiones con esos vértices (FILAS)
               matriz[x][d]=0;
               matriz[y][d]=0;
         }

         //Borramos las Columnas
         for(int d=0; d<n; d++){
               matriz[d][x]=0;
               matriz[d][y]=0;
         }

      }
   }
}

tdespues = high_resolution_clock::now();

// Matriz resultante

cout << "MATRIZ MODIFICADA: \n";

for (int a=0;a<n;a++){
       for (int b=0;b<n;b++){
          cout << " " << matriz[a][b] << " ";
       }
       cout << "\n";
   }
   cout << "\n\n";

cout << "Resultado: [";

while (!vertice.empty()){       //Impresion de resultados
   u=vertice.front();
       cout << u << " ";
       vertice.pop_front();
}

cout << "]\n" << endl;

transcurrido = duration_cast<duration<double>>(tdespues - tantes);
```

```
cout << n << "         " << transcurrido.count() << endl;

return 0;

}
```

→ Para recubrimiento de árboles:

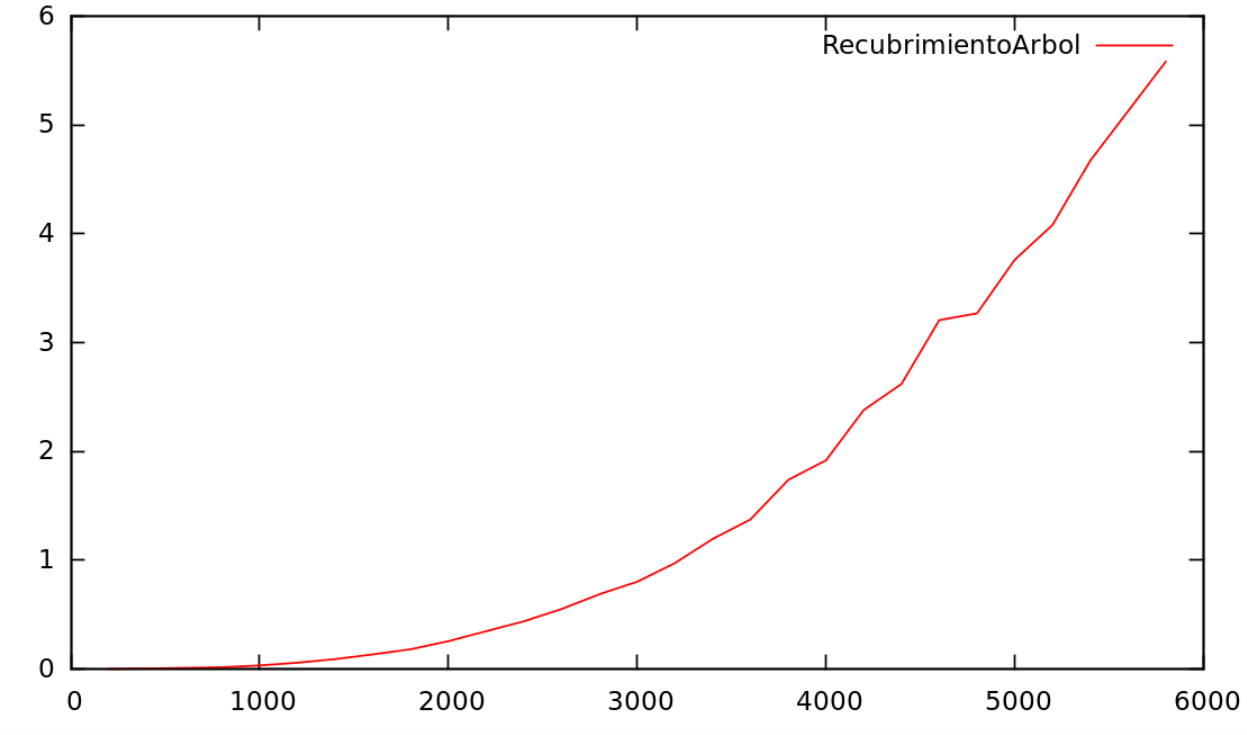
```
#include <iostream>
using namespace std;
//#include <ctime>
#include <cstdlib>
#include <cassert>
#include <list>
#include <chrono>

using namespace std::chrono;

high_resolution_clock::time_point tantes, tdespues;
duration<double> transcurrido;

list<int> cola;

double uniforme() //Genera un número uniformemente distribuido en el
                  //intervalo [0,1) a partir de uno de los generadores
                  //disponibles en C.
{
 int t = rand();
 double f = ((double)RAND_MAX+1.0);
 return (double)t/f;
}


void borrah(int **arbol,int tam, int nodop){

for (int i=0;i<tam;i++)
   arbol[nodop][i]=0;

}

int padre(int **arbol,int tam, int nodoh){

int cont=0;

for (int i=0;i<tam;i++){
   for(int j=0;j<tam; j++){
      if(arbol[i][j]==1)
         cont++;
      if(cont==nodoh)
         return i;
   }
}
return -1;

}

void borracp(int **arbol,int nodo, int tam){
```

```
bool control=true;

for (int i=0;i<tam && control;i++){
   if(arbol[i][nodo]==1){
      arbol[i][nodo]=0;
      control=false;
   }
}

}

int ultimahoja(int **arbol,int tam){

int cont=0;

for (int i=0;i<tam;i++){
   for(int j=0; j<tam;j++){
      if(arbol[i][j]==1)
         cont++;
   }
}

return cont;
}

bool arbolvacio(int **arbol, int tam){

for (int i=0;i<tam;i++){
   for(int j=0;j<tam;j++){
      if(arbol[i][j]==1)
         return false;
   }
}
return true;
}



int main(int argc, char * argv[])
{

  if (argc != 3)
    {
     cerr << "Formato " << argv[0] << " <num_nodes>" <<  " <num_hijos_max>" << endl;
     return -1;
    }

  int numnodes = atoi(argv[1]);
  int maxh=atoi(argv[2]);


srand(time(NULL));
int **v;
v =  new int * [numnodes];
  assert(v);

for (int i = 0; i < numnodes; i++)
      v[i]= new int [numnodes];

int n=1;                //cuenta el número de nodos generados hasta ahora
int i=0;                //etiqueta del nodo
```

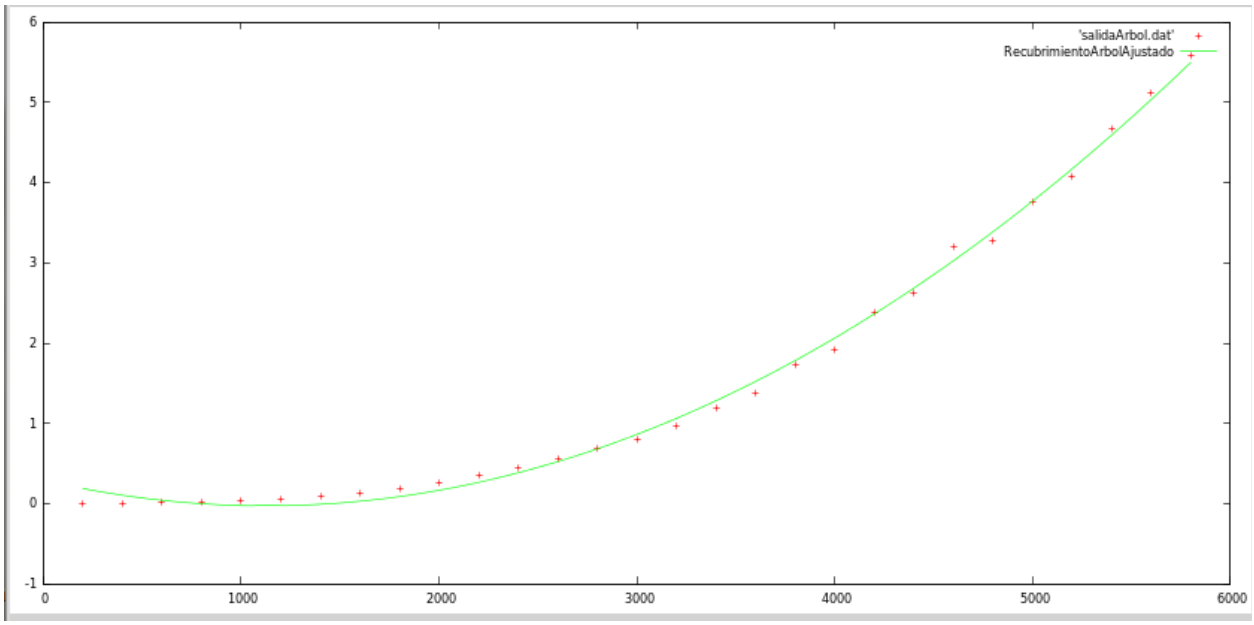
```
cola.push_back(i);                  //es una cola FIFO
while (n < numnodes) {
   i=cola.front();
   cola.pop_front();
   double u=uniforme();
   int ch=1+(int)(maxh*u);                  //entero aleatorio entre 1 y maxh
   if ((ch+n) > numnodes) ch=numnodes-n;          //para no generar más de numnodes nodos
      for (int j=n; j<ch+n; j++) {
          v[i][j]=1;                  //v es la matriz de adyacencia del árbol
          cola.push_back(j);
    }
   n=n+ch;
}

if(numnodes<20){
   while (!cola.empty()) cola.pop_front();

      for (int i=0; i<numnodes; i++) {
         for (int j=0; j<numnodes; j++)
            cout << v[i][j] << " ";
            cout << " " << endl;
   }
}


int nodo,npadre;
list<int> sol; //cola FIFO conjunto solución
tantes = high_resolution_clock::now();
while(!arbolvacio(v,numnodes)){

   nodo=ultimahoja(v,numnodes);         //Obtengo la hoja con mayor profundida del arbol
   npadre=padre(v,numnodes,nodo);  //Obtengo el padre del nodo hoja anterior
   sol.push_back(npadre);      //Incluyo el nodo padre anteriormente obtenido en el conjunto solució
   borrah(v,numnodes,npadre);  //Elimino las conexiones con los hijos para le nodo anteriormente incluido en el conjunto
solucion
   borracp(v,npadre,numnodes); //Elimino la conexión del nodo solucion anterior con su padre.

}
tdespues = high_resolution_clock::now();
transcurrido = duration_cast<duration<double>>(tdespues - tantes);


cout<<"\n\n\tConjunto Solución: ";

   while (!sol.empty()){         //impresion de resultados
      i=sol.front();
         cout<<i<<" ";
         sol.pop_front();
   }

cout<<endl;

cout << numnodes <<" "<< transcurrido.count()<< endl;

}
```

5. Gráficas de tiempos del árbol:

→ Gráfica de recubrimiento del árbol:



→ Gráfica ajustada de recubrimiento de un arbol.


'salidaArbol.dat'
RecubrimientoArbolAjustado
6
5
4
3
2
1
0
-1
0
1000
2000
3000
4000
5000
6000