## horizontal line



Práctica 3: Recubrimiento de un Grafo no Dirigido

2019

Realizado por:

Miguel Ángel Campos Cubillas

Alejandro Pinel Martínez

Guillermo Palomino Sánchez

Pablo Lombardero Ros

Nikita Stetskiy

# Problema: Recubrimiento de un Grafo no Dirigido

Consideramos un grafo no dirigido G = (V,E). Para G, definimos un conjunto U llamado 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 o y ∈ U. Un conjunto de nodos es un recubrimiento minimal de G si es un recubrimiento con el menor número posible de nodos.

# Objetivos

1. Diseñar un Algoritmo Greedy para intentar obtener un recubrimiento minimal de G. Demostrar que el algoritmo es correcto, o dar un contraejemplo.
2. Diseñar un Algoritmo Greedy que obtenga el recubrimiento minimal para el caso particular de grafos que sean árboles.
3. Opcionalmente, realizar un estudio experimental de las diferencias entre los dos algoritmos anteriores cuando se aplican a árboles.

# Algoritmo greedy para el recubrimiento minimal de un grafo

En esta primera parte pretendemos encontrar un algoritmo voraz que sea capaz de encontrar un recubrimiento minimal de G.

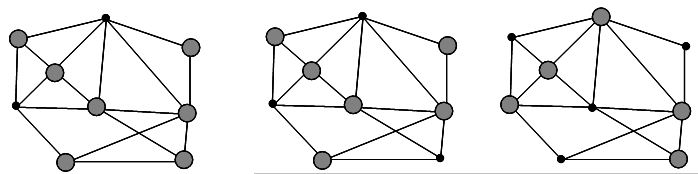


Fig 1: Recubrimiento de G. Fig 2: Recubrimiento minimal de G. Fig 3: Recubrimiento mínimo.

Elementos del algoritmo

* **Conjuntos de candidatos (C) :** vértices del grafo no seleccionados con grado mayor que 0.
* **Conjuntos de aristas no seleccionadas (A) :** aristas que no han sido cubiertas todavía por ningún nodo seleccionado.
* **Conjuntos de candidatos seleccionados (S) :** nodos seleccionados por la función selección.
* **Función solución (FS) :** conjunto de aristas vacío.
* **Función factibilidad (FF) :** determina si es posible completar el conjunto de candidatos seleccionados para alcanzar una solución al problema, en el cual el conjunto de nodos no está vacío.
* **Función selección :** función que selecciona uno de los nodos adyacentes al nodo con menor grado (contando sólo las aristas restantes) de los candidatos excluyendo a aquellos de grado 0.
* **Función objetivo :** elegir el mínimo conjunto de vértices que constituyan un recubrimiento óptimo.

Funcionamiento del Algoritmo

Cuando en un grafo tenemos un nodo de grado 1, significa que ese nodo está unido únicamente a otro nodo mediante una arista. Para envolver esa arista, será necesario siempre tomar ese nodo o su vecino. Ante esta elección, no tiene sentido escoger el nodo de grado 1, pues sólo tomaría una arista, mientras que si tomamos el vecino podemos estar, potencialmente, envolviendo más aristas.

Basado en ese principio, el algoritmo buscará los nodos con menor grado para tomar a sus vecinos. Si el nodo de menor grado tiene solo grado 1, se tomará automáticamente el nodo vecino, si tiene varios vecinos, tomará uno de ellos.

A continuación, eliminará de la lista de aristas todas aquellas que queden cubiertas y eliminará de la lista de candidatos los nodos que tras este proceso se hayan quedado con grado 0.

Mientras queden aristas sin cubrir, repetirá este proceso hasta tener todo el grafo envuelto.

Pseudocódigo

|  |
| --- |
| Recubrimiento minimal (N, B)  S = Ø  C = N //N es Vértices del grafo  A = B //B es el conjunto de aristas del grafo sin envolver  mientras (A != Ø) hacer //Función solución (el conjunto de aristas del grafo está vacío)  x = FS(C) //Seleccionar el adyacente al nodo de menor grado.  C = C - {x} //Lo elimina del conjunto de candidatos.  S = S U {x} //Añadir el vértice al conjunto de seleccionados.  A = A - {x.aristas} //Eliminar las aristas cubiertas del conjunto A.  C = C - {nodos de grado 0 en |C,A|} //Eliminar nodos de grado 0  fin mientras  devolver S  fin |

Demostración de solución óptima

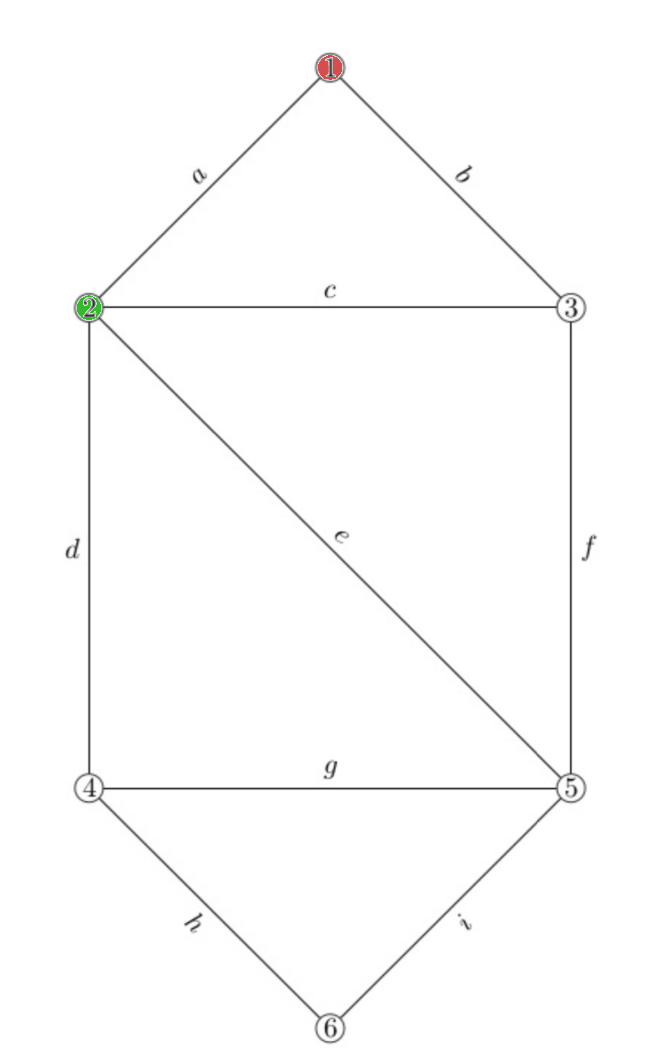
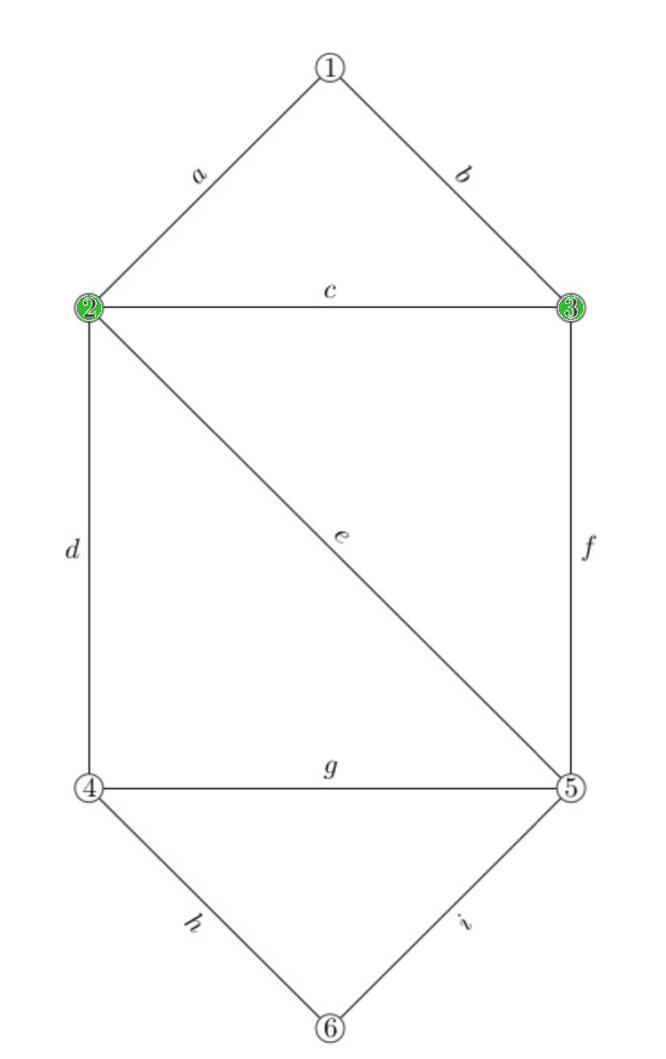
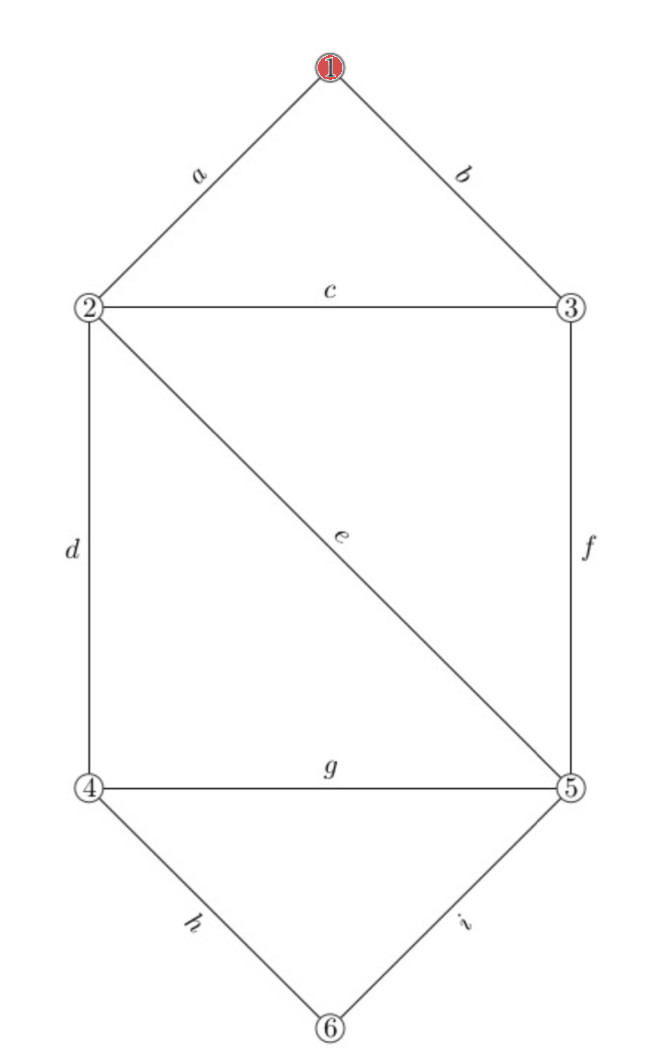
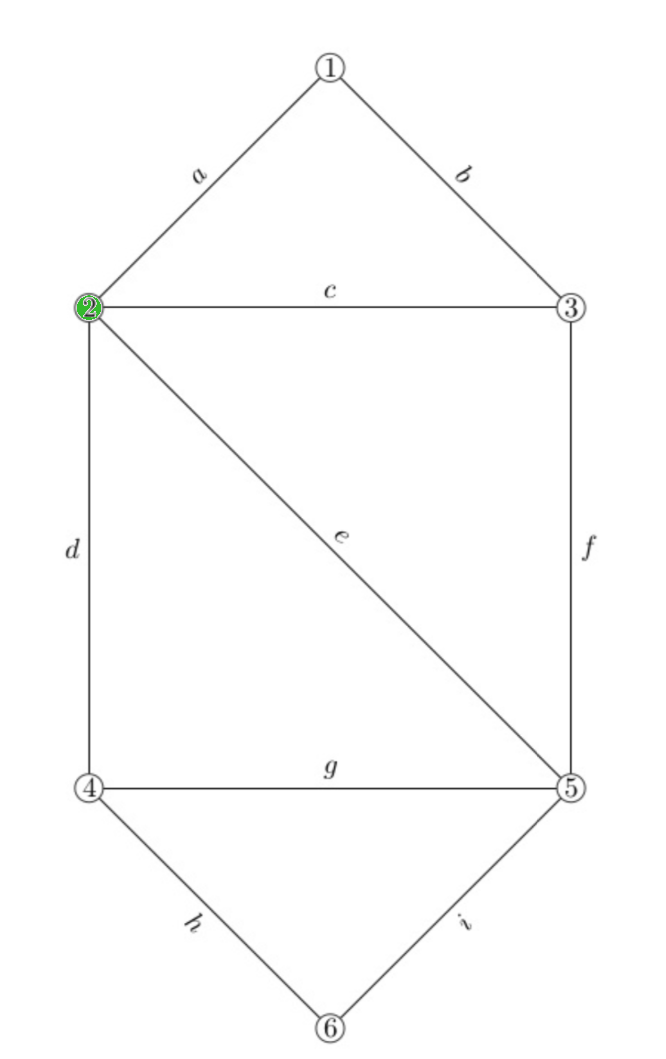
Partimos de que una solución con nuestro algoritmo es S y consideramos que hay una solución S’ mejor. Esta solución cumple que tiene al menos un nodo seleccionado menos, luego |S’| < |S|

Tomemos el nodo p con menor grado del grafo. La arista o aristas que tenga ese nodo han de ser recubiertas para tener la solución y para ello solo hay dos posibilidades, tomar ese nodo o tomar a su adyacente. S tiene al adyacente y como S’ difiere de S debe tomar el propio p. Como el grado de p es menor o igual que el grado del nodo adyacente, el número de aristas cubiertas por S’ es menor o igual que el número de aristas cubiertas por S. Para compensar las aristas no cubiertas por ese nodo será necesario coger un número igual o mayor de nodos que S, lo que nos lleva a una contradicción, pues habíamos supuesto que |S’| < |S|.

Código del Algoritmo

|  |
| --- |
| bool comprobarSiEs0(int i, vector< vector<int> > & B){  bool Es0 = true;  for(int j = 0; i < B.size(); i++){  if(B[j][i] != 0 || B[i][j] != 0){  Es0 = false;  }  }  return Es0;  }  int funcionSeleccion(vector<int> & N, vector< vector<int> > & B){  int x;  int minGrado = 100000;  for(int i = 0; i < N.size() || minGrado == 1; i++){  int grado = 0;  int nodo = N[i];  for(int j = 0; j < B.size(); j++){  if(B[j][nodo] != 0 || B[nodo][j] != 0){  grado++;  }  }  if(grado < minGrado && grado > 0) x = nodo;  }  bool encontrado = false;  for(int i = 0; i < B.size() && !encontrado; i++){  if(B[x][i] != 0 || B[i][x]){  x = i;  encontrado = true;  }  }  return x;  }  vector<int> recubriMin(vector<int> & N, vector< vector<int> > & B, int aristas){  vector<int> S;  while(aristas != 0){  int x = funcionSeleccion(N, B);  for(int i = 0; i < N.size() ; i++){  if(N[i] == x){  N.erase(N.begin()+i);  break;  }  }  S.push\_back(x);  for( int i = 0; i < B.size(); i++){  if(B[i][x] == 1){  aristas--;  B[i][x] = 0;  if(comprobarSiEs0(i, B)){  for(int j = 0; j < N.size(); j++){  if(N[j] == x){  N.erase(N.begin()+j);  break;  }  }  }  }  if(B[x][i] == 1){  aristas--;  B[x][i] = 0;  if(comprobarSiEs0(i, B)){  for(int j = 0; j < N.size(); j++){  if(N[j] == x){  N.erase(N.begin()+j);  break;  }  }  }  }  }  }  return S;  } |

Ejemplo del funcionamiento del Algoritmo

* A continuación se muestra la secuencia que seguiría el algoritmo para encontrar la solución en el siguiente grafo.
* Los vértices en rojo representan los nodos con menor grado en cada iteración, mientras que los verdes representan a los nodos escogidos.
* 

# 

# Algoritmo greedy para el recubrimiento minimal de un árbol

Tratando el problema del recubrimiento de grafos aplicando concretamente a árboles, nos encontramos con que la aproximación greedy sí que nos da un resultado optimal. El algoritmo greedy utilizado sería el siguiente:

|  |
| --- |
| 1. vector<**int**> recubrimientoGreedy(**int** \*\*v, **int** numnodes){  2. **int** cont = 0, hijos = 1, indice = 0, cont\_ant, totales = numnodes, i=0;  3. **bool** escogidos[numnodes];  4. **for** (**int** a=0; a<numnodes; a++)  5. escogidos[a] = **false**;  6. vector<**int**> res; //Conjunto de candidatos solucion, inicialmente vacío.  7. //Los candidatos son los nodos del arbol  8. **while**(totales > 1){ //Repasa el arbol hasta que se hayan borrado todos los nodos excepto la raíz. Es tambien funcion solucion  9. indice = 0;  10. hijos = 1;  11. **while** (cont != 0 || hijos != 0){ //Bucle que recorre arbol por niveles  12. **for**(i=indice; i<indice+hijos;i++){ //Bucle que recorre filas de la matriz de adyacencia  13. cont\_ant = cont;  14. **for**(**int** j=0; j<numnodes; j++) //Bucle que recorre columnas de la matriz de adyacencia  15. **if**(v[i][j] == 1){  16. cont++;  17. }  18. **if**(cont\_ant == cont) //Si es hoja (es decir, no hay ningun 1 en la fila). Esto tambien es función de seleccion.  19. **for** (**int** t=0; t<i; t++)  20. **if** (v[t][i] == 1){ //, escojo al padre  21. v[t][i] = 0; //Borro al nodo de la fila del padre  22. totales--;  23. **for** (**int** b=0; b<t; b++)  24. **if**(v[b][t] == 1){ //Borro al padre de la fila del abuelo  25. v[b][t] = 0;  26. totales--;  27. }  28.  29. **if**(!escogidos[t]) { //Vigilo que el padre no este metido  30. res.push\_back(t);  31. escogidos[t] = **true**; //Lo marco como escogido  32. }  33.  34. }  35. }  36. //Se actualizan los indicadores necesarios para el siguiente nivel del arbol  37. indice = i;  38.  39. hijos = cont;  40. cont = 0;  41. }  42. }  44. **return** res;  45. } |

Funcionamiento del Algoritmo

En lo que respecta al funcionamiento a grandes rasgos del algoritmo, se basa en recorrer el árbol por niveles detectando las hojas del mismo. Una vez ha encontrado una, busca a su padre y lo introduce en el conjunto de nodos resultantes. Acto seguido pasa a podar el árbol por el padre, es decir, elimina tanto al padre como al hijo. Cuando ha terminado el proceso, lo repite desde el principio con el árbol resultante de los borrados anteriores, y así sucesivamente hasta que solo queda el nodo raíz.

En cuanto a las partes típicas de un algoritmo voraz, a continuación se expone con qué parte del código se corresponden cada una de ellas:

* **Conjunto de candidatos (C):** los nodos de cada nivel sobre los que va iterando.
* **Conjunto de seleccionados (S):** el vector res va almacenando las elecciones tomadas, formando al final del proceso la solución del problema.
* **Función solución (FS):** se trata del bucle (while) más externo de todos. Controla el momento en el que se ha alcanzado la solución. En el caso de nuestro algoritmo, lo hará cuando solo quede la raíz.
* **Función de factibilidad (FF):** en nuestro caso, no será necesaria, ya que el conjunto de candidatos siempre acaba formando una solución
* **Función selección:** se trata de la condición de la línea 18, donde lo que se hace es comprobar si hay algún 1 en la fila de la matriz de adyacencia sobre la que se está iterando, es decir, si el nodo que posee la fila tiene algún hijo. Si no lo tiene, es obvio que se trata de una hoja. Luego lo que hace esta función de selección es encontrar una hoja y, a partir de aquí, elimina tanto al nodo hoja como a su padre, añadiendo este último al conjunto de candidatos en caso de que no hubiera sido insertado anteriormente.
* **Función objetivo:** como función objetivo nos encontramos con la impresión por pantalla de los nodos resultantes realizada al final del programa, y fuera de esta función, y cuyo código es el siguiente:

cout << "Los nodos que recubren son: " << endl;

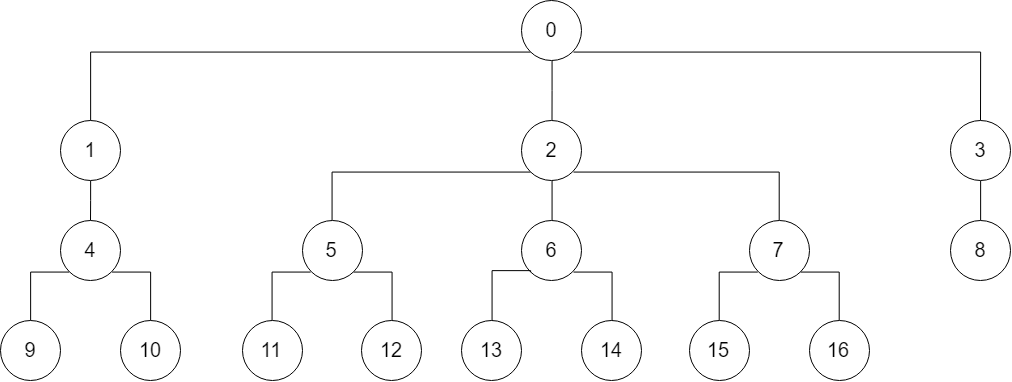
**for** (**int** t=0; t<resultado.size();t++)

cout << resultado[t] << " ";

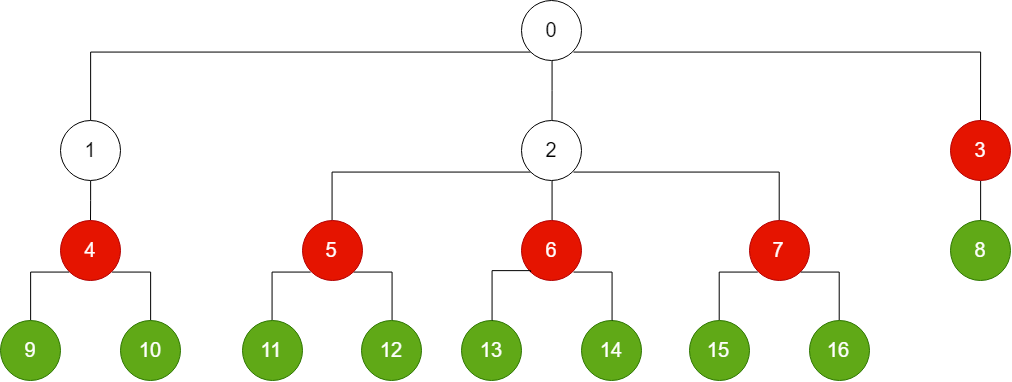
cout << endl;

A continuación, se expone un **ejemplo** para entender mejor el funcionamiento de este algoritmo:

Partimos del siguiente árbol:

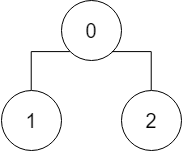


Tras una primera vuelta, se selecciona lo siguiente:

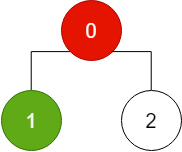


donde los nodos verdes son las hojas y los rojos, sus padres, que son además los nodos que seleccionamos y los puntos por los que se poda el árbol.

Tras aplicar este paso nos quedaría un árbol tal que así:



Seleccionamos esto:



Y nos queda:



En este punto, el tamaño del árbol ya no sería estrictamente mayor que uno y, por tanto, se acaba el proceso, con un conjunto de nodos resultante:

resultado = {0, 3, 4, 5, 6, 7}

Demostración de solución óptima

Lo primero que haremos será tomar lo siguiente como punto de partida: tenemos un recubrimiento minimal U que no contiene ninguna hoja del árbol dado. Esto se puede comprobar fácilmente si tenemos en cuenta que en cualquier cobertura optimal A se pueden reemplazar todas sus hojas por sus padres y se obtiene un conjunto que nunca podrá ser mayor que A.

Ahora, volvamos al recubrimiento minimal U. Digamos que hallamos otro conjunto de nodos de menor tamaño. En cualquier caso, dicho conjunto tendrá que cubrir las aristas que unen las hojas con sus padres, y eso solo puede hacerse de dos maneras: escogiendo las hojas o escogiendo a los padres. Puesto que en U hemos escogido a los padres, en este conjunto B escogeremos a las hojas. En este caso, será imposible que los nodos cubran más aristas a parte de la que conecta a las hojas con sus padres; por otro lado, escoger al padre da como resultado la cobertura de un mínimo de 2 aristas (a no ser que estemos ante un árbol de dos nodos, caso en el que da igual padre que hijo).

Sumando a esto que un conjunto que contiene a los padres de las hojas será siempre de igual o menor tamaño que aquel que contiene a las hojas, llegamos a una contradicción: la supuesta solución óptima, aquella que contiene a las hojas, es forzosamente mayor en tamaño que la solución propuesta, lo que demuestra que el conjunto escogido por el algoritmo greedy es el mejor posible.

# Pruebas de los algoritmos

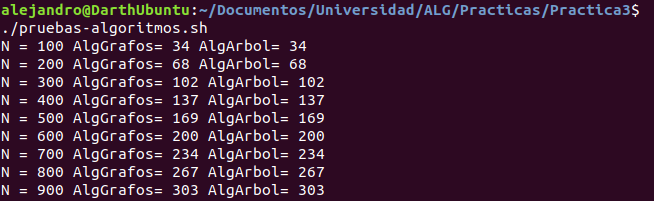
Para comprobar cómo actúan los algoritmos propuestos se va a probar a utilizarlos con una serie de árboles aleatorios.

Para ello utilizaremos el generador de árboles aleatorios, que genera una matriz representando un árbol aleatorio. Admite como parámetros el número de nodos y el número máximo de hijos de cada nodo.

Este árbol con n nodos y n-1 aristas se utiliza tanto para probar el algoritmo de grafos como el de árboles.

|  |
| --- |
| 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;  }  while (!cola.empty()) cola.pop\_front();  vector <int> N(numnodes);  vector < vector <int> > A(numnodes, std::vector<int>(numnodes));  for (int i=0; i<numnodes; i++) {  for (int j=0; j<numnodes; j++)  A[i][j] = v[i][j];  N[i] = i;  }  vector<int> resultado1 = recubriMin(N, A, numnodes - 1);  vector<int> resultado2 = recubrimientoGreedy(v, numnodes);  cout << "N = " << numnodes << " AlgGrafos= " << resultado1.size() << " AlgArbol= " << resultado2.size() << endl;  for(int i=0; i<numnodes; i++)  delete []v[i];  delete []v;  } |

Al probar con una batería de pruebas de 100 a 900 devuelve los siguientes resultados:



Ambos algoritmos devuelven soluciones con los mismos tamaños, ya que ambos devuelven una solución óptima.