Universidad Autónoma Metropolitana
Unidad Azcapotzalco
División de Ciencias Básicas e Ingeniería

Licenciatura en Ingeniería en Computación

Proyecto Terminal
Encajes primitivos de gráficas planares exteriores

Jorge Arturo Pérez Arcos
208333378

Trimestre 2014 Invierno

Profesor Titular
Departamento de Sistemas
Dr. Francisco Javier Zaragoza Martínez

Índice

Resumen 3
Introducción 3
Antecedentes 4
Justificación 5
Objetivos 5
Marco teórico 5
Desarrollo del proyecto 7
  Algoritmo 1. Triangulaciones diferentes. 7
  Algoritmo 2. Encajes primitivos - Convexo. 8
  Algoritmo 3. Encajes primitivos – No necesariamente convexo. 9
Resultados 10
Análisis y discusión de resultados 11
Conclusiones 12
Referencias 13
Bibliografía 13
Código fuente 14

Resumen

Se implementó un algoritmo para generar las distintas triangulaciones de un polígono de N vértices ($4 \leq N$), y para las triangulaciones generadas, se implementaron dos algoritmos para encontrar el encaje primitivo de gráficas planares exteriores. Con estos algoritmos se genera un polígono convexo y otro no necesariamente convexo.

Las salidas generadas se envían a archivos en formato .tex, que serán compilados por LaTeX, para crear archivos donde se puedan visualizar gráficamente los encajes primitivos.

Introducción

En un polígono convexo todos los ángulos interiores son menores a 180 grados, lo que equivale a decir que todos los vértices apuntan hacia afuera, en cambio, en un polígono cóncavo al menos uno de sus ángulos interiores mide más de 180 grados, lo que equivale a decir que alguno de sus vértices apunta hacia adentro.

Una gráfica no dirigida puede definirse como una pareja ordenada $G = (V, E)$ donde V es un conjunto finito de vértices y E es un conjunto finito de aristas. Una arista puede definirse como un par no ordenado de vértices $E = \{u, v\}$ donde $u, v \in V$. Se dice que dos vértices u, v son vecinos si el par $\{u, v\} \in E$. Una gráfica no dirigida es planar exterior si esta puede ser dibujada en un plano, de tal forma que las aristas no se crucen y que todos los vértices se encuentren en la frontera de la gráfica dibujada.

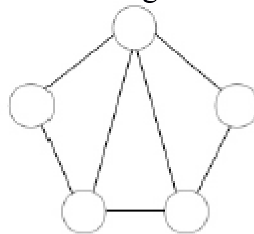

Figura 1. Gráfica planar exterior.

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

Se puede ver a un polígono (convexo o no necesariamente convexo) como una gráfica planar exterior para la cual se va a encontrar la rejilla de tamaño mínimo, de tal forma que se pueda dibujar la gráfica utilizando segmentos de líneas primitivos. Un encaje

primitivo, es el dibujado mediante segmentos de líneas primitivos, en este caso de gráficas planares exteriores, en la rejilla de enteros.

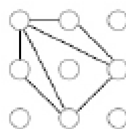

Figura 2. Encaje primitivo de la gráfica mostrada en la figura 1.

Antecedentes

Existen diversos artículos que tratan alguno de los objetivos de este proyecto. El problema de encontrar el encaje primitivo de una gráfica plana, se puede encontrar en el artículo *Every four-colorable graph is isomorphic to a subgraph of the Visibility Graph of the Integer Lattice* [1], por Francisco Javier Zaragoza Martínez[1] y David Flores Peñaloza[2], donde muestran que una gráfica 4-coloreable se puede dibujar en el plano, de manera que sus vértices sean coordenadas enteras y usando segmentos de líneas primitivos para dibujar las aristas.

Otro artículo interesante, elaborado por Marek Chrobak[3] y Shin-ichi Nakano[4], con título *Minimun-width grid drawings of plane graphs* [2], habla sobre cómo dibujar una gráfica plana, de manera que sus vértices tengan coordenadas enteras en el plano y usando segmentos de líneas rectas (no primitivos) para dibujar las aristas, con el objetivo extra de minimizar el tamaño de la rejilla encontrada.

En el artículo *A linear-time algorithm for drawing a planar graph on a grid* [3] de Marek Chrobak[3] y T.H.Payne[5], se muestra un algoritmo de tiempo lineal, que dibuja una gráfica plana de n-vértices en una rejilla de (2n-4)x(n-2), usando segmentos de líneas rectas (no primitivos) para dibujar las aristas.

Dentro de la Universidad Autónoma Metropolitana, se pueden encontrar proyectos terminales sobre combinatoria. Uno de esos proyectos es *Generación de polígonos con triangulaciones ortogonales* [4], donde con una cantidad N de triángulos unitarios ortogonales, se generan polígonos conexos no congruentes.

Otro proyecto terminal interesante, es *Algoritmos para coloración robusta de árboles binarios* [5], en donde se trabaja con el problema de coloración robusta, generando una 3-coloración, en dos gráficas sobre el mismo conjunto de vértices.

[1]Departamento de Sistemas, Universidad Autónoma Metropolitana Plantel Azcapotzalco
[2]Departamento de Matemáticas, Universidad Nacional Autónoma de México
[3]Department of Computer Science and Engineering, University of California, Riverside
[4]Department of Computer Science, Gunma University, Japan
[5]Department of Computer Science and Engineering, University of California, Riverside

Justificación

Este proyecto consiste en la selección e implementación de varios algoritmos para generar polígonos, así como todas las posibles triangulaciones de cada uno de los polígonos generados, obteniendo de esta manera, las gráficas planares exteriores que se van a dibujar en una rejilla con el menor tamaño posible, utilizando segmentos de líneas primitivos. Se busca obtener una clasificación de las imágenes primitivas de gráficas planares exteriores que se generen.

El problema de dibujar gráficas es muy interesante, debido principalmente al reto matemático, algorítmico y de programación que acarrea. La implementación de los algoritmos que se utilizarán durante el desarrollo de este proyecto, requiere de habilidades matemáticas y de programación, propias de un ingeniero en computación.

Anteriormente se han llevado a cabo proyectos donde se busca dibujar gráficas planares en una rejilla, en algunos se logró dibujar gráficas con sus respectivas triangulaciones, buscando llegar a algún tipo especial de dibujo para la gráfica seleccionada, por ejemplo, el dibujo de una gráfica convexa. Lo que hace diferente a este proyecto, es el uso de segmentos de líneas primitivos para el dibujado, así como la optimización del tamaño de la rejilla donde se va a dibujar, de manera que se encuentre la mínima rejilla para dibujar la gráfica, ya sea convexa o no.

En un futuro se podrían mejorar los algoritmos que aquí se utilicen o encontrar algoritmos mejores, para analizar los futuros resultados obtenidos y compararlos con los que se obtengan en este proyecto. También es posible que se implementen algoritmos para trabajar con otros tipos de gráficas (árboles por ejemplo).

Objetivos

1. Seleccionar e implementar un algoritmo para la generación de polígonos con lados *primitivos*.
2. Seleccionar e implementar un algoritmo para la generación de polígonos convexos con lados *primitivos*.
3. Seleccionar e implementar un algoritmo para triangular polígonos.
4. Implementar un algoritmo para la generación de encajes primitivos de gráficas planares exteriores.
5. Presentar los resultados de la ejecución del algoritmo.

Marco teórico

La envolvente convexa (convex hull) de un conjunto de puntos P, es el polígono convexo más pequeño $CH(P)$, para el cual cada punto en P pertenece a la frontera de $CH(P)$ o a su interior. Visto de otra manera, podemos imaginar a los puntos como clavos en una tabla y a la envolvente convexa como una liga tan grande como para envolver a los

clavos de la tabla. Si soltamos la liga, esta envolvería a un área tan pequeña como sea posible, esa área es la envolvente convexa de esos puntos/clavos.

Figura 3. Analogía de la liga para encontrar la envolvente convexa.

Como cada vértice en *CH(P)* es un vértice en el polígono original *P*, el algoritmo para encontrar la envolvente convexa es básicamente un algoritmo que decide qué vértices en *P* deberían ser escogidos como parte de la envolvente convexa. Hay disponibles varios algoritmos para encontrar la envolvente convexa, pero aquí hablaremos un poco sobre el scan de Graham que se ejecuta en $O(n \log n)$.

El scan de Graham, primero ordena los n puntos de P de manera angular, utilizando como pivote al punto más abajo y a la derecha de todos, es decir, con la componente y más pequeña y en caso de empate, se busca la componente x más grande. Este algoritmo mantiene una pila S de candidatos para formar parte de *CH(P)*. Cada punto en P es insertado una vez en S y los puntos que no formarán parte de *CH(P)*, tarde o temprano serán eliminados de S. El scan de Graham mantiene este invariante: los tres primeros puntos en la pila S, siempre deben hacer un giro a la izquierda (esta es básicamente la propiedad de un polígono convexo).

Dada la naturaleza de este algoritmo, el único problema para encontrar nuestro encaje primitivo convexo de una gráfica planar, es que no se estaría optimizando el tamaño de la rejilla donde se pretende encajar la gráfica.

Otros algoritmos que pueden ayudar a resolver el problema, son los que hacen búsqueda con retroceso, pues este método resuelve un problema por buscar en todo el espacio de búsqueda hasta encontrar una solución. El problema de estos algoritmos, es que al tener que buscar en todo el espacio de búsqueda, suelen tomar algo de tiempo para dar una solución (en casos grandes, aunque no en casos pequeños).

La ventaja de usar algoritmos de búsqueda con retroceso, es que al buscar en todo el espacio de búsqueda, se pueden encontrar varias soluciones optimizadas para los encajes primitivos tal que encajen en rejillas más pequeñas, soluciones que con algoritmos de envolvente convexa serían descartadas, entonces se llega al problema de elegir tiempo vs optimización.

Desarrollo del proyecto

Se realizaron tres algoritmos principales para este proyecto; el primero se encarga de generar todas las triangulaciones diferentes de un polígono con N vértices, el segundo encuentra encajes primitivos convexos y el tercero encuentra encajes primitivos no necesariamente convexos. Respecto a los algoritmos dos y tres, se optimizó el tamaño de la rejilla donde se generó el encaje primitivo.

A continuación, se muestra el pseudocódigo de cada algoritmo.

Algoritmo 1. Triangulaciones diferentes. Este algoritmo se realizó usando búsqueda con retroceso y transformación de matrices (las gráficas se representaron mediante matrices de adyacencia), pues es necesario conocer todas las triangulaciones posibles de cada polígono, para después podar y eliminar las triangulaciones que se repiten. Se envía una gráfica *g* a la función que va quitando triángulos del polígono original, uno a uno mediante llamadas recursivas, hasta llegar a un caso base en donde se construye un polígono con los triángulos extraídos anteriormente. Cada que se encuentra una triangulación, se debe comparar con las triangulaciones generadas anteriormente para evitar guardar triangulaciones idénticas.

g.v: cantidad de vértices que pertenecen al polígono actualmente
g.tot: cantidad de vértices de la gráfica
tri: tipo de dato triangulación, contiene tres puntos (a, b, c) y los vecinos (anterior-siguiente) de cada punto
q: pila que contiene las triangulaciones extraídas de *g*
total: cantidad de gráficas en el vector *sol*
sol: vector que contiene las gráficas (no isomorfismos) con sus triangulaciones

```
Triangula(g)
        si g.v == 3 entonces
                pila tmp(q)
                grafica g´ = g
                mientras tmp no esté vacía
                        extraer triangulación de tmp y agregar a g´
                fin mientras

                para i desde 0 hasta total
                        si g´ y sol[i] son isomorfismos entonces
                                regresar
                        si no
                                ordenar los vecinos de cada vértice en g´
                                agregar g´ a sol
                                total = total +1
                        fin si
                fin para
        fin si
```

```
triangulación tri
para i desde 0 hasta g.tot
        si g.vecinos[i][0] ¡= g.vecinos[i][1] entonces
                a = i
                b = g.vecinos[a][1]
                c = g.vecinos[b][1]
                tri.a[0] = a
                tri.a[1] = g.vecinos[a][0]
                tri.a[2] = g.vecinos[a][1]
                tri.b[0] = b
                tri.b[1] = g.vecinos[b][0]
                tri.b[2] = g.vecinos[b][1]
                tri.c[0] = c
                tri.c[1] = g.vecinos[c][0]
                tri.c[2] = g.vecinos[c][1]

                eliminar triangulación tri de g
                Triangula(g)
                Agregar triangulación tri a g
        fin si
fin para
```

Algoritmo 2. Encajes primitivos - Convexo. El algoritmo que se realizó para resolver el problema, es una modificación del scan de Graham y de búsqueda con retroceso, debido a que los encajes primitivos que se encuentren, deben cumplir con la condición de ser convexos y usar un tamaño de rejilla mínimo. Las modificaciones hechas a este algoritmo, evitan la búsqueda de soluciones en casos innecesarios, disminuyendo el tiempo de ejecución y optimizando el tamaño de la rejilla utilizada para encontrar el encaje primitivo convexo. Este algoritmo utiliza un vector de puntos ordenado angularmente, que incluye los puntos de la rejilla donde se buscará el encaje primitivo; al igual que en el scan de Graham, se escoge al pivote (*P[0]*)como primer elemento para formar parte del encaje primitivo convexo, pero al último elemento (*P[N-1]*) se le escoge de manera temporal sólo para poder hacer la verificación de convexidad (3 puntos mostrando un giro contrarreloj), sin embargo, ese punto no forma parte del encaje primitivo desde el principio, sólo el pivote.

faltan: cantidad de vértices por encajar
sobran: cantidad de vértices libres en la rejilla de enteros
vertAct: vértice del polígono que se pretende encajar en la rejilla
vertMatAct: punto en la rejilla que sigue al último candidato para formar parte del encaje
vp: vector con los puntos de la rejilla (global)
tamvp: tamaño del vector vp
fp: vector con los puntos que pertenecen al encaje primitivo
tamfp: tamaño del vector fp

```
Convexo(faltan, sobran, vertAct, vertMatAct, tamvp, fp)
    bool: solve

    si (faltan == 0) entonces
        dibujar gráfica
        regresa 1;
    fin si

    para i desde vertMatAct hasta tamvp && sobran >= faltan
        si ccw(fp[tamfp-2], fp[tamfp-1], vp[i]) entonces
            para cada arista de vertAct
                verificar que sea un segmento primitivo
                verificar que no intersecte con otras aristas
            fin para

            si vp[i] es un punto válido entonces
                agregar vp[i] a puntos de encaje primitivo
                agregar aristas encontradas
                solve = Convexo(faltan-1, sobran-1, vertAct+1, i+1, tamvp,
                fp)
                si solve == 1 entonces regresa 1
                eliminar aristas encontradas
                sacar vp[i] de los puntos del encaje primitivo
            fin si
        fin si

        sobran = sobran -1
    fin para

    regresa 0
```

Algoritmo 3. Encajes primitivos – No necesariamente convexo. El algoritmo que se realizó para resolver el problema hace uso de búsqueda con retroceso, logrando minimizar el tamaño de rejilla usado para el encaje sin preocuparse por la convexidad del encaje encontrado. Se utiliza una matriz booleana donde se marcan los puntos ocupados de los no ocupados.

faltan: cantidad de vértices por encajar
sobran: cantidad de vértices libres en la rejilla de enteros
vertAct: vértice del polígono que se pretende encajar en la rejilla
tamvp: tamaño del vector vp

```
NnConvexo(faltan, sobran, vertAct, tamvp)
    bool: solve
    int: sobran2 = sobran

    si (faltan == 0) entonces
```

```
        dibujar gráfica
        regresa 1;
fin si

para i desde 0 hasta tamvp && sobran2 >= faltan
        si ocupados[vp[i].x][ vp[i].y] == 0 entonces
                sobran2 = sobran2 - 1
                para cada arista de vertAct
                        verificar que sea un segmento primitivo
                        verificar que no intersecte con otras aristas
                fin para

                si vp[i] es un punto válido entonces
                        agregar vp[i] a puntos de encaje primitivo
                        agregar aristas encontradas
                        ocupados[vp[i].x][ vp[i].y] = 1
                        solve = NnConvexo(faltan-1, sobran-1, vertAct+1, tamvp)
                        si solve == 1 entonces regresa 1
                        ocupados[vp[i].x][ vp[i].y] = 0
                        eliminar aristas encontradas
                        sacar vp[i] de los puntos del encaje primitivo
                fin si
        fin si
fin para

regresa 0
```

Resultados

Hay dos tipos de resultados en este proyecto, por una parte, los obtenidos por el algoritmo de triangulación, por otra parte, los obtenidos por los algoritmos que encuentran los encajes primitivos. Para el primer caso, los resultados se encuentran en archivos de texto que contienen todas las triangulaciones diferentes para un valor de N ($4 \leq N$). Para el segundo caso, los resultados se muestran con ayuda de código LaTeX y el paquete TikZ para poder dibujar las gráficas generadas por los algoritmos.

Este es un ejemplo de un archivo de salida del algoritmo de triangulación.

```
//////////5.txt//////////
5
0 4 1 2 3 4
1 2 0 2
2 3 0 1 3
3 3 0 2 4
4 2 0 3
```

El archivo incluye al principio el valor de N para el cuál se buscaron las distintas triangulaciones, en este caso N = 5, después se incluyen las gráficas con sus respectivas triangulaciones en forma de listas de adyacencia. Como se observa en el archivo, sólo existe una triangulación posible para N = 5.

Lo siguiente, es observar los archivos generados al buscar los encajes primitivos (convexo y no necesariamente convexo). El siguiente ejemplo, muestra una de las gráficas generadas por los algoritmos que generan encajes primitivos, en este caso usaremos una de las gráficas donde N=9.

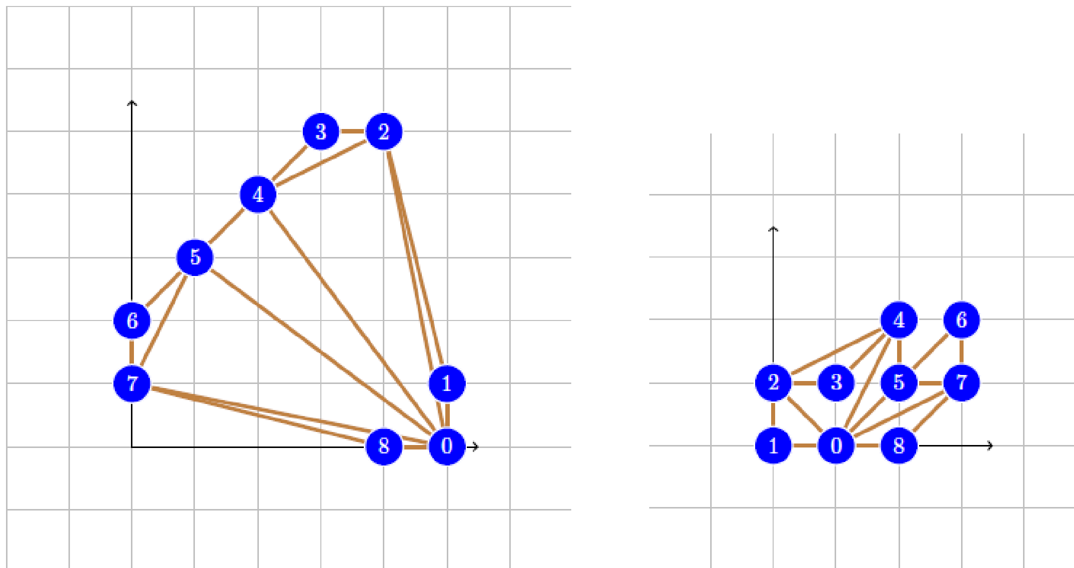

Figura 4. Ambas imágenes pertenecen a la misma gráfica. A la izquierda se muestra la gráfica convexa y a la derecha la gráfica no necesariamente convexa.

Se observa que las gráficas no se parecen mucho, la gráfica convexa fue dibujada en una rejilla de 6x6, mientras que la gráfica no necesariamente convexa fue dibujada en una rejilla de 4x4. También es notoria la optimización en cuanto al espacio requerido para dibujar las gráficas no necesariamente convexas, pues al no exigir convexidad, hay más posibilidades para dibujar una gráfica.

Análisis y discusión de resultados

Hasta el momento, este proyecto ha generado resultados para N = 12, la razón de esto se debe al algoritmo de triangulación, puesto que al ser un algoritmo de búsqueda con retroceso, busca en todo el espacio de direcciones para encontrar todas las posibles triangulaciones para cada polígono, además de que una vez generada una triangulación, esta debe rotar N veces para ser compara con las demás gráficas encontradas con el fin de evitar

polimorfismos. Dado que el algoritmo de triangulación tiene tiempo de complejidad exponencial, crece demasiado el tiempo de ejecución conforme crece N.

En el caso de N = 12, el algoritmo de triangulación tardó aproximadamente 3 días para mostrar resultados (1424 triangulaciones diferentes), por esa razón, decidí parar la ejecución del algoritmo ya que no cuento con el equipo necesario para mantener la ejecución de un proceso por tanto tiempo. Para N=13, el tiempo de ejecución sería aproximadamente de 1 mes.

En el caso de los algoritmos para generar encajes primitivos, el tiempo no es un problema, pero sí puede observarse que crece de acuerdo al tamaño de la entrada.

La siguiente tabla muestra un resumen de los resultados obtenidos.

N	Número de triangulaciones	Rango		Tiempo de ejecución		Tiempo promedio por gráfica	
		Convexo	NN convexo	Convexo	NN convexo	Convexo	NN convexo
4	1	2-2	2-2	0.000s	0.000s	---	---
5	1	3-3	3-3	0.000s	0.000s	---	---
6	4	3-4	3-3	0.000s	0.000s	---	---
7	6	4-5	3-4	0.000s	0.000s	---	---
8	19	4-6	4-4	0.010s	0.030s	0. 52ms	1.57ms
9	49	4-8	4-4	0.100s	0.120s	2.04ms	2.44ms
10	150	4-9	4-5	1.690s	0.650s	11.26ms	4.33 ms
11	442	5-10	4-5	17.40s	7.070s	39.36ms	15.99ms
12	1424	5-14	4-5	3m25.17s	5m7.41s	144.08ms	215.87ms

Tabla 1. Resumen de los resultados.

Se puede observar que a partir de N=11, el tiempo empieza a ser cada vez más grande, esto es debido al enfoque de búsqueda con retroceso que usan los algoritmos con el fin de optimizar el tamaño de la rejilla. Como se mencionó anteriormente, si se hubiera adoptado el algoritmo de envolvente convexa para generar los encajes primitivos convexos, el tiempo de complejidad habría mejorado pero a cambio, se obtendrían tamaños de rejilla más grandes.

Conclusiones

Gracias a los datos obtenidos, puede observarse que el tiempo de ejecución para cada gráfica, aumenta conforme aumenta N debido a que el espacio de búsqueda de los algoritmos de búsqueda con retroceso, aumenta drásticamente. No es igual encontrar una solución para N=8, que tarda aproximadamente 1 ms, que encontrar una solución para N=12, que tarda aproximadamente 180 ms.

Con los resultados también podemos observar que se encuentran tamaños de rejilla de $\lceil\sqrt{N}\rceil+1$, para las gráficas no necesariamente convexas y para algunas gráficas convexas, así como también, se observa que los límites superiores para tamaños de rejilla de las gráficas convexas, están cerca de N.

Mientras se ejecutaban los algoritmos, se pudo observar que en algunas gráficas se encontraba una solución casi instantánea, mientras que en otras gráficas, el tiempo de ejecución era muy grande ya que se encontraba una solución en aproximadamente 20 segundos.

Referencias

[1] Francisco Javier Zaragoza Martínez y David Flores Peñaloza, “Every four-colorable graph is isomorphic to a subgraph of the Visibility Graph of the Integer Lattice”, Proceedings of the 21st Canadian Conference on Computational Geometry (CCCG 09), pp. 91-94, 2009.

[2] Marek Chrobak y Shin-ichi Nakano, “Minimum-width grid drawings of plane graphs”, Computational Geometry vol. 11, pp. 29-54, 1998.

[3] Marek Chrobak y T. Payne, “A linear-time algorithm for drawing a planar graph on a grid”, Information Processing Letters 54, pp. 241-246, 1995.

[4] Rodríguez Villalobos, C. A. “Algoritmos para coloración robusta de árboles binarios”, Proyecto terminal de Ingeniería en Computación, Universidad Autónoma Metropolitana Azcapotzalco, 2008.

[5] Olaguibert Segura, C.A. “Generación de polígonos con triangulaciones ortogonales”, Proyecto terminal de Ingeniería en Computación, Universidad Autónoma Metropolitana Azcapotzalco, 2009.

Bibliografía

O'Rourke Joseph, "Convex Hull in Two Dimensions", *Computational geometry in C*, pp. 63-96.

H.Cormen Thomas, E. Leiserson Charles, L. Rivest Ronald, Stein Clifford, "Computational Geometry, *Introduction to Algorithms*, pp. 933-957.

Halim Steven, Halim Felix, "Complete Search", *Competitive Programming 2*, pp. 39-46.

TikZ and PGF examples, http://www.texample.net/tikz/examples/, consultada el 23 de marzo de 2014.

Código fuente

//////////Algoritmo para generar polígonos//////////

```
#include<iostream>
#include<fstream>
using namespace std;

int main()
{
   ofstream sal;
   char arch_sal[30];
   int n, i, a, b;

   cout << "N = ";
   cin >> n;
   cout << "Archivo de salida: ";
   cin >> arch_sal;
   sal.open(arch_sal);

   for(i = 0; i < n; i++){
      sal << i << " ";
      a = i-1;
      b = i+1;
      if(a == -1) a = n-1;
      if(b == n) b = 0;
      sal << a << " " << b << endl;
   }

   sal.close();

   return 0;
}
```

//////////Algoritmo para triangular polígonos//////////

```
#include<iostream>
#include<fstream>
#include<stack>
#include<vector>
#include<algorithm>
using namespace std;

class Grafica
{
public:
   int v;
   int tot;
   vector< vector < int > > vecinos;
   vector< int >grado;

   Grafica(int v)
   {
      this->v = v;
```

```
        this->tot = v;
        this->vecinos = vector< vector< int > >(v, vector< int >(0));
        this->grado = vector< int >(v, 0);
    }
};

typedef struct t{
    int a[3];
    int b[3];
    int c[3];
}triangulacion;

stack<triangulacion> q;
vector< Grafica  > res;
int total = 0;

void elimina(Grafica  *g, triangulacion tri)
{
    q.push(tri);
    (*g).v = (*g).v - 1;
    (*g).vecinos[tri.a[0]][1]  = tri.b[2];
    (*g).vecinos[tri.c[0]][0]  = tri.b[1];
    (*g).vecinos[tri.b[0]][0]  = (*g).vecinos[tri.b[0]][1]  = -1;
    (*g).grado[tri.b[0]]  -= 2;
}

void agrega(Grafica  *g)
{
    triangulacion tri = q.top();
    q.pop();
    (*g).v = (*g).v + 1;
    (*g).vecinos[tri.a[0]][1]  = tri.a[2];
    (*g).vecinos[tri.c[0]][0]  = tri.c[1];
    (*g).vecinos[tri.b[0]][0]  = tri.b[1];
    (*g).vecinos[tri.b[0]][1]  = tri.b[2];
    (*g).grado[tri.b[0]]  += 2;
}

bool compara(Grafica  a, Grafica  b)
{
    int i, j;

    for(i = 0; i < a.tot; i++){
        for(j = 0; j < a.grado[i]; j++)
            if(a.vecinos[i][j]  != b.vecinos[i][j])
                return false;
    }

    return true;
}

bool isomorfismo(Grafica  a, Grafica  b)
{
    int i, j, k, t;

    //Ordenar vecinos de gráfica
```

```
    for(i = 0; i < a.tot; i++)
            sort(a.vecinos[i].begin(), a.vecinos[i].end());

    for(i = 0; i < a.tot; i++){
        for(j = 0; j < a.tot && a.grado[j] == b.grado[j]; j++){}

        if(j == a.tot)
            if(compara(a, b))
                return true;

        //Girar gráfica
        for(k = 0; k < a.tot; k++)
            for(j = 0; j < a.grado[k]; j++){
                a.vecinos[k][j]++;
                if(a.vecinos[k][j] == a.tot)
                    a.vecinos[k][j] = 0;
            }

        for(k = 0; k < a.v; k++)
            sort(a.vecinos[k].begin(), a.vecinos[k].end());

        vector< int > aux = a.vecinos[a.tot-1];
        t = a.grado[a.tot-1];
        for(k = a.tot-2; k >= 0; k--){
            a.vecinos[k+1] = a.vecinos[k];
            a.grado[k+1] = a.grado[k];
        }

        a.vecinos[0] = aux;
        a.grado[0] = t;
        ///////////////////////
    }

    return false;
}

void triangula(Grafica g)
{
    if(g.v == 3){
        int i, j;
        stack<triangulacion> tmp(q);
        triangulacion x;
        Grafica N(g.tot);

        for(i = 0; i < N.tot; i++){
            for(j = 0; j < g.grado[i]; j++)
                N.vecinos[i].push_back(g.vecinos[i][j]);

            N.grado[i] = g.grado[i];
        }

        while(!tmp.empty()){
            x = tmp.top();
            tmp.pop();
            N.vecinos[x.a[0]].push_back(x.a[2]);
            N.grado[x.a[0]] += 1;
```

```
            N.vecinos[x.b[0]].push_back(x.b[1]);
            N.vecinos[x.b[0]].push_back(x.b[2]);
            N.grado[x.b[0]]  += 2;
            N.vecinos[x.c[0]].push_back(x.c[1]);
            N.grado[x.c[0]]  += 1;
        }

        if(total == 0){
             for(i = 0; i < N.tot; i++)
                        sort(N.vecinos[i].begin(), N.vecinos[i].end());
            res.push_back(N);
            total++;
        }
        else{
            for(i = 0; i < total; i++)
                if( isomorfismo(N,  res[i]) )
                        break;
            if(i  == total){
              for(i = 0; i < N.tot; i++)
                                sort(N.vecinos[i].begin(), N.vecinos[i].end());
                res.push_back(N);
                total++;
            }
        }
    }
    else{
        int a, b, c, i;
        triangulacion tri;
        for(i = 0; i < g.tot; i++){
            if(g.vecinos[i][0]  != g.vecinos[i][1]){
                a = i;
                b = g.vecinos[a][1];
                c = g.vecinos[b][1];

                tri.a[0]  = a;
                tri.a[1]  = g.vecinos[a][0];
                tri.a[2]  = g.vecinos[a][1];
                tri.b[0]  = b;
                tri.b[1]  = g.vecinos[b][0];
                tri.b[2]  = g.vecinos[b][1];
                tri.c[0]  = c;
                tri.c[1]  = g.vecinos[c][0];
                tri.c[2]  = g.vecinos[c][1];

                elimina(&g,   tri);
                triangula(g);
                agrega(&g);
            }
        }
    }
}

int main()
{
    ifstream ent;
    char arch_ent[30], arch_sal[30];
```

```
    triangulacion tri;
    int actu, ante, desp, vert;

    cout << "Archivo de entrada: ";
    cin >> arch_ent;

    ent.open(arch_ent);

    vert = 0;
    while(!ent.eof()  && ent >> actu >> ante >> desp)
        vert++;

    ent.close();
    ent.open(arch_ent);

    Grafica  g(vert);
    while(!ent.eof()  && ent >> actu >> ante >> desp){
        g.vecinos[actu].push_back(ante);
        g.vecinos[actu].push_back(desp);
        g.grado[actu] += 2;
    }

    ent.close();

    tri.a[0]  = 0;
    tri.a[1]  = g.vecinos[0][0];
    tri.a[2]  = g.vecinos[0][1];
    tri.b[0]  = 1;
    tri.b[1]  = g.vecinos[1][0];
    tri.b[2]  = g.vecinos[1][1];
    tri.c[0]  = 2;
    tri.c[1]  = g.vecinos[2][0];
    tri.c[2]  = g.vecinos[2][1];
    elimina(&g,  tri);

    triangula(g);
    ////////////////////////////////////////
    ofstream sal;
    sal.open("triangulaciones.txt");
    sal << vert << endl;
    for(int  t = 0; t < total; t++)
        for(int i = 0; i < res[t].tot; i++){
            sal << i << " " << res[t].grado[i];
            for(int j = 0; j < res[t].grado[i]; j++)
                sal << " " << res[t].vecinos[i][j];

            sal << endl;
        }

     sal.close();
    ////////////////////////////////////////

    return  0;
}
```

//////////Algoritmos para encontrar los encajes primitivos (convexo y no convexo)//////////

```
#include<iostream>
#include<algorithm>
#include<fstream>
#include<vector>
#include<cstdio>
#include<cmath>
#include<cstring>
#include<sys/types.h>
#include<sys/wait.h>
#include<unistd.h>
#define EPS 1e-9
using namespace std;

/*
   Estructuras utilizadas para resolver el problema.
   -Puntos
   -Lineas
   -Segmentos
*/
////////////////////////////////////////////////////
struct point{
   double x, y;
   bool operator < (point other){
      if( fabs(x < - other.x) < EPS )
         return x < other.x;
      return y < other.y;
   }
};

struct line{double a, b, c;};

struct segment{
   point a;
   point b;
   line l;
};
////////////////////////////////////////////////////

/*
   Variables globales:
      -matrizAdy  => Matriz de adyacencia.
      -archConv   => Contador para archivos de gráficas convexas.
      -archNoConv        => Contador para archivos de gráficas no convexas.
      -ocupados   => Matriz para diferenciar los puntos ocupados de los no ocupados.
      -pivot      => Pivot para ordenamiento angular de la rejilla de enteros.
      -vp        => Vector que contiene los puntos de la rejilla de enteros.
      -pos       => Vector que contiene los puntos del encaje de la gráfica.
      -aristas    => Vector que contiene los segmentos de linea recta que pertenecen a la gráfica.
*/
////////////////////////////////////////////////////
int matrizAdy[30][30],  archConv = 1, archNoConv = 1;
bool ocupados[30][30];
point pivot;
```

```
vector<point> vp;
vector<point> pos;
vector&lt;segment&gt; aristas;
////////////////////////////////////////////////////

/*
   Funciones de ayuda para el algoritmo de envolvente convexa.
*/
////////////////////////////////////////////////////
double cross(point p, point q, point r){
   return (r.x-q.x)*(p.y-q.y) - (r.y-q.y)*(p.x-q.x);
}

bool collinear(point p, point q, point r){
   return fabs(cross(p, q, r)) < EPS;
}

double dist(point p1, point p2){
   return hypot(p1.x-p2.x, p1.y-p2.y);
}

bool ccw(point p, point q, point r){
   return cross(p, q, r) >= 0;
}

bool angleCmp(point a, point b){
   if(collinear(pivot, a, b)){
      if(a.y == b.y) return dist(pivot, a) > dist(pivot, b);
      else return dist(pivot, a) < dist(pivot, b);
   }
   double d1x = a.x - pivot.x, d1y = a.y - pivot.y;
   double d2x = b.x - pivot.x, d2y = b.y - pivot.y;
   return (atan2(d1y, d1x) - atan2(d2y, d2x)) < 0;
}
////////////////////////////////////////////////////

/*
   Función que devuelve el tamaño mínimo para la rejilla de enteros;
   ahí se va a buscar el encaje de cada gráfica.
*/
int tamanio(int n){
   int i;
   for(i = 2; n > i*i; i++){}
   return i;
}

/*
   Función que encuentra la recta que pasa por los puntos p1 y p2.
*/
void pointsToLine(point p1, point p2, line *l){
   if(p1.x == p2.x){
      l->a = 1.0;    l->b = 0.0;    l->c = -p1.x;
   }
   else{
      l->a = -(p1.y - p2.y)/(p1.x - p2.x);
      l->b = 1.0;
```

```
        l->c = -(l->a * p1.x)-(l->b * p1.y);
    }
}

/*
    Función que verifica si un segmento de linea recta es primitivo.
*/
bool seg_primitivo(point p1, point p2, line l){
    double minX, maxX, minY, maxY;
    bool solve = true;

    if(fabs(l.b) < EPS || fabs(l.a) < EPS){
        if(fabs(1.0-dist(p1, p2)) < EPS)
            return true;
        else
            return false;

    }

    minX = fmin(p1.x, p2.x);
    maxX = fmax(p1.x, p2.x);
    minY = fmin(p1.y, p2.y);
    maxY = fmax(p1.y, p2.y);

    for(int i = (int)minX+1; i < (int)maxX && solve; i++)
        for(int j = (int)minY+1; j < (int)maxY; j++)
            if(fabs(l.a*(double)i + l.b*(double)j + l.c) < EPS){
                solve = false;
                break;
            }

    return solve;
}

/*
    Funciones para verificar la intersección entre segmentos de linea recta.
*/
////////////////////////////////////////////////////////////////
bool pointInSegment(point r, point a, point b){
    return ( (r.x < fmax(a.x, b.x)) && (r.x > fmin(a.x, b.x)) && (r.y < fmax(a.y, b.y)) && (r.y > fmin(a.y, b.y))
);
}

bool areParallel(line l1, line l2){
    return ( fabs(l1.a-l2.a) < EPS ) && ( fabs(l1.b-l2.b) < EPS );
}

bool areSame(line l1, line l2){
    return areParallel(l1, l2) && ( fabs(l1.c-l2.c) < EPS );
}

bool intersect(line l, point a, point b){
    bool solve = true;
    double x, y;
    point tmp;
    int segmentsTotal= (int)aristas.size();
```

```
    for(int i = 0; i < segmentsTotal && solve; i++){
        if( areParallel(l, aristas[i].l) )
            continue;

        if( areSame(l, aristas[i].l) )
            if(pointInSegment(a, aristas[i].a, aristas[i].b) || pointInSegment(b, aristas[i].a, aristas[i].b) ||
pointInSegment(aristas[i].a, a, b) || pointInSegment(aristas[i].b, a, b)){
                solve = false;
                continue;
            }

        x = (aristas[i].l.b*l.c - l.b*aristas[i].l.c)/(aristas[i].l.a*l.b - l.a*aristas[i].l.b);
        if(fabs(l.b) > EPS)
            y = -(l.a*x + l.c)/l.b;
        else
            y = -(aristas[i].l.a*x + aristas[i].l.c)/aristas[i].l.b;

        tmp.x = x; tmp.y = y;

        //Casos para lineas verticales.
        if( fabs(aristas[i].l.b) < EPS && tmp.y < fmax(aristas[i].a.y, aristas[i].b.y) && tmp.y >
fmin(aristas[i].a.y, aristas[i].b.y) )
            return false;

        if( fabs(l.b) < EPS && tmp.y < fmax(a.y, b.y) && tmp.y > fmin(a.y, b.y) )
            return false;


        //Casos para lineas horizontales.
        if( fabs(aristas[i].l.a) < EPS && tmp.x < fmax(aristas[i].a.x, aristas[i].b.x) && tmp.x >
fmin(aristas[i].a.x, aristas[i].b.x) )
            return false;

        if( fabs(l.a) < EPS && tmp.x < fmax(a.x, b.x) && tmp.x > fmin(a.x, b.x) )
            return false;

        //Cualquier otro caso.
        if( pointInSegment(tmp, aristas[i].a, aristas[i].b) && pointInSegment(tmp, a, b) )
            return false;
    }

    return true;
}
///////////////////////////////////////////////////

/*
        Función encargada de la compilación de archivos .TEX, así como de la eliminación
        de los archivos extra generados por la compilación.
*/
void ejecutar(char* com){
        execl("/bin/sh", "/bin/sh", "-c", com, (char*) 0);
}

/*
        Función que genera un archivo .TEX con el dibujo del encaje de la gráfica.
```

```
*/
void dibujar(bool clave, int tamvp){
        ofstream sal;
        int limX,  limY,  tamtmp, estado;
        char t = 'a', arch_sal[50], num[20],  com1[50],  com2[50];
        char* tmpPTR;
        pid_t pid;
        if(clave)  sprintf(num, "%d", archNoConv++);
        else sprintf(num, "%d", archConv++);
        strcat(num, ".tex");
        strcpy(arch_sal, num);

        /*
          Cabeceras para el archivo de salida.
                Rejilla  y ejes X/Y como apoyo para la gráfica.
        */
        sal.open(arch_sal);
        sal << "\\documentclass{standalone}\n\\usepackage{tikz}\n\\begin{document}\n";
        sal << "   \\begin{tikzpicture}[every node/.style={draw,circle}]\n";
        sal << "      \\draw [help lines, lightgray, ultra thin] (-2,-2)  grid (";
        limX  = limY  = pos[0].x;
        if(clave){
                tamtmp  = (int)sqrt(tamvp);
                limX  = limY  = tamtmp-1;
        }
        sal << limX+2  << "," << limY+2   << ");\n";
        sal << "      \\draw [->, thick]  (0,0)  -- (" << limX  << ".5,0);\n";
        sal << "      \\draw [->, thick]  (0,0)  -- (0," << limY  << ".5);\n";

        /*
                Código para las aristas.
        */
        tamtmp  = (int)aristas.size();
        for(int i = 0;  i < tamtmp;  i++){
                limX  = aristas[i].a.x;  limY  = aristas[i].a.y;
                sal << "      \\draw [ultra thick, brown] (" << limX  << "," << limY  << ") -- (";
                limX  = aristas[i].b.x;  limY  = aristas[i].b.y;
                sal << limX  << "," << limY  << ");\n";
        }

        /*
                Código para los vértices
        */
        tamtmp  = (int)pos.size();
        for(int i = 0;  i < tamtmp;  i++){
                limX  = pos[i].x;  limY  = pos[i].y;
                sal << "      \\node [white, fill=blue]  (" << t++ << ") at (";
                sal << limX  << "," << limY  << ") { " << i << " };\n";
        }

        sal << "   \\end{tikzpicture}\n";
        sal << "   \\end{document}";
        sal.close();

        /*
                Se crea el archivo PDF de salida y se eliminan  los archivos restantes.
```

```
        */
        strcpy(com1, "pdflatex ");
        strcat(com1, arch_sal);
        if(pid = fork() > 0) wait(&estado);
        if(pid == 0) ejecutar(com1);                    //Se crea archivo PDF con la gráfica

        strcpy(com2, "rm ");
        strcat(com2, arch_sal);
        if(pid = fork() > 0) wait(&estado);
        if(pid == 0) ejecutar(com2);                    //Se elimina archivo .tex

        tmpPTR = strstr(com2, ".tex");
        strncpy(tmpPTR, ".log", 4);
        if(pid = fork() > 0) wait(&estado);
        if(pid == 0) ejecutar(com2);                    //Se elimina archivo .log

        tmpPTR = strstr(com2, ".log");
        strncpy(tmpPTR, ".aux", 4);
        if(pid = fork() > 0) wait(&estado);
        if(pid == 0) ejecutar(com2);                    //Se elimina archivo .aux

        strcpy(com2, "mv "); com2[3] = '\0';
        tmpPTR = strstr(arch_sal, ".tex");
        strncpy(tmpPTR, ".pdf", 4);
        strcat(com2, arch_sal);
        if(clave) strcat(com2, " /home/lug/PT/NoConvexo");
        else strcat(com2, " /home/lug/PT/Convexo");
        if(pid = fork() > 0) wait(&estado);
        if(pid == 0) ejecutar(com2);                    //Se mueve *.pdf a un directorio específico
}

/*
   Modificación del algoritmo envolvente convexa.
   CH2 es el algoritmo que busca el encaje primitivo.
   CH es quien prepara todo y llama a CH2.
*/
bool CH2(int faltan, int sobran, int vertAct, int vertMatActual, vector<point> fp, int tamfp, int tamvp){
   line l;
   segment tmp;
   bool solve, flag;
   int contAristas;

   /*
      Se encontró el encaje.
   */
   if(faltan == 0){
      dibujar(false, 0);
      return true;
   }

   for(int i = vertMatActual; i < tamvp && sobran >= faltan; i++, sobran--){
      if( ccw(fp[tamfp-2], fp[tamfp-1], vp[i]) ){
         contAristas = 0;
         flag = true;
         for(int j = 0; j < vertAct && flag; j++)
            if(matrizAdy[vertAct][j] == 1){
```

```
                pointsToLine(pos[j], vp[i], &l);
                if( seg_primitivo(pos[j], vp[i], l) && intersect(l, pos[j], vp[i]) ){
                    tmp.a = pos[j]; tmp.b = vp[i]; tmp.l = l;
                    aristas.push_back(tmp);
                    contAristas++;
                }
                else{
                    for(int k = 0; k < contAristas; k++)
                        aristas.pop_back();

                    flag = false;
                }
            }

        if(flag){
            pos.push_back(vp[i]);
            fp.push_back(vp[i]);
            solve = CH2(faltan-1,  sobran-1, vertAct+1, i+1, fp, tamfp+1, tamvp);
            if(solve) return true;
            pos.pop_back();
            fp.pop_back();
            for(int j = 0; j < contAristas; j++)
                aristas.pop_back();
        }
      }
    }

    return false;
}

bool CH(int tamvP, int faltan){
    /*
        P0  => Posición del pivot en P.
        N   => Tamaño del vector P.
    */
    int P0 = 0, N = tamvP*tamvP;

    /*
        Se ordena P, tomando como pivot al punto más cercano a la esquina
        inferior derecha de la rejilla de puntos.
    */
    for(int i = 1; i < N; i++)
        if(vp[i].y  < vp[P0].y || vp[i].y == vp[P0].y  && vp[i].x > vp[P0].x)
            P0 = i;

    point temp = vp[0]; vp[0] = vp[P0]; vp[P0] = temp;
    pivot = vp[0];
    sort(++vp.begin(), vp.end(), angleCmp);

    /*
        Se agrega el elemento N-1 temporalmente para verificar el giro a la
        izquierda, al final se eliminará (a menos que sea elegido como
        último elemento).
    */
    vector<point> finalP;
    finalP.push_back(vp[N-1]);  finalP.push_back(vp[0]);
```

```
    pos.push_back(vp[0]);

    return CH2(faltan-1,  N-1,  1, 1, finalP,  2, N);
}

/*
            Función para encontrar el encaje de la gráfica  no convexa.
*/
bool NC(int  faltan, int sobran, int tamvp, int vertAct){
            int sobran2 = sobran, contAristas;
            bool flag, solve;
            line l;
            segment tmp;

            if(faltan  == 0){
                        dibujar(true,  tamvp);
                        return true;
            }

            for(int i = 0; i < tamvp && sobran2 >= faltan; i++)
                        if(ocupados[ (int)vp[i].x  ][ (int)vp[i].y  ] == false){
                                    sobran2--;
                                    contAristas = 0;
          flag = true;
                                    for(int j = 0; j < vertAct && flag; j++)
             if(matrizAdy[vertAct][j]   == 1){
                 pointsToLine(pos[j], vp[i], &l);
                 if(  seg_primitivo(pos[j],  vp[i],  l) && intersect(l, pos[j], vp[i])  ){
                    tmp.a = pos[j]; tmp.b = vp[i]; tmp.l = l;
                    aristas.push_back(tmp);
                    contAristas++;
                 }
                 else{
                    for(int k = 0; k < contAristas; k++)
                       aristas.pop_back();

                    flag = false;
                 }
             }

            if(flag){
             pos.push_back(vp[i]);
             ocupados[ (int)vp[i].x  ][ (int)vp[i].y  ] = 1;
             solve = NC(faltan-1,  sobran-1, tamvp, vertAct+1);
             if(solve)  return true;
             ocupados[ (int)vp[i].x  ][ (int)vp[i].y  ] = 0;
             pos.pop_back();
             for(int  j = 0; j < contAristas; j++)
                 aristas.pop_back();
          }
                        }

            return false;
}

int main(){
```

```
/*
    N          => Cantidad de vértices de los polígonos a leer.
    tamMin      => Tamaño mínimo que debe tener la matriz para encontrar una solución.
    arch_ent    => Nombre del archivo de entrada.
    actVert => Vértice actual.
    grado   => Grado del vértice actual.
    arista  => Conexión entre el vértice actual y otro vértice de la gráfica.
    tamMin2 => Variable de respaldo para tamMin.
    posVP   => Posición en el vector de puntos.
    solve   => Flag que indica si se encontró el encaje primitivo de la gráfica.
*/
ifstream ent;
int N, tamMin, actVert, grado, arista, tamMin2, posVP;
char arch_ent[30];
bool solve;

/*
    El usuario brindará el nombre del archivo con el cual se va a trabajar.
    Se lee la cantidad de vértices que tendrán las gráficas del archivo
    y encontramos el tamaño mínimo que debe tener la matriz donde buscaremos
    el encaje primitivo de la gráfica.
*/
cout << "Archivo de entrada: ";
cin >> arch_ent;
ent.open(arch_ent);
ent >> N;
tamMin = tamanio(N);

while( !ent.eof() && ent >> actVert >> grado ){
    /*
        Se transforman las gráficas de lista a matriz de adyacencia y se busca
        el encaje primitivo de cada gráfica.
    */
    //solve = false;
    memset(matrizAdy, 0, sizeof matrizAdy);
    for(int j = 0; j < grado; j++){
                                ent >> arista;
                                matrizAdy[actVert][arista] = 1;
                    }
    for(int i = 1; i < N; i++){
        ent >> actVert >> grado;
        for(int j = 0; j < grado; j++){
            ent >> arista;
            matrizAdy[actVert][arista] = 1;
        }
    }

    tamMin2 = tamMin;

    while(!solve){
        /*
            Se genera un vector que contiene a los puntos de la rejilla de
            enteros, en donde se buscará el encaje de la gráfica.
        */
        point tmp;
        for(int i = 0; i < tamMin2; i++)
```

```
            for(int j = 0; j < tamMin2; j++){
                tmp.x = (double)j; tmp.y = (double)i;
                vp.push_back(tmp);
            }
        solve = CH(tamMin2, N);
        vp.clear();
        aristas.clear();
        pos.clear();
        tamMin2++;
    }

    solve = false;
    tamMin2 = tamMin;
    memset(ocupados, 0, sizeof ocupados);

    while(!solve){
        point tmp;
        for(int i = 0; i < tamMin2; i++)
          for(int j = 0; j < tamMin2; j++){
              tmp.x = (double)j; tmp.y = (double)i;
              vp.push_back(tmp);
          }

        for(int i = 0; i < tamMin2 && solve == false; i++){
                    tmp.x = (double) i; tmp.y = 0;
                pos.push_back(tmp);
                ocupados[i][0] = 1;
                solve = NC(N-1, (tamMin2*tamMin2)-1, tamMin2*tamMin2, 1);
                ocupados[i][0] = 0;
                pos.pop_back();
        }

        vp.clear();
      aristas.clear();
      pos.clear();
      tamMin2++;
    }
  }

  ent.close();

  return 0;
}
```