Universidad Autónoma Metropolitana Unidad Azcapotzalco
División de Ciencias Básicas e Ingeniería
Licenciatura en Ingeniería en Computación

Proyecto Terminal:
Encajes primitivos de árboles planos

Celene Dorali Alfaro Quintero
208301193

Trimestre 2013 Invierno

Dr. Francisco Javier Zaragoza Martínez
Profesor Titular
Departamento de Sistemas

Índice de contenido

1. Resumen

Para este proyecto se han seleccionando e implementado algoritmos que generan encajes primitivos de árboles. Para ello se emplearon cuatro algoritmos, tres de ellos se ejecutan en tiempo lineal y encuentran encajes primitivos grandes mientras, que el otro algoritmo genera encajes primitivos pequeños pues utiliza la técnica de búsqueda con retroceso que le permite probar todas las posiciones posibles de la rejilla pero tiene un tiempo de ejecución exponencial.

Los cuatro algoritmos reciben como entrada un archivo con todos los árboles posibles de N vértices, los procesan uno por uno y se generan los resultados en un archivo, este archivo contiene el código necesario y listo para ser compilado en LaTeX para poder visualizar gráficamente los resultados obtenidos de cada algoritmo para cada árbol que fue procesado.

2. Objetivo

Diseñar e implementar algoritmos recursivos para encontrar encajes primitivos de árboles planos.

3. Objetivos particulares

1. Implementar tres algoritmos lineales que encuentran encajes primitivos grandes de árboles planos.
2. Diseñar e implementar un algoritmo de búsqueda con retroceso para encontrar encajes

4. Introducción

Una gráfica se define como un par ordenado G = (V, E), donde V es un conjunto finito de vértices y E es un conjunto finito de aristas. Una arista se define como un par no ordenado de vértices E={u,v} donde u, v ∈ V.

En base a la definición anterior, un árbol se define como una gráfica G, que no tiene circuitos y es conexa. Una gráfica es plana si ésta puede ser dibujada en un plano de tal forma que las aristas no se crucen. Todos los árboles son gráficas planas (ver figura 1).

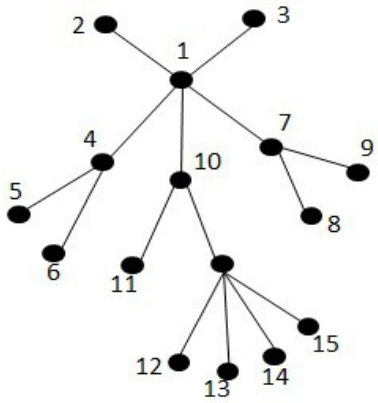

Figura 1. Árbol plano.

Un punto de la rejilla se define como un punto en el plano cuyas coordenadas son enteras. La

rejilla de enteros Z^2, es el conjunto de todos los puntos de la rejilla. Si el segmento de línea que une dos puntos de la rejilla no contiene un tercer punto de Z^2, se dice que los puntos son visibles entre sí. Un segmento de línea es primitivo si sus puntos finales son dos puntos de la rejilla que son visibles entre sí.

Un encaje primitivo es el dibujado mediante segmentos primitivos, en este caso de árboles planos en la rejilla de enteros. Para un árbol plano se va a encontrar la rejilla de tamaño mínimo (es decir, la rejilla cuadrada con el lado más pequeño posible) de forma que se pueda dibujar el árbol utilizando segmentos primitivos. Ver ejemplo de la figura 2.

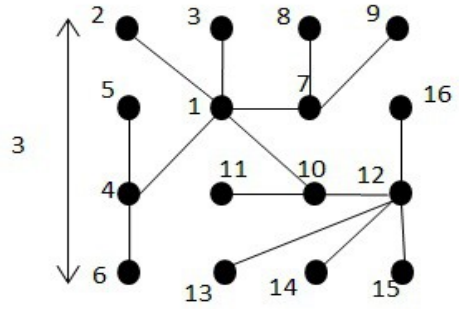

Figura 2. Encaje primitivo del árbol mostrado en la figura 1, en una rejilla de lado 3.

Es importante mencionar que para dibujar el árbol en la rejilla con segmentos de líneas primitivos no se puede incurrir en los casos mostrados en la Figura 3.

Figura 3. A la izquierda, segmentos de líneas cruzados. A la derecha se muestra un segmento de línea pasando encima de un punto de la rejilla.

5. Justificación

Para los algoritmos lineales el encaje primitivo que generen el tamaño de la rejilla será grande, en cambio para el algoritmo que utiliza búsqueda con retroceso el tamaño de la rejilla será mínimo es decir, se busca optimizar el tamaño de la rejilla que se obtiene con el algoritmo.

Dibujar árboles es de suma importancia ya que en diversas áreas es muy común el uso de estas estructuras sólo por mencionar algunas y hacer clara la importancia de tales estructuras mencionaremos algunas áreas en donde podemos encontrar el uso de árboles. Son utilizados en las taxonomías del parentesco entre especies, para el estudio de parentesco entre lenguas, en los

diccionarios, ya que a partir de una palabra se realiza una búsqueda en el árbol para saber si está incluida en el conjunto, en un sistema operativo la jerarquía de archivos que se utiliza se puede representar con la visualización gráfica de la estructura. En los mapas de sitios web así como el historial de navegación también se utiliza este tipo de gráficas. En el área de la biología y la química las gráficas se aplican a los árboles evolutivos, los árboles filogenéticos moleculares, mapas genéticos, vías bioquímicas, en la propagación de enfermedades y funciones de proteínas. Otras áreas de aplicación son sistemas orientados a objetos (navegadores de clase), redes semánticas, programación lógica, en los compiladores, en gráficos de llamadas de subrutina, diagramas de entidad relación que se usan en el diseño de bases de datos. Hasta para dibujar el árbol genealógico de una persona, entre muchas otras aplicaciones. Para cada una de estas aplicaciones al momento de generar un árbol se necesitan ciertas restricciones, es por eso que a este proyecto se le agregaron algunas restricciones para hacerlo más interesante y con fines de investigación. A todas las aplicaciones mencionadas se debe la importancia de poder dibujar árboles o cualquier otro tipo de gráficas aunque en este proyecto sólo nos centramos en los árboles. La implementación y análisis que necesitan estos algoritmos durante el desarrollo de este proyecto requieren habilidades matemáticas y de programación que son imprescindibles para un ingeniero en computación.

En un futuro estos algoritmos se podrían mejorar o se podrían encontrar algoritmos que sean mejores, para analizar los resultado que se obtengan y compararlos con los que se obtengan en este proyecto.

6. Algoritmos

Para este proyecto se desarrollaron cuatro algoritmos, tres de ellos son algoritmos lineales que generan encajes primitivos grandes de árboles planos y el otro algoritmo utiliza la técnica de búsqueda con retroceso para generar encajes primitivos pequeños.

6.1 Algoritmos lineales

El primer algoritmo coloca los vértices del árbol en la rejilla según el nivel en el que se encuentre. Es decir, cuando un vértice tiene hijos estos son colocados en el siguiente nivel de la rejilla.

6.1.1 Primer algoritmo lineal

Algoritmo 1 Pseudocódigo del primer algoritmo lineal que encuentra encajes primitivos grandes.

```
Algoritmo1(vértice en el que comienza el algoritmo k)
visto[k] ← incrementar orden
para todo hijo del vértice k hacer
    si no ha sido visitado el vértice k entonces
        Salida[nivel[v]][vacio(nivel[v])] ← v
        Algoritmo1(v)
    fin si
fin para
```

6.1.2 Segundo algoritmo lineal

El segundo algoritmo toma en cuenta el nivel del vértice del árbol que se desea acomodar en la rejilla y la altura en la que se encuentra, para así colocarlo en la coordenada dada por el nivel y altura.

Algoritmo 2 Pseudocódigo del segundo algoritmo lineal que encuentra encajes primitivos grandes.

```
Algoritmo2 (vértice en el que comienza el algoritmo k)
vértice t
visto[k] ← incrementar orden
para todo hijo del vértice k  hacer
      si no ha sido visitado el vértice k entonces
         Salida[nivel[v]][visto[v] ← v
         Algoritmo2(v)
      fin si
fin para
```

6.1.3 Tercer Algoritmo lineal

Para el tercer algoritmo se considera igual que en el algoritmo anterior el nivel y la altura del vértice del árbol. La diferencia de este algoritmo con respecto al anterior es que el tamaño de la rejilla es menor, pues para colocar el vértice sus coordenadas son el nivel menos la altura y la altura del vértice.

Algoritmo 3 Pseudocódigo del tercer algoritmo lineal que encuentra encajes primitivos grandes.

```
Algoritmo3(vértice en el que comienza el algoritmo k)
Vértice t
visto[k] ← incrementar orden
para todo hijo del vértice k hacer
      si no ha sido visitado el vértice k entonces
          si visto [v]-nivel[v]<=0 entonces
              Salida[nivel[v][vacio(nivel[v])] ← v
               Algoritmo2(v)
          sino entonces
               Salida[nivel[v]][(visto[v]-nivel3[v])] ← v
          fin si
          Algoritmo3(t → dato)
       fin si
fin para
```

6.2 Algoritmo de búsqueda con retroceso

Este algoritmo genera encajes primitivos de los árboles que recibe pero en una rejilla menor que los algoritmos lineales explicados anteriormente. El algoritmo comienza probando con una rejilla de tamaño $\lceil \sqrt{n} \rceil$ y dependiendo de que se logre o no acomodar el árbol en esta rejilla continuará aumentando el tamaño de la rejilla hasta que se logre. Se intentará colocar un vértice

en cada una de las posiciones de dicha rejilla, para cada posición verifica que el segmento que se desea colocar no cruce con algún otro segmento ya colocado o pase sobre un punto que no le corresponde (ver Figura 3), si esto no ocurre entonces coloca el segmento en la posición y continua con otro vértice del árbol, en caso contrario el segmento no se colocará en esa posición y continuará probando con las demás posiciones. Esto se hace hasta que todos los vértices del árbol hayan sido colocados en la rejilla.

Algoritmo 4 Pseudocódigo del algoritmo que controla el tamaño de la rejilla y las posiciones en donde se intentará colocar el vértice.

```
para m ← ⌈ √n ⌉ ; m<n; m ← m+1 hacer
    para x ← 0; x<m; x ← x+1 hacer
        para y ← 0; y<m; y ← y+1 hacer
            p ← Coloca(1,x,y)
            si p==1 entonces regresa m
            fin si
        fin para
    fin para
fin para
```

El algoritmo que se muestra anteriormente necesita de una función que usa la técnica de búsqueda con retroceso, a continuación se muestra el algoritmo de dicha función.

Algoritmo 5 Pseudocódigo del algoritmo que utiliza la técnica de búsqueda con retroceso el cual encuentra encajes primitivos pequeños.

```
Coloca(v, x, y)
  si v > N entonces regresa Falso
  fin si
  si !Vacía(A[x][y]) entonces regresa Falso
  fin si
  si v>=2 entonces
     u ← padre de v
     si !Segmento(u,x,y) entonces regresa Falso
       fin si
       si Cruza(x,y,v) entonces regresa Falso
       fin si
       para i ← 0; i < n; i ← i +1 hacer
           si VW[0][i]==u && VW[1][i]==v entonces
               break
           si  Vacía(VW[0][i]) && Vacía(VW[1][i]) entonces
               VW[0][i] ← u
               VW[1][i] ← v
           fin si
       fin para
 fin si
 A[x][y] ← v
 PosF[0][v] ← x
 PosF[1][v] ← y
 para w ← 0; w < m; w ← w+1 hacer
```

```
        para z ← 0; z < m; z ← z+1 hacer
            p ← Coloca(v+1 , w , z)
            si p=Verdadero entonces regresa Verdadero
            fin si
        fin para
    fin para
    A[x][y] ← 0
    VW[0][i] ← -1
    VW[1][i] ← -1
    PosF[0][v] ← -1
    PosF[1][v] ← -1
    regresa Falso
```

6.2.1 Segmento y cruce

Para el algoritmo anterior se necesitaron dos funciones esenciales para que funcione correctamente, son las funciones de Cruce y Segmento.

La primera función esencial verifica que el segmento a colocar no cruce con alguno otro que ya esté colocado, el segmento a colocar se prueba contra todos los que ya están, regresando verdadero si se cruza con alguno y falso si no cruza, para esta función se usa el algoritmo llamado CCW que es invocado desde la función intersecta(). Se muestra el pseudocódigo para dicha función.

Algoritmo 6 Pseudocódigo del algoritmo que verifica si un segmento se cruza con otro ya colocado.

```
si v==2 entonces regresa Falso
fin si
punto1.x ← x
punto1.y ← y
punto2.x ← xp
punto2.y ← yp
segmento1.puntos1 ← punto1
segmento1.puntos2 ← punto2
para i ← 0; i<n; i ← i+1 hacer
si !Vacía(VW[0][i]) && !Vacía(VW[1][i]) entonces
    point3.x ← PosF[0][VW[0][i]]
    point3.y ← PosF[1][VW[0][i]]
    point4.x ← PosF[0][VW[1][i]]
    point4.x ← PosF[1][VW[1][i]]
    segmento2.puntos1 ← punto3
    segmento2.puntos2 ← punto4
    si x==PosF[0][VW[0][i]] && y==PosF[1][VW[0][i]] entonces continua
    si x==PosF[0][VW[1][i]] && y==PosF[1][VW[1][i]] entonces continua
    si xp==PosF[0][VW[1][i]] && yp==PosF[1][VW[1][i]] entonces continua
    si xp==PosF[0][VW[0][i]] && yp==PosF[1][VW[0][i]] entonces continua
    si  !intersecta(segmento1,segmento2) entonces regresa Verdadero
    fin si
fin si
```

fin para
regresa Falso
El algoritmo CCW es invocado desde intersecta(), se muestra el pseudocódigo de esta función y el de la función CCW [1]:

Algoritmo 7 Pseudocódigo de la función intersecta.

```
intersecta(Segmento l1, Segmento l2)
regresa
((CCW(l1.puntos1,l1.puntos2,l2.puntos1)*CCW(l1.puntos1,l1.puntos2,l2.puntos2))<=0) &&
((CCW (l2.puntos1,l2.puntos2,l1.puntos1) * CCW(l2.puntos1,l2.puntos2,l1.puntos2))<=0)
```

Algoritmo 8 Pseudocódigo del algoritmo CCW.

```
CCW(punto p0, punto p1, punto p2)
dx1 ← p1.x1-p0.x1
dy1 ← p1.y1-p0.y1
dx2 ← p2.x1-p0.x1
dy2 ← p2.y1-p0.y1
si (dx1*dy2>dy1*dx2) entonces regresa +1
si (dx1*dy2<dy1*dx2) entonces regresa -1
si ((dx1*dx2<0) || (dy1*dy2<0)) entonces regresa -1
si ((dx1*dx1+dy1*dy1)<(dx2*dx2+dy2*dy2)) entonces regresa +1
regresa 0
```

La segunda función esencial verifica que el segmento no pase por encima de un punto de coordenadas enteras (ver Figura 3) regresando falso si el segmento no es válido y verdadero para cuando el segmento es válido, esta función utiliza el algoritmo de Euclides el cual verifica si dos números son primos relativos, con esto podemos saber si el segmento pasa por otro punto. La función segmento busca que la resta de los puntos del segmento a colocar se encuentren en el arreglo donde se almacena el resultado del algoritmo de Euclides.

Algoritmo 9 Pseudocódigo del algoritmo que verifica que el segmento sea válido.

```
para i ← 0; i<n; i ← i+1 hacer
    para j ← 0;  i<n; i ← i+1 hacer
        si A[i][j]==u entonces
            aux ← abs(j - y)
            xp ← i
            yp ← j
            aux2 ← abs(i - x)
            para c ← 0;  c<n; c ← c+1 hacer
                si  F[c][0]==aux &&  F[c][1]==aux2 entonces regresa Verdadero
                fin si
            fin para
         fin si
    fin para
fin para
regresa Falso
```

6.3 Generación de árboles

Para generar todos los posibles árboles de N vértices se utilizó una herramienta llamada Sage [2]. Sage es un sistema computacional el cual reúne y unifica bajo un sólo entorno, lenguaje y jerarquía de objetos toda una colección de software matemático.

A continuación se muestra el código que es necesario usar en la plataforma de Sage para generar todos los árboles que se desean.

```
sage: tree_iterator = graphs.trees(N)
sage: for T in tree_iterator:
...       print T.edges(labels=False)
```

Donde N es el número de vértices de los árboles que se desean generar. Sage regresa todos los árboles posibles para el número de vértices que se introdujo en el código anterior, en seguida se muestra el formato en el que Sage da el resultado para árboles de 4 vértices:

```
[(0, 1), (0, 3), (1, 2)]
[(0, 1), (0, 2), (0, 3)]
```

Con base al formato anterior fue necesario diseñar una rutina que generara un archivo con una forma más simple de leer los datos, pues este formato es un poco incómodo al momento de procesar varios árboles.

6.4 LaTeX

Para el resultado de los cuatro algoritmos ya explicados anteriormente se utilizó una función en la que genera el código en LaTeX para ser compilado posteriormente de la ejecución de los programas, el código se genera con base a la matriz en la que se almacena el resultado de los encajes primitivos y la lista de adyacencia en la que se almacena el árbol procesado. Se utiliza el paquete TikZ[3] de LaTeX para poder dibujar los árboles que dan como resultado estos algoritmos.

Los programas generan un archivo con extensión .tex y una vez compilado se pueden observar los resultados generando un PDF con el uso de LaTeX.

7. Ejemplo

Para el caso de n=12 tenemos un total de 551 árboles, los cuatro algoritmos los procesan obteniendo un resultado para cada árbol. En este ejemplo sólo se mostrará el resultado que se obtiene para un árbol de esos 551, esto para poder observar y comparar los resultados que genera cada algoritmo.

Para los tres algoritmos lineales los resultados para este árbol se muestran en la Figura 4.

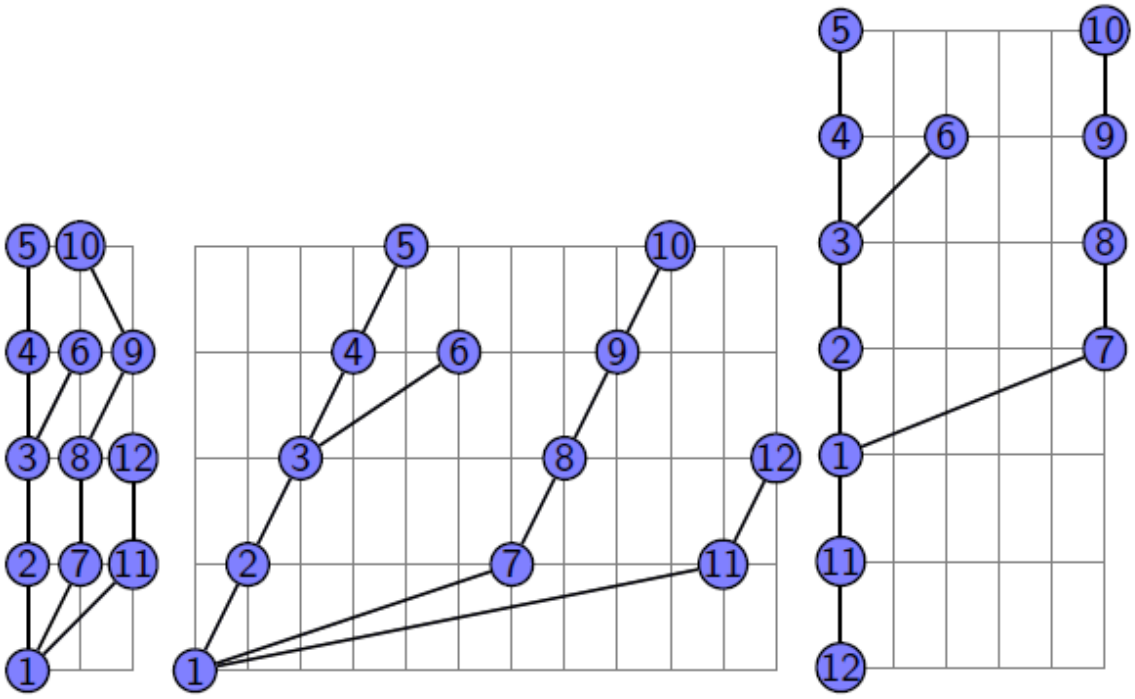

Figura 4. Resultado de los tres algoritmos lineales para un árbol de 12 vértices.

Ahora para el algoritmo que utiliza búsqueda con retroceso el resultado para el mismo árbol de los algoritmos anteriores se muestra en la Figura 5.

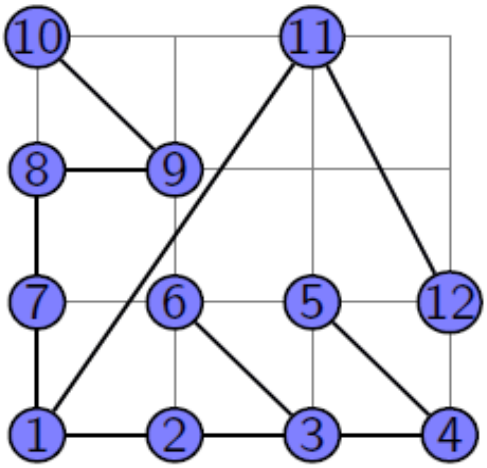

Figura 5. Resultado del algoritmo de búsqueda con retroceso para un árbol de 12 vértices.

Como se puede observar, los tres algoritmos lineales utilizan más espacio en la rejilla comparados con el algoritmo de búsqueda con retroceso. El primer algoritmo utilizó una rejilla de tamaño de 2x4, el segundo algoritmo una rejilla de tamaño de 11x4, el tercer algoritmo una rejilla de tamaño de 5x6, mientras que el algoritmo de búsqueda con retroceso utilizó una rejilla de tamaño 3x3.

Durante la ejecución del algoritmo de búsqueda con retroceso se logró detectar árboles que requerían más tiempo para encontrar una solución debido a su estructura, a continuación se muestran algunos (ver Figuras 6 y 7).

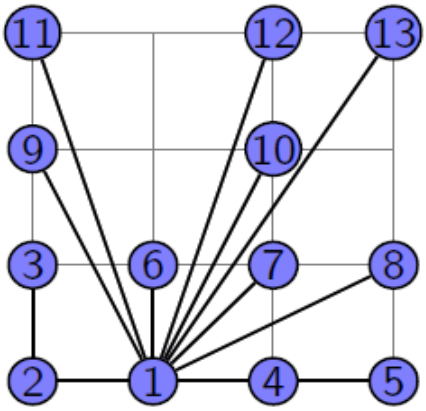

Figura 6. Resultado para un árbol de 13 vértices.

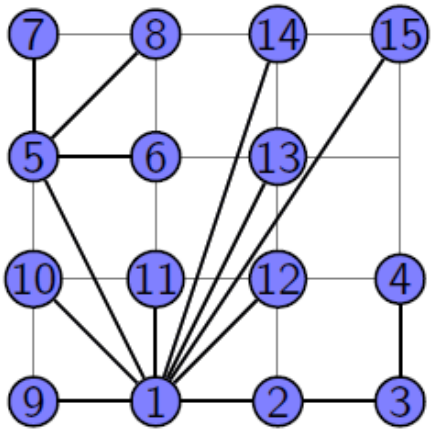

Figura 7. Resultado para un árbol de 15 vértices.

Para los encajes primitivos que se muestran con 13 y 15 vértices el programa requiere de varios minutos para encontrar una solución. Para el caso de 13 vértices el tiempo aproximado fue de 10 minutos, mientras que para el caso de 15 vértices se requirió un tiempo de 15 a 20 minutos aproximadamente.

8. Resumen de los resultados obtenidos

Se obtuvieron resultados para $2 \leq N \leq 19$, los rangos, el número de árboles que fueron procesados para cada caso y el tiempo de ejecución para los algoritmos se muestran en la siguiente tabla.

n	Rango				Número de árboles	Tiempo de ejecución		Tiempo promedio para cada árbol	
	Algoritmo 1	Algoritmo 2	Algoritmo 3	Búsqueda con retroceso		Algoritmos lineales	Búsqueda con retroceso	Algoritmos lineales	Búsqueda con retroceso
2	1 – 2	1 – 2	1 – 2	1 – 2	1	----	----	----	----
3	2 – 2	2 – 3	2 – 2	2 – 2	1	----	----	----	----
4	3 – 3	3 – 4	3 – 3	2 – 2	2	----	----	----	----
5	3 – 4	3 – 5	3 – 4	2 – 3	3	0m0.003s	0m0.002s	1.00ms	0.67ms
6	4 – 5	4 – 6	4 – 5	2 – 3	6	0m0.004s	0m0.004s	0.67ms	0.67ms
7	4 – 6	4 – 7	4 – 6	3 – 3	11	0m0.005s	0m0.009s	0.45ms	0.81ms
8	5 – 7	5 – 8	5 – 7	3 – 3	23	0m0.008s	0m0.020s	0.34ms	0.86ms
9	5 – 8	5 – 9	5 – 8	3 – 3	47	0m0.016s	0m0.213s	0.34ms	0.45ms
10	6 – 9	6 – 10	6 – 9	3 – 4	106	0m0.023s	0m0.096s	0.21ms	0.90ms
11	6 – 10	6 – 11	6 – 10	3 – 4	235	0m0.086s	0m25.130s	0.36ms	106.93ms
12	7 – 11	7 – 12	7 – 11	4 – 4	551	0m0.186s	1m33.763s	0.33ms	170.16ms
13	7 – 12	7 – 13	7 – 12	4 – 4	1301	0m0.428s	27m33.068s	0.32ms	127.06ms
14	8 – 13	8 – 14	8 – 13	4 – 4	3159	0m2.120s	56m13.192s	0.67ms	1067.80ms
15	8 – 14	8 – 15	8 – 14	4 – 4	7741	0m3.239s	4h36m3.492s	0.41ms	1860.60ms
16	9 – 15	9 – 16	9 – 15	4 – 5	19320	0m9.234s	32h30m3.15s	0.47ms	6056.67ms
17	9 – 16	9 – 17	9 – 16	---	48629	0m27.213s	---	0.55ms	---
18	10 – 17	10– 18	10 – 17	---	123867	1m26.028s	---	0.69ms	---
19	10 – 18	10 – 19	12– 18	---	317955	4m13.609s	---	0.79ms	---

Tabla 1. Resumen de los resultados obtenidos.

Como se observa en la anterior (Tabla 1), los tres algoritmos lineales son muy rápidos y es posible ejecutar el programa hasta para 19 vértices procesando un total de 317955 árboles en menos de 5 minutos. Mientras que para el algoritmo de búsqueda con retroceso no se logró ejecutar para 17 vértices pues el tiempo de ejecución aumenta considerablemente, es por eso que en la tabla no se muestra el tiempo de ejecución a partir de este número.

En la Figura 8 se muestra una gráfica de N contra el tiempo promedio por árbol de los algoritmos lineales, junto con la recta de mínimos cuadrados.

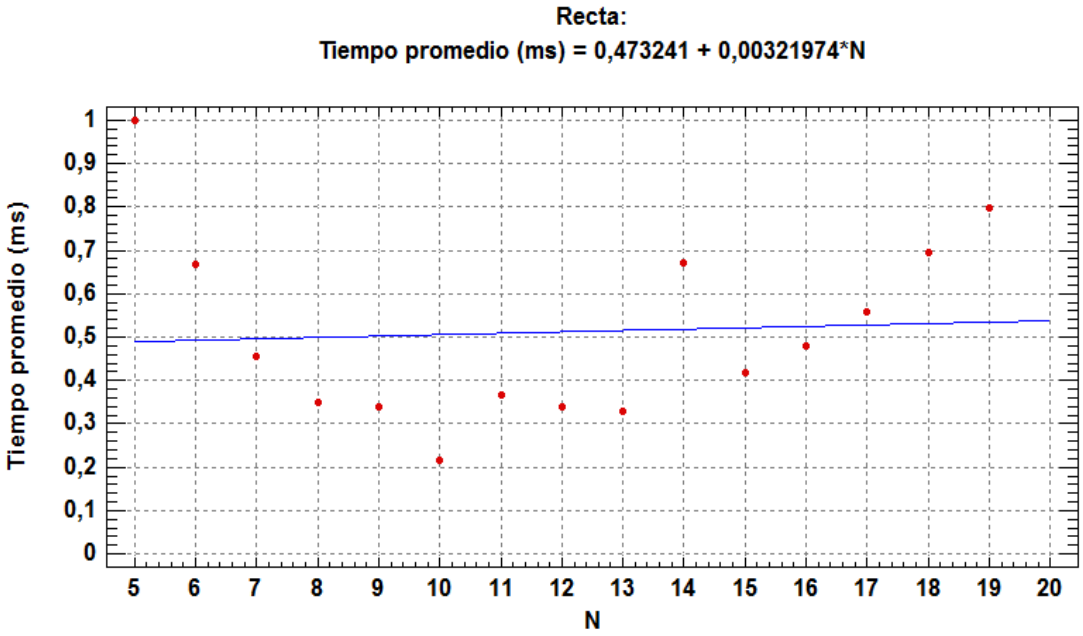

Figura 8. Gráfica de los algoritmos lineales.

En la Figura 9 se muestra una gráfica de N contra el tiempo promedio por árbol del algoritmo de búsqueda con retroceso, junto con una curva exponencial que ajusta el conjunto de datos.

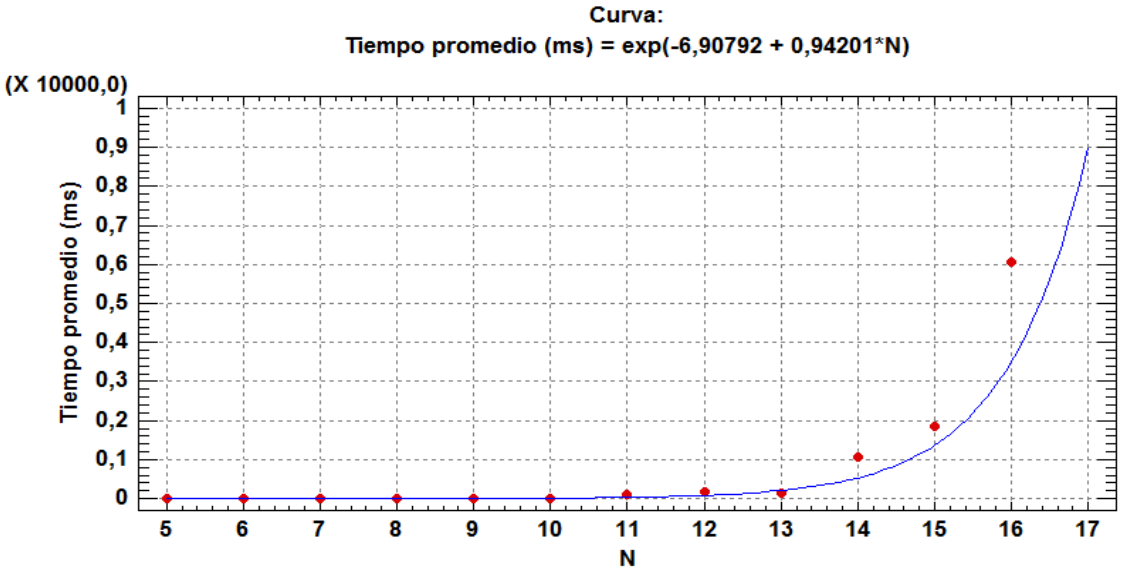

Figura 9. Gráfica del algoritmo de búsqueda con retroceso.

9. Conclusiones

Como ya se mencionó anteriormente, los tres algoritmos lineales son muy rápidos y pueden procesar muchos árboles en poco tiempo pero los encajes primitivos que se generan para cada árbol son grandes comparados con el algoritmo de búsqueda con retroceso. El problema de este algoritmo es que mientras aumente el número de árboles que se desean procesar el tiempo de ejecución aumenta rápidamente ya que tarda mucho más en encontrar la solución pues el uso de la técnica de búsqueda con retroceso le permite probar todas las posiciones de la rejilla y así encontrar que el tamaño de la rejilla sea menor que en los otros algoritmos, en casi todos los casos probados podemos observar en la tabla que el tamaño de la rejilla es de $\lceil \sqrt{n} \rceil$ mientras que el tamaño de las rejillas para los otros algoritmos varía entre n y n-1, la diferencia de los tamaños es considerable pero a cambio de esto se necesita mucho más tiempo en la ejecución.

Al momento de la ejecución del algoritmo de búsqueda con retroceso se logró observar que el tiempo que tardaba para algunos árboles variaba demasiado pues algunos árboles eran procesados muy rápidamente mientras que otros tardaban varios minutos.

10. Bibliografía

[1] R. Sedgewick, "Elementary geometric methods", *Algorithms in C*, pp. 348 – 352.

[2] *Sage: Open Source Mathematics Software,* http://www.sagemath.org/, consultada el 31 de marzo de 2013.

[3] *TikZ and PGF examples*, http://www.texample.net/tikz/examples/, consultada el 5 de abril de 2013.

[4] L. Joyanes, "Estructuras de datos avanzadas", *Algoritmos y estructuras de datos. Una perspectiva en C,* pp. 415 – 572.

11. Código en C de los tres algoritmos lineales

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#define N 20
typedef int Item;
typedef struct elemento{
  int dato;
  struct elemento* siguiente;
}Nodo;

void inserta(Nodo** cabeza, Item entrada);
void visita(int k);
void Algoritmo_1(int k);
void Algoritmo_2(int k);
void Algoritmo_3(int k);
void Ancho(unsigned short int ad[N][N]);
void NivelF(unsigned short int ad[N]);
void imprimir(unsigned short int ad[N][N]);
void Latex(unsigned short int A1[N][N],unsigned short int A2[N][N],unsigned short int A3[N][N]);
Nodo* crearNodo(Item x),**cabeza,*ptr;
unsigned short int ** CrearM(int n,int j);
unsigned short int * CrearA(int n);
Nodo** CrearN(int n);
void Algoritmos(unsigned short int a[N][2]);
Item d;
FILE *Algoritmo1;
int
n=0,orden=0,niv=1,orden2=0,AnchoF=0,nivelF=0,AF1=0,AF2=0,AF3=0,NF1=0,NF2=0,NF3=0;
unsigned short int visto[N],visto2[N],nivel[N],Salida[N][N],Salida2[N][N],Salida3[N]
[N],nivel3[N];

int main(){
int sal=0,datoa=0,datob=0,auxn=0,i=0,cont=0,arbol=0,ab,cd;
unsigned short int a[N][2];
FILE *entrada=fopen("EntradaA.txt","r");
Algoritmo1=fopen("Algoritmos.tex","w");
fprintf(Algoritmo1,"\n\\documentclass{beamer}");
fprintf(Algoritmo1,"\n\\usepackage{tikz}");
fprintf(Algoritmo1,"\n\\begin{document}");
fprintf(Algoritmo1,"\n\\tikzstyle{vertice} = [circle,inner sep=0pt,minimum size=4mm,thick]");
fprintf(Algoritmo1,"\n\\tikzstyle{negro} = [vertice,draw=black,fill=blue!50]");
if(entrada==NULL){
   perror("Error al abrir archivo.");
   return -1;
}
sal=fscanf(entrada,"%d",&n);
auxn=n;
while(auxn==n){
```

```
 for(ab=0;ab<N;ab++)
      for(cd=0;cd<2;cd++)
       a[ab][cd]=0;
cont=0;
arbol++;
printf("Arbol %d\n",arbol);
   for(i=1;i<n;i++){
      fscanf(entrada,"%d %d",&datoa,&datob);
      a[cont][0]=datoa+1;
      a[cont][1]=datob+1;
      cont++;
   }
   Algoritmos(a);
   fscanf(entrada,"%d",&auxn);
   free(cabeza);
   free(ptr);
}
fprintf(Algoritmo1,"\n\\end{document}");
fclose(entrada);
fclose(Algoritmo1);
return 0;
}

void Algoritmos(unsigned short int a[N][2]){
int
k=0,i=0,j=0,aux1=30000000,aux11=30000000,aux2=30000000,aux22=30000000,aux3=30000000,
aux33=30000000,cd,de;
unsigned short int aux[N][N], auxS2[N][N],auxS3[N][N];
    cabeza=CrearN(n+1);
    for(i=0;i<N;i++)
      for(j=0;j<N;j++){
       aux[i][j]=0;
       nivel3[j]=0;
          auxS2[i][j]=0;
          auxS3[i][j]=0;
          }
    for(i=0;i<n;i++){
       inserta(&cabeza[a[i][0]],a[i][1]);
       inserta(&cabeza[a[i][1]],a[i][0]);
    }
    for(i=1;i<=n;i++){
       orden=0;
          for(cd=0;cd<N;cd++)
        for(de=0;de<N;de++){
          Salida[cd][de]=0;
          visto[de]=0;
             visto2[de]=0;
             nivel[de]=0;
          }
       nivel[i]=1;
```

```
 visita(i);
//Algoritmo 1
Salida[1][1]=i;
Algoritmo_1(i);
Ancho(Salida);
NivelF(nivel);
if(abs(AnchoF-nivelF)<aux1 && (AnchoF*nivelF)<aux11){
   aux1=abs(AnchoF-nivelF);
   aux11=AnchoF*nivelF;
   AF1=AnchoF;
      NF1=nivelF;
   for(j=1;j<=n;j++)
     for(k=1;k<=n;k++)
        aux[j][k]=Salida[j][k];
}
//Algoritmo 2
for(cd=0;cd<N;cd++)
   for(de=0;de<N;de++){
      Salida2[cd][de]=0;
         visto2[de]=0;
     }
Salida2[1][1]=i;
Algoritmo_2(i);
Ancho(Salida2);
if(abs(AnchoF-nivelF)<aux2 && (AnchoF*nivelF)<aux22){
   aux2=abs(AnchoF-nivelF);
   aux22=AnchoF*nivelF;
   AF2=AnchoF;
      NF2=nivelF;
   for(j=1;j<=n;j++)
     for(k=1;k<=n;k++)
        auxS2[j][k]=Salida2[j][k];
}
//Algoritmo 3
     for(cd=0;cd<N;cd++)
   for(de=0;de<N;de++){
      Salida3[cd][de]=0;
         visto2[de]=0;
     }
Salida3[1][1]=i;
for(j=0;j<=n;j++)//Se  necesita usar desde el nivel 0
     nivel3[j]=nivel[j]-1;
Algoritmo_3(i);
Ancho(Salida3);
if(abs(AnchoF-nivelF)<aux3 && (AnchoF*nivelF)<aux33){
   aux33=AnchoF*nivelF;
   aux3=abs(AnchoF-nivelF);
   AF3=AnchoF;
      NF3=nivelF;
   for(j=1;j<=n;j++)
```

```
          for(k=1;k<=n;k++)
             auxS3[j][k]=Salida3[j][k];
       }
    }
     Latex(aux,auxS2,auxS3);
}

unsigned short int* CrearA(int n){
  unsigned short int *A,i;
  A=(unsigned short int*)malloc(sizeof(unsigned short int)*n);
  for(i=0;i<n;i++)
       A[i]=0;
return A;
}

Nodo** CrearN(int n){
  Nodo **A;
  unsigned short int i;
  A=(Nodo**)malloc(sizeof(Nodo*)*n);
  for(i=0;i<n;i++)
    A[i]=NULL;
return A;
}

unsigned short int** CrearM(int n, int j){
  unsigned short int **A,i,k;
  A=(unsigned short int**)malloc(sizeof(unsigned short int*)*n);
  for (i=0;i<n;i++)
     A[i]=(unsigned short int*)malloc(sizeof(unsigned short int)*n);
  for(i=0;i<n;i++)
     for(k=0;k<n;k++)
       A[i][k]=0;
return A;
}

void Latex(unsigned short int A1[N][N],unsigned short int A2[N][N],unsigned short int A3[N][N]){
int i=0,k=0;
fprintf(Algoritmo1,"\n\\begin{frame} \\frametitle{Los tres algoritmos}");
fprintf(Algoritmo1,"\n\\begin{center}");
fprintf(Algoritmo1,"\n\\begin{tikzpicture}[xscale=0.5]");
fprintf(Algoritmo1,"\n\\draw [help lines] (0,0) grid(%d,%d);",AF1-1,NF1-1);
 for(i=1;i<=n;i++)
  for(k=1;k<=n;k++)
     if(A1[i][k]!=0)
      fprintf(Algoritmo1,"\n\\node[negro] (%d) at (%d,%d) {%d};",A1[i][k],k-1,i-1,A1[i][k]);
   for(i=1;i<=n;i++){
      for(k=0,ptr=cabeza[i];ptr;){
        fprintf(Algoritmo1,"\n\\draw[negro] (%d) -- (%d);",i,ptr->dato);
        k++;
        ptr=ptr->siguiente;
```

```
      }
    }
fprintf(Algoritmo1,"\n\\end{tikzpicture}");
fprintf(Algoritmo1,"\n\\begin{tikzpicture}[xscale=0.5]");
fprintf(Algoritmo1,"\n\\draw [help lines] (0,0) grid(%d,%d);",AF2-1,NF2-1);
 for(i=1;i<=n;i++)
   for(k=1;k<=n;k++)
     if(A2[i][k]!=0)
       fprintf(Algoritmo1,"\n\\node[negro] (%d) at (%d,%d) {%d};",A2[i][k],k-1,i-1,A2[i][k]);
    for(i=1;i<=n;i++){
       for(k=0,ptr=cabeza[i];ptr;){
         fprintf(Algoritmo1,"\n\\draw[negro] (%d) -- (%d);",i,ptr->dato);
         k++;
         ptr=ptr->siguiente;
       }
    }
fprintf(Algoritmo1,"\n\\end{tikzpicture}");
fprintf(Algoritmo1,"\n\\begin{tikzpicture}[xscale=0.5]");
fprintf(Algoritmo1,"\n\\draw [help lines] (0,0) grid(%d,%d);",AF3-1,NF3-1);
 for(i=1;i<=n;i++)
   for(k=1;k<=n;k++)
     if(A3[i][k]!=0)
       fprintf(Algoritmo1,"\n\\node[negro] (%d) at (%d,%d) {%d};",A3[i][k],k-1,i-1,A3[i][k]);
    for(i=1;i<=n;i++){
       for(k=0,ptr=cabeza[i];ptr;){
         fprintf(Algoritmo1,"\n\\draw[negro] (%d) -- (%d);",i,ptr->dato);
         k++;
         ptr=ptr->siguiente;
       }
    }
fprintf(Algoritmo1,"\n\\end{tikzpicture}");
fprintf(Algoritmo1,"\n\\end{center}");
fprintf(Algoritmo1,"\n\\end{frame}");
A1=NULL;
A2=NULL;
A3=NULL;
}

void Ancho(unsigned short int ad[N][N]){
  int i,j,aux2=0;
   AnchoF=0;
   for(i=1;i<=n;i++){
     if(aux2>AnchoF)
         AnchoF=aux2;
      aux2=0;
       for(j=1;j<=n;j++)
          if(ad[i][j]!=0){
              aux2=j;
          }
   }
```

```
}

void NivelF(unsigned short int ad[N]){
    int i;
    nivelF=ad[0];
    for(i=0;i<=n;i++)
      if (nivelF<ad[i])
        nivelF=ad[i];
}

void inserta(Nodo** cabeza, Item entrada){
    Nodo *ultimo;
    ultimo=*cabeza;
    if(ultimo==NULL)
     *cabeza=crearNodo(entrada);
    else {
      for(;ultimo->siguiente;)
         ultimo=ultimo->siguiente;
      ultimo->siguiente=crearNodo(entrada);
    }
}

Nodo* crearNodo(Item x){
   Nodo *a;
   a=(Nodo*)malloc(sizeof(Nodo));
   a->dato=x;
   a->siguiente=NULL;
   return a;
}

void visita(int k){
  Nodo *t;
  visto[k]=++orden;
  for(t=cabeza[k];t!=NULL;t=t->siguiente){
     if(visto[t->dato]==0)
       nivel[t->dato]=nivel[k];
     if(visto[t->dato]==0){
        nivel[t->dato]=nivel[k]+1;
        visita(t->dato);
     }
  }
}

int vacio(int nivel){
unsigned short int i;
for(i=1;i<n;i++){
  if(Salida[nivel][i]==0)
    return i;
}
return -1;
```

```
}

void Algoritmo_1(int k){
  Nodo *t;
  visto2[k]=++orden2;
  for(t=cabeza[k];t!=NULL;t=t->siguiente){
     if(visto2[t->dato]==0){
          Salida[nivel[t->dato]][vacio(nivel[t->dato])]=t->dato;
          Algoritmo_1(t->dato);
     }
  }
}

void Algoritmo_2(int k){
  Nodo *t;
  visto2[k]=++orden2;
  for(t=cabeza[k];t!=NULL;t=t->siguiente){
     if(visto2[t->dato]==0){
          Salida2[nivel[t->dato]][visto[t->dato]]=t->dato;
          Algoritmo_2(t->dato);
     }
  }
}

void Algoritmo_3(int k){
  Nodo *t;
  visto2[k]=++orden2;
  for(t=cabeza[k];t!=NULL;t=t->siguiente){
     if(visto2[t->dato]==0){
          if((visto[t->dato]-nivel3[t->dato])<=0)
            Salida3[nivel3[t->dato]][vacio(nivel[t->dato])]=t->dato;
          else
            Salida3[nivel[t->dato]][(visto[t->dato]-nivel3[t->dato])]=t->dato;
     Algoritmo_3(t->dato);
     }
  }
}
```

12. Código en C del algoritmo de búsqueda con retroceso

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#define N 17

typedef int Item;
typedef struct elemento{
 int dato;
 struct elemento* siguiente;
```

```
}Nodo;

struct point{int x1,y1; char c;};
struct line{struct point p1,p2;};

void inserta(Nodo** cabeza, Item entrada);
void visita(int k);
void Ancho(int ad[N][N]);
void Alto(int ad[N][N]);
void Latex(int A1[N][N]);
Nodo* crearNodo(Item x),**cabeza,*ptr;
Item d;
FILE *salida, *Algoritmo1;
int n,Aa[N][N],F[(N+1)*(N+1)][2],p,xp,yp,PosF[N][N],VW[N]
[N],m,pa,p,AnchoF=0,nivelF=0,nivel[N],visto[N],orden,nn,B[N][N];

int Euclides(int a, int b){
  if(b==0)
     return a;
  return Euclides(b,a%b);
}

Nodo** CrearN(int n){
   Nodo **A;
   int i;
   A=(Nodo**)malloc(sizeof(Nodo*)*n);
   for(i=0;i<n;i++)
     A[i]=NULL;
return A;
}

int CCW(struct point p0, struct point p1, struct point p2){
int dx1,dx2,dy1,dy2;
dx1=p1.x1-p0.x1;
dy1=p1.y1-p0.y1;
dx2=p2.x1-p0.x1;
dy2=p2.y1-p0.y1;
if(dx1*dy2>dy1*dx2) return +1;
if(dx1*dy2<dy1*dx2) return -1;
if((dx1*dx2<0) || (dy1*dy2<0)) return -1;
if((dx1*dx1+dy1*dy1)<(dx2*dx2+dy2*dy2)) return +1;
return 0;
}

int intersect(struct line l1, struct line l2){
return((CCW(l1.p1,l1.p2,l2.p1)*CCW(l1.p1,l1.p2,l2.p2))<=0) &&
((CCW(l2.p1,l2.p2,l1.p1)*CCW(l2.p1,l2.p2,l1.p2))<=0);
}

int Cruza(int x,int y,int v){
```

```
int vvv;
struct line segmento1,segmento2;
struct point p11,p22,p33,p44;
if(v==2) return 1;
p11.x1=x;
p11.y1=y;
p11.c='1';
p22.x1=xp;
p22.y1=yp;
p22.c='2';
segmento1.p1=p11;
segmento1.p2=p22;
for(vvv=0;vvv<n;vvv++){
   if(VW[0][vvv]>0 && VW[1][vvv]>0){
     p33.x1=PosF[0][VW[0][vvv]];
     p33.y1=PosF[1][VW[0][vvv]];
     p33.c='3';
     p44.x1=PosF[0][VW[1][vvv]];
     p44.y1=PosF[1][VW[1][vvv]];
     p44.c='4';
     segmento2.p1=p33;
     segmento2.p2=p44;
     if(x==PosF[0][VW[0][vvv]] && y==PosF[1][VW[0][vvv]]) continue;
     if(x==PosF[0][VW[1][vvv]] && y==PosF[1][VW[1][vvv]]) continue;
     if(xp==PosF[0][VW[1][vvv]] && yp==PosF[1][VW[1][vvv]]) continue;
     if(xp==PosF[0][VW[0][vvv]] && yp==PosF[1][VW[0][vvv]]) continue;
     if(intersect(segmento1,segmento2)==1) return 0;
   }
}
return 1;
}

int Segmento(int u, int x,int y){
int i,j,aux,aux2,c;
for(i=0;i<nn+1;i++)
  for(j=0;j<nn+1;j++)
    if(Aa[i][j]==u){
      aux=fabs(j-y);
      xp=i;
      yp=j;
      aux2=fabs(i-x);
      for(c=0;c<pa;c++)
        if((F[c][0]==aux && F[c][1]==aux2) || (F[c][0]==aux2 && F[c][1]==aux))
        return 1;
     }
return 0;
}

int Coloca(int v,int x,int y){
int u=0,i=0,w=0,z=0,b=0,zz=0,cc=0,j;
```

```
   if(v>n) return 1;
   if(Aa[x][y]!=0) return 0;
   if(v>=2){
     ptr=cabeza[v];
     u=ptr->dato;
     if(Segmento(u,x,y)==0) return 0;
     if(Cruza(x,y,v)==0) return 0;
     i=0;
     while(cc!=1){
       if(VW[0][i]==u && VW[1][i]==v)
          cc=1;
       else if(VW[0][i]==-1 && VW[1][i]==-1){
          VW[0][i]=u;
          VW[1][i]=v;
          cc=1;
       }
       i++;
      }
      }
      Aa[x][y]=v;
      PosF[0][v]=x;
      PosF[1][v]=y;
      for(w=0;w<m;w++)
        for(z=0;z<m;z++){
              p=Coloca(v+1,w,z);
           if(p==1) return 1;
        }
     Aa[x][y]=0;
     VW[0][i]=-1;
     VW[1][i]=-1;
     PosF[0][v]=-1;
     PosF[1][v]=-1;
     return 0;
}

int FuncionM(){
int x,y,i,k;
double mm;
mm=sqrt(n);
   for(m=ceil(mm);m<n;m++){
          for(x=0;x<m;x++){
         for(y=0;y<m;y++){
               p=Coloca(1,x,y);
            if(p==1) return m;
          }
          }
    }
return -1;
}
```

```
void Algoritmos(int a[N][2]){
int k,i,j,cont=0;
   cabeza=CrearN(n+1);
   for(i=0;i<N;i++)
     for(j=0;j<N;j++){
       Aa[i][j]=0;
       nivel[j]=0;
           visto[j]=0;
           }
   pa=(n+1)*(n+1);
   orden=0;
   nivel[1]=1;
   visita(1);
   for(i=0;i<=n;i++)
      for(j=0;j<=n;j++){
        PosF[i][j]=-1;
            VW[i][j]=-1;
       }
   for(i=0;i<n;i++){
      inserta(&cabeza[a[i][0]],a[i][1]);
      inserta(&cabeza[a[i][1]],a[i][0]);
   }
   FuncionM();
   Ancho(Aa);
   Alto(Aa);
   Latex(Aa);
}

void Latex(int A1[N][N]){
int i,k;

fprintf(Algoritmo1,"\n\\begin{frame} \\frametitle{Algoritmo}");
fprintf(Algoritmo1,"\n\\begin{center}");
fprintf(Algoritmo1,"\n\\begin{tikzpicture}");
fprintf(Algoritmo1,"\n\\draw [help lines] (0,0) grid(%d,%d);",AnchoF,nivelF);
 for(i=0;i<nn+1;i++)
   for(k=0;k<nn+1;k++)
     if(A1[i][k]!=0)
      fprintf(Algoritmo1,"\n\\node[negro] (%d) at (%d,%d) {%d};",A1[i][k],k,i,A1[i][k]);
   for(i=1;i<=n;i++){
      for(k=0,ptr=cabeza[i];ptr;){
        fprintf(Algoritmo1,"\n\\draw[negro] (%d) -- (%d);",i,ptr->dato);
        k++;
        ptr=ptr->siguiente;
      }
   }
fprintf(Algoritmo1,"\n\\end{tikzpicture}");
fprintf(Algoritmo1,"\n\\end{center}");
fprintf(Algoritmo1,"\n\\end{frame}");
}
```

```
int main(){
int sal=0,datoa,datob,auxn,i,j,cont=0,a[N][2],arbol=0,ab,cd;
FILE *entrada=fopen("EntradaA.txt","r");
Algoritmo1=fopen("AlgoritmoBR.tex","w");
fprintf(Algoritmo1,"\n\\documentclass{beamer}");
fprintf(Algoritmo1,"\n\\usepackage{tikz}");
fprintf(Algoritmo1,"\n\\begin{document}");
fprintf(Algoritmo1,"\n\\tikzstyle{vertice} = [circle,inner sep=0pt,minimum size=4mm,thick]");
fprintf(Algoritmo1,"\n\\tikzstyle{negro} = [vertice,draw=black,fill=blue!50]");
sal=fscanf(entrada,"%d",&n);
auxn=n;
nn=ceil(sqrt(n));
for(i=0;i<=n;i++)
      for(j=0;j<=n;j++)
        B[i][j]=Euclides(i,j);

for(i=0;i<n;i++){
   for(j=0;j<n;j++)
       if(B[i][j]==1){
          F[cont][0]=i;
          F[cont][1]=j;
          cont++;
       }
   }
while(auxn==n){
 for(ab=0;ab<n;ab++)
     for(cd=0;cd<2;cd++)
       a[ab][cd]=0;
printf("\nArbol: %d",arbol);
cont=0;
  for(i=1;i<n;i++){
      fscanf(entrada,"%d %d",&datoa,&datob);
      a[cont][0]=datoa+1;
      a[cont][1]=datob+1;
      cont++;
  }
  Algoritmos(a);
  fscanf(entrada,"%d",&auxn);
  arbol++;
}
fprintf(Algoritmo1,"\n\\end{document}");
fclose(Algoritmo1);
fclose(entrada);
return 0;
}

void Ancho(int ad[N][N]){
  int i,j,aux2=0;
  AnchoF=0;
```

```
   for(i=0;i<nn+1;i++){
     if(aux2>AnchoF)
         AnchoF=aux2;
      aux2=0;
       for(j=0;j<nn+1;j++)
          if(ad[i][j]!=0){
              aux2=j;
          }
   }
}

void Alto(int ad[N][N]){
   int i,j,aux2=0;
   nivelF=0;
   for(i=0;i<nn+1;i++)
     for(j=0;j<nn+1;j++){
       if(ad[i][j]!=0)
          aux2=i;
       if(aux2>nivelF)
          nivelF=aux2;
     }
}

void visita(int k){
   Nodo *t;
   visto[k]=++orden;
   for(t=cabeza[k];t!=NULL;t=t->siguiente){
       if(visto[t->dato]==0)
          nivel[t->dato]=nivel[k];
       if(visto[t->dato]==0){
          nivel[t->dato]=nivel[k]+1;
          visita(t->dato);
       }
   }
}

void inserta(Nodo** cabeza, Item entrada){
     Nodo *ultimo;
     ultimo=*cabeza;
     if(ultimo==NULL)
       *cabeza=crearNodo(entrada);
     else {
        for(;ultimo->siguiente;)
           ultimo=ultimo->siguiente;
        ultimo->siguiente=crearNodo(entrada);
     }
}

Nodo* crearNodo(Item x){
    Nodo *a;
```

```
    a=(Nodo*)malloc(sizeof(Nodo));
    a->dato=x;
    a->siguiente=NULL;
    return a;
}
```

13. Código en C para generar el archivo de entrada con los árboles

```
#include <stdio.h>

int main(){
int n,i,a,c=0,d=0;
FILE *entrada=fopen("Ent.txt","r");
FILE *Salida=fopen("EntradaA.txt","w");
if(entrada==NULL || Salida==NULL){
   perror("Error al abrir archivo.");
   return -1;
}
a=fscanf(entrada,"%d\n",&n);
while(a!=-1){
fprintf(Salida,"%d\n",n);
fscanf(entrada,"[");
   for(i=0;i<n;i++){
      a=fscanf(entrada,"(%d, %d), ",&c,&d);
      if(a!=0 && a!=-1)
           fprintf(Salida,"%d %d\n",c,d);
      a=fscanf(entrada,"]\n");
   }
}
fprintf(Salida,"0");
fclose(entrada);
fclose(Salida);
printf("Se generÃ³ el archivo EntradaA.txt..\n");
return 0;
}
```