

PROYECTO

Seminario de solución de problemas de algoritmia

Sección D11

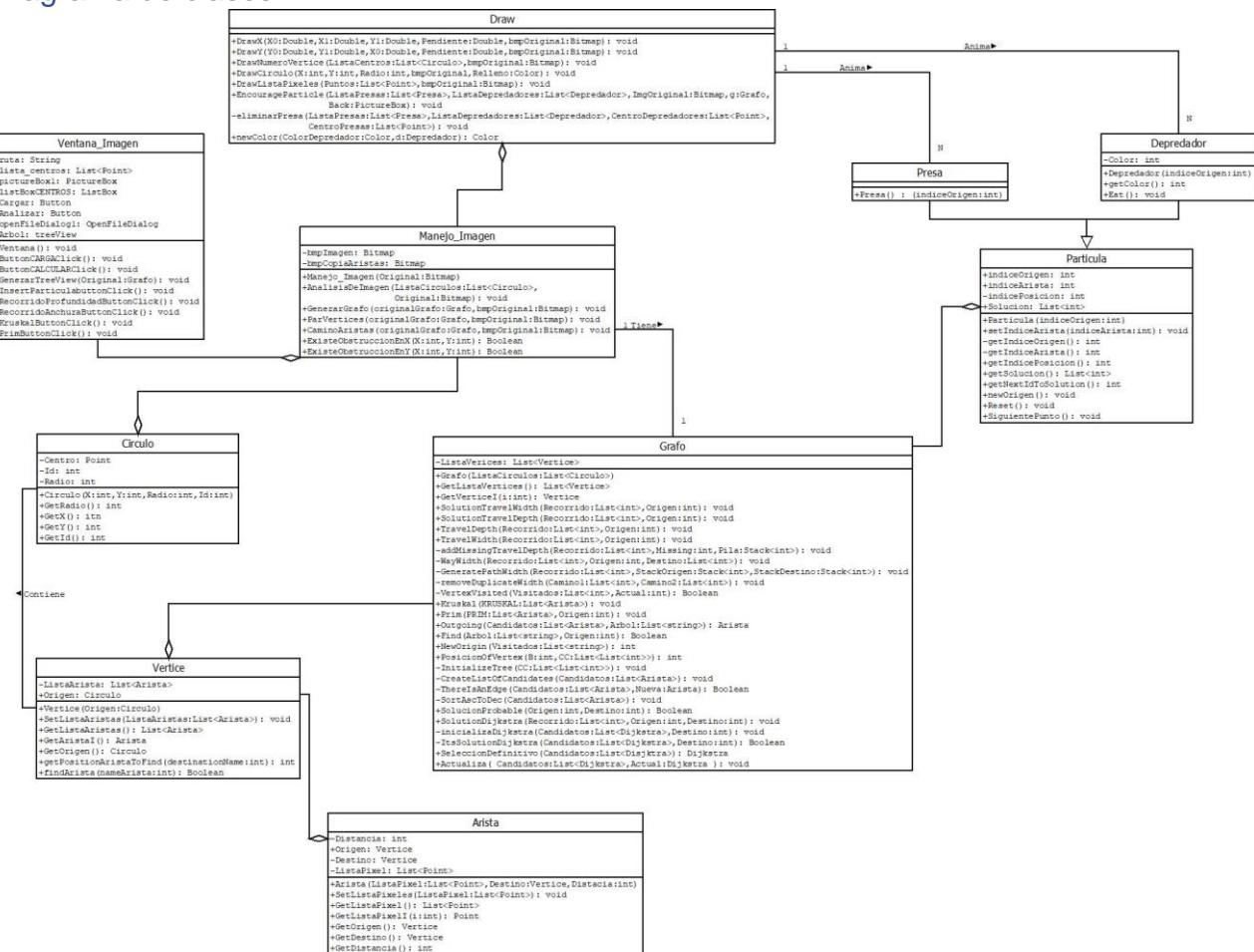
Amaro lechuga Jashua Ricardo
212387784

Tabla de contenido

Diagrama de clases:	3
Objetivo:	4
Marco Teórico:	5
Círculo:	5
Circunferencia:	5
Ecuación de la recta:	5
Grafo:	6
Algoritmo de Fuerza Bruta:	6
Plano cartesiano:	7
RGBA:	7
Pila:	7
Cola:	7
Recorridos:	8
Profundidad: (Ilustración 4)	8
Amplitud: (Ilustración 5)	8
ARM:	8
Kruskal:	9
Prim:	9
Dijkstra:	9
Desarrollo:	10
Etapa 1:	10
Desing:	10
Source:	10
Etapa 2:	11
I. Construcción del grafo:	11
II. Solución par de puntos más cercanos (Ilustración 3):	12
III. Solución camino de 4 vértices (Ilustración 4):	12
Etapa 3:	13
I. Recorrido en Profundidad (Ilustración 3):	13
II. Recorrido en Amplitud (Ilustración 4):	14
Etapa 4:	14
I. Kruskal (Ilustración 1):	14

II. PRIM (Ilustración 5):	15
Etapa 5:.....	15
I. Dijkstra (Ilustración 1):	15
Pruebas y Resultados:.....	16
Etapa 1:.....	16
Etapa 2:.....	17
Etapa 3:.....	19
Etapa 4:.....	20
Etapa 5:.....	21
Conclusiones:.....	21
Etapa 1:.....	21
Etapa 2:.....	22
Etapa 3:.....	22
Etapa 4:.....	23
Etapa 5:.....	23
Apéndice(s):.....	23
Etapa 1:.....	25
Etapa 2:.....	36
Etapa 3:.....	51
Etapa 4:.....	61
Etapa 5:.....	68

Diagrama de clases:



Objetivo:

Diseñe un sistema computacional que encuentre todos los círculos contenidos en una imagen, la cual será subida por el usuario. De cada circulo encontrado, Se deberá encontrar su centroide.

Los centroides encontrados serán mostrados al usuario en una ventana. En esa misma se deberá mostrar al usuario la imagen que el subió y sobre la cual se trabajó.

El sistema analice una imagen que representa un entorno. Genere un grafo a partir de la imagen analizada, cada vértice es un circulo de la imagen y cada vértice contiene adyacencias (Aristas) que unen a un circulo con todos los demás siempre y cuando se pueda trazar una línea recta desde un vértice (origen) a otro (destino). Cualquier figura en la imagen puede obstruir la conexión de un vértice a otro, incluso los mismos vértices.

El sistema tendrá la posibilidad de agregar dos tipos de entidades al grafo, una presa y depredadores, el usuario elige el vértice inicial de la presa, y el vértice a donde intentará llegar la presa. Los depredadores se colocan de forma aleatoria en el grafo y el algoritmo de desplazamiento de los mismos lo decide el desarrollador. Las partículas se pueden desplazar de un vértice a otro a través de sus aristas, es decir, una partícula puede ir del vértice a al vértice b si existe una arista que conecta a los dos vértices. El desplazamiento es visualmente progresivo.

Si la partícula presa choca con una partícula depredadora, la partícula presa desaparece y la depredadora visualmente muestra que ha alcanzado a la presa (por ejemplo, con cambio de colores, o tamaño). La simulación termina cuando la presa llega a su objetivo o es alcanzada por un depredador.

Asumiendo que la entidad presa conoce por completo el entorno, una vez que la partícula se encuentra en algún vértice, el usuario puede seleccionar otro vértice del grafo, y la partícula debe dirigirse a ese vértice asegurando que la ruta que tome es la más corta.

Marco Teórico:

Círculo:

Es la región del plano delimitada por una circunferencia y que posee un área definida (*ilustración 1*).

- El centro es el punto equidistante a todos los puntos de una circunferencia. Señalado con el nombre C en la figura.
- Un radio es cualquier segmento que une el centro de la circunferencia con un punto cualquiera de la misma. El radio también es la longitud de los segmentos del mismo nombre. Señalado con el nombre R en la figura.
- Un diámetro es cualquier segmento que une dos puntos de la circunferencia pasando por su centro. El diámetro también es la longitud del segmento del mismo nombre. Señalado con el nombre D en la figura.

Circunferencia:

La circunferencia es una curva plana y cerrada tal que todos sus puntos están a igual distancia del centro

Ecuación de la recta:

Un tipo de ecuación lineal es la forma punto-pendiente, la cual nos proporciona la pendiente de una recta y las coordenadas de un punto en ella. La forma punto-pendiente de una ecuación lineal se escribe como $(y - y_1) = m(x - x_1)$. En esta ecuación, m es la pendiente y (x_1, y_1) son las coordenadas del punto.

Veamos de dónde es que viene esta fórmula de punto-pendiente. Aquí está la gráfica de una recta genérica con dos puntos trazados en ella (*Ilustración 1*).

La pendiente de la recta "aumenta conforme va". Ése es el cambio vertical entre dos puntos (la diferencia entre las coordenadas en y) dividida entre el cambio horizontal sobre el mismo segmento (la diferencia entre las corneadas en x). Esto

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$

puede escribirse como $m = \frac{y - y_1}{x - x_1}$. Esta ecuación es la fórmula de la pendiente.

Ahora digamos que uno de esos puntos es un punto genérico (x, y) , lo cual significa que puede ser cualquier punto en la recta, y el otro punto es un punto específico, (x_1, y_1) . Si sustituimos estas coordenadas en la fórmula,

$$m = \frac{y - y_1}{x - x_1}$$

obtenemos $m = \frac{y - y_1}{x - x_1}$. Ahora podemos manipular un poco la ecuación al multiplicar ambos lados de la fórmula por $(x - x_1)$. Que se simplifica a $(y - y_1) = m(x - x_1)$ (*Ilustración 2*).

$(y - y_1) = m(x - x_1)$ es el punto-pendiente de la fórmula. Hemos convertido la

fórmula de la pendiente en la fórmula punto-pendiente. No lo hicimos sólo por diversión, sino porque la fórmula punto-pendiente es a veces más útil que la fórmula de la pendiente, por ejemplo, cuando necesitamos encontrar la ecuación de una recta dados un punto y la pendiente.

Grafo:

Un grafo es un conjunto de objetos llamados vértices o nodos unidos por enlaces llamados aristas o arcos, que permiten representar relaciones binarias entre elementos de un conjunto. Son objeto de estudio de la teoría de grafos.

Típicamente, un grafo se representa gráficamente como un conjunto de puntos (vértices o nodos) unidos por líneas (aristas).

Propiedades:

- Adyacencia: dos aristas son adyacentes si tienen un vértice en común, y dos vértices son adyacentes si una arista los une.
- Incidencia: una arista es incidente a un vértice si ésta lo une a otro.
- Ponderación: corresponde a una función que a cada arista le asocia un valor (costo, peso, longitud, etc.), para aumentar la expresividad del modelo. Esto se usa mucho para problemas de optimización, como el del vendedor viajero o del camino más corto.
- Etiquetado: distinción que se hace a los vértices y/o aristas mediante una marca que los hace únicamente distinguibles del resto.

Algoritmo de Fuerza Bruta:

La búsqueda por fuerza bruta, búsqueda combinatoria, búsqueda exhaustiva o simplemente fuerza bruta, es una técnica trivial, pero a menudo usada, que consiste en enumerar sistemáticamente todos los posibles candidatos para la solución de un problema, con el fin de chequear si dicho candidato satisface la solución al mismo.

Por ejemplo, un algoritmo de fuerza bruta para encontrar el divisor de un número natural n consistiría en enumerar todos los enteros desde 1 hasta n , chequeando si cada uno de ellos divide n sin generar resto. Otro ejemplo de búsqueda por fuerza bruta, en este caso para solucionar el problema de las ocho reinas (posicionar ocho reinas en el tablero de ajedrez de forma que ninguna de ellas ataque al resto), consistiría en examinar todas las combinaciones de posición para las 8 reinas (¡en total $64! / 56! = 178.462.987.637.760$ posiciones diferentes), comprobando en cada una de ellas si las reinas se atacan mutuamente.

La búsqueda por fuerza bruta es sencilla de implementar y, siempre que exista, encuentra una solución. Sin embargo, su coste de ejecución es proporcional al número de soluciones candidatas, el cual es exponencialmente proporcional al tamaño del problema. Por el contrario, la búsqueda por fuerza bruta se usa

habitualmente cuando el número de soluciones candidatas no es elevado, o bien cuando éste puede reducirse previamente usando algún otro método heurístico.

Plano cartesiano:

Es un sistema bidimensional de intersección de rectas, por definición, considera como el punto cero de las rectas y se conoce como origen de coordenadas. Al eje horizontal o de las abscisas se le asigna los números reales de las equis ("x"); y al eje vertical o de las ordenadas se le asignan los números reales de las yes ("y").

Al cortarse las dos rectas, dividen al plano en cuatro regiones o zonas, que se conocen con el nombre de cuadrantes (*ilustración 1*);

- Primer cuadrante "I": Región superior derecha
- Segundo cuadrante "II": Región superior izquierda
- Tercer cuadrante "III": Región inferior izquierda
- Cuarto cuadrante "IV": Región inferior derecha

RGBA:

- R: Color rojo
- G: Color verde
- B: Color azul
- A: Indica transparencia 255 representa un color sólido y debajo de ese numero indica transparencia de color, siendo 0 la inexistencia del color dado a que es completamente transparente

Pila:

Las pilas son estructuras de datos que tienen dos operaciones básicas: push (para insertar un elemento) y pop (para extraer un elemento). Su característica fundamental es que al extraer se obtiene siempre el último elemento que acaba de insertarse. Por esta razón también se conocen como estructuras de datos LIFO (del inglés Last In First Out). Una posible implementación mediante listas enlazadas sería insertando y extrayendo siempre por el principio de la lista. Gracias a las pilas es posible el uso de la recursividad (lo veremos en detalle en el tema siguiente). La variable que llama al mismo procedimiento en el q está, habrá que guardarla, así como el resto de variables de la nueva llamada, para a la vuelta de la recursividad ir sacándolas, esto es posible a la implementación de pilas.

Cola:

Las colas también son llamadas FIFO (First In First Out), que quiere decir “el primero que entra es el primero que sale”.

Se inserta por un sitio y se saca por otro, en el caso de la cola simple se inserta por el final y se saca por el principio. Para gestionar este tipo de cola hay que recordar siempre cual es el siguiente elemento que se va a leer y cuál es el último elemento que se ha introducido.

Recorridos:

Profundidad: (Ilustración 4)

Es equivalente a recorrer un árbol por niveles. Dado un nodo v , se visitan primero todos los nodos adyacentes a v , luego todos los que están a distancia 2 (y no visitados), a distancia 3, y así sucesivamente hasta recorrer todos los nodos.

Búsqueda primero en profundidad

- Es necesario llevar la cuenta de los nodos visitados (y no visitados).
- El recorrido no es único: depende del vértice inicial y del orden de visita de los vértices adyacentes.
- El orden de visita de unos nodos puede interpretarse como un árbol: árbol de expansión en profundidad asociado al grafo.
- El recorrido en profundidad de un grafo conexo, G ,
- Las aristas de T aristas de G empleadas en el recorrido en profundidad.
- La raíz de T punto de partida de la exploración de G . Si el grafo no es conexo, tendremos un árbol de recubrimiento por cada componente conexa.
- La exploración en profundidad de un grafo nos permite numerar los nodos del grafo en pre-orden.

Amplitud: (Ilustración 5)

Es equivalente a recorrer un árbol por niveles, dado un nodo v , se visitan primero todos los nodos adyacentes a v , luego todos los que están a distancia 2 (y no visitados), a distancia 3, y así sucesivamente hasta recorrer todos los nodos. Pila:

Las pilas son estructuras de datos que tienen dos operaciones básicas: push (para insertar un elemento) y pop (para extraer un elemento). Su característica fundamental es que al extraer se obtiene siempre el último elemento que acaba de insertarse. Por esta razón también se conocen como estructuras de datos LIFO (del inglés Last In First Out). Una posible implementación mediante listas enlazadas sería insertando y extrayendo siempre por el principio de la lista. Gracias a las pilas es posible el uso de la recursividad (lo veremos en detalle en el tema siguiente). La variable que llama al mismo procedimiento en el q está, habrá que guardarla, así como el resto de variables de la nueva llamada, para a la vuelta de la recursividad ir sacándolas, esto es posible a la implementación de pilas.

ARM:

Dado un grafo conexo y no dirigido, un árbol recubridor mínimo de ese grafo es un subgrafo que tiene que ser un árbol y contener todos los vértices del grafo inicial. Cada arista tiene asignado un peso proporcional entre ellos, que es un número representativo de algún objeto, distancia, etc.; y se usa para asignar un peso total al árbol recubridor mínimo computando la suma de todos los pesos de las aristas

del árbol en cuestión. Un árbol recubridor mínimo o un árbol expandido mínimo es un árbol recubridor que pesa menos o igual que otros árboles recubridores. Todo grafo tiene un bosque recubridor mínimo.

Kruskal:

El algoritmo de Kruskal es un algoritmo de la teoría de grafos para encontrar un árbol recubridor mínimo en un grafo conexo y ponderado. Es decir, busca un subconjunto de aristas que, formando un árbol, incluyen todos los vértices y donde el valor de la suma de todas las aristas del árbol es el mínimo. Si el grafo no es conexo, entonces busca un bosque expandido mínimo (un árbol expandido mínimo para cada componente conexa).

Prim:

El algoritmo de Prim es un algoritmo perteneciente a la teoría de los grafos para encontrar un árbol recubridor mínimo en un grafo conexo, no dirigido y cuyas aristas están etiquetadas.

En otras palabras, el algoritmo encuentra un subconjunto de aristas que forman un árbol con todos los vértices, donde el peso total de todas las aristas en el árbol es el mínimo posible. Si el grafo no es conexo, entonces el algoritmo encontrará el árbol recubridor mínimo para uno de los componentes conexos que forman dicho grafo no conexo.

La principal diferencia entre Prim y Kruskal es que Prim tiene un origen y a partir de ese origen se forma el ARM y Prim solo agrega aristas al árbol si estas aristas están conectadas al componente conexo.

Dijkstra:

El algoritmo de Dijkstra, también llamado algoritmo de caminos mínimos, es un algoritmo para la determinación del camino más corto, dado un vértice origen, hacia el resto de los vértices en un grafo que tiene pesos en cada arista.

La idea subyacente en este algoritmo consiste en ir explorando todos los caminos más cortos que parten del vértice origen y que llevan a todos los demás vértices; cuando se obtiene el camino más corto desde el vértice origen hasta el resto de los vértices que componen el grafo, el algoritmo se detiene. Se trata de una especialización de la búsqueda de costo uniforme y, como tal, no funciona en grafos con aristas de coste negativo (al elegir siempre el nodo con distancia menor, pueden quedar excluidos de la búsqueda nodos que en próximas iteraciones bajarían el costo general del camino al pasar por una arista con costo negativo).

Desarrollo:

Etapa 1:

Desing:

La primera parte del trabajo que se realizó fue concepción de la ventana que se mostraría al usuario teniendo en cuenta el objetivo de la práctica, los elementos que se agregaron a la ventada se encuentran en la parte superior izquierda en este caso, están en Tool y en Windows Forms (*ilustración 2*).

Se agregaron dos botones (Cargar Imagen y Analizar) para que cuando fueran presionados uno de ellos (Cargar imagen) te permitiera seleccionar la imagen con círculos con la que deseamos trabajar y el otro botón (Analizar) mostrara el proceso de búsqueda de centroides y sus ubicaciones (*ilustración 3*).

Después se agregó una pictureBox la cual permitiría que tanto como la imagen seleccionada, así como el proceso de análisis sean mostrados al usuario, la pictureBox se agrega debajo de los dos botones y aparece como un cuadro vacío (*ilustración 3*).

También se agregó una listBox que es la que mostraría la lista de puntos encontrados al analizar la imagen seleccionada (*ilustración 3*).

Y por último en la ventana de diseño se agrega un openFileDialog que es el que nos permitiría mediante el botón de “Cargar Imagen” seleccionar una imagen de nuestra computadora (*ilustración 3*).

Source:

Primero se empezó con la codificación de botón de cargar, primero se manda a llamar una ventada para poder seleccionar un archivo, después la ubicación del archivo que da guardada en openFileDialog y se extrae para poderla guardar en una variable de tipo cadena, para poder hacer que en la pictureBox aparezca la imagen seleccionada (*ilustración 4*).

Nos pasamos a el botón de calcular que es donde se realiza la detección de los centroides, el coloreado de los círculos y sus centros, así como la impresión en la listBox de los centros encontrados en la imagen.

Lo primero que se necesita es recorrer la imagen en “X” y “Y” para poder analizar la existencia de un círculo negro, para eso utilizamos en primer for para poder movernos en “X” y “Y”, la variable “i” de mueve en “Y” y “j” en “X” y la variable “c” de tipo Color nos va a servir para analizar el color del pixel donde estamos (*ilustración 5*).

En a la instrucción if vemos como utilizamos “c” para ver si el pixel analizado es color negro utilizando el sistema RGB siendo 0 en todos los campos negro, el a se utiliza para ver si es sólido o translucido el color, se crean e inicializan variables para el cálculo del área del círculo (*ilustración 6*).

Em primer for mueve en "Y" hacia abajo hasta que se encuentre con un pixel blanco y para el ciclo, se calcula la posición media en el primer pixel negro encontrado y el ultimo; los siguientes dos tiene el mismo funcionamiento, solo que parten del punto medio en "Y" que se calculó y se desplazan a la izquierda y a la derecha y otro hacia la izquierda y después de que se rompan los ciclos se calcula la posición "X" media, con esos 2 puntos ("Y" y "X") se obtiene el centro del circulo (*ilustración 8*).

Primero se calcula mediante la dimensiones en "X" y "Y" del circulo una proporción a 5% del mismo; Se manda a llamar la función de colorear con los daros de cirulos completo para ponerlo de un color diferente al negro y que cuando el primer for repita el ciclo lo omita y la segunda llamada es para dibujar el centro del circulo a una relación del 5% del mismo (*ilustración 8*).

Después de actualiza la picturaBox, se le da un valor a la variable centro de tipo Point y se guarda en la lista (*ilustración 9*)

Se muestra cómo es que es el proceso de dibujado del circulo, primero se saca la distancia que hay de "X" a la derecha y "X" a la Izquierda y se hace lo mismo con "Y"; Se crea una variable de tipo Graphics para poder colorar el circulo, se crea un Brush para el color que vamos a utilizar para el circulo, se usa la función FillEllipse de Graphics para poder colorear el circulo, los parámetros que se piden son: color, "X" en la parte izquierda, "Y" en su parte alta, distancia en "X", distancia en "Y". (*ilustración 10*).

Las listBox se iguala a la lista de puntos para que aparezca en pantalla toda la lista completa en orden de entrada de datos (*ilustración 10*). impresión

Etapa 2:

I. Construcción del grafo:

Para la construcción del grafo primero se guardan todos los vértices que tendrá mediante su constructor el cual toma una lista de círculos que va agregando a su lista de vértices, esto se hace por media de un ciclo FOR el cual itera de 1 hasta N y dentro de FOR hace una asignación de que tomaremos como C (constante) para sacar su complejidad algorítmica (Ilustración 1).

La agregación de vértices en el grafo tiene una complejidad de: O(n)

Sin embargo, eso solo es para la asignación de vértices.

Para la construcción completa del grafo que es la asignación de aristas los vértices se usa otra función.

En la función que realiza la verificación de existencia y asignación de aristas toma la lista de vértices que teníamos ya creada en grafo y comienza a iterar por cada posición con dos FOR anidados, uno para origen y otro para el destino, estos dos FOR a pesar de estar

anidados funcionan de manera independiente ya que cada uno itera de i hasta N entonces tomaremos cada FOR como N , sin embargo como dentro del FOR más interno se realiza la validación de existencia de arista con otro sería una N más pero esta vez solo sería $N-1$ ya que cuando destino y origen son iguales no se realiza el FOR de validación (Ilustración 2) .

$$T(n) = n * n * (n-1)$$

$$\Leftrightarrow N^3$$

II. Solución par de puntos más cercanos (Ilustración 3):

En la solución de los dos puntos (Vértices) más cercanos primero se toma un vértice de la lista de vértices con un Foreach el cual hace que la primera iteración se dé i hasta N entonces ese Foreach se toma como N .

Sin embargo, como se ve en la ilustración se nota que tomamos el vértice del Foreach anterior para iterar sobre sus aristas los cual hace que están anidados y que el segundo Foreach sea dependiente del primero por los cual para sacar $T(n)$ en este caso se van a poner como sumatorias en lugar de solo N .

Las operaciones que se realizan dentro del Foreach más interno se tomaran como C (constante), entonces $T(n)$ queda de la siguiente forma:

$$T(n) = \sum_{i=1}^N \sum_{j=1}^i C$$

$$T(n) = C \frac{n(n+1)}{2}$$

Lo cual hace que O grande quede de la siguiente forma:

$$O(n^2)$$

III. Solución camino de 4 vértices (Ilustración 4):

Para la solución del camino de 4 vértices se tomo como base el mismo algoritmo anterior solo que este en lugar de iterar para 2 vértices con el Foreach como era el caso anterior, ahora iteramos para 4 entonces tendremos 4 sumatorias anidadas, sin embargo cabe recalcar que no podemos seguir iterando si los vértices ya

fueron tomados como origen en algún Foreach por eso con cada iteración le agregamos -1 para dejar claro que vamos a pasar por todos los vértices menos sobre los que ya se tomaron previamente.

Todas las operaciones de asignación realizadas en se tomarán como C (constante) y ya que solo el Foreach mas interno es el único que tiene so pone C

Entonces $T(n)$ estaría de la siguiente forma:

$$T(n) = \sum_{i=1}^n \sum_{j=1}^i \sum_{k=1}^{j-1} \sum_{h=1}^{k-2} C$$

Dicha sumatoria genera la siguiente O grande:

$$O(n^4)$$

Etapa 3:

- I. Recorrido en Profundidad (Ilustración 3):
La construcción del recorrido en profundidad se encuentra como métodos del grafo. El metodo cuenta con los parámetros de origen, que sirve para saber desde cual punto vértice del grafo se empieza a recorrer y una lista que se llama recorrido.

El método comienza agregando a una pila al origen y sobre ese se comienza a construir el recorrido, se extrae un elemento de la pila y ese ser convierte en actual, si actual no ha sido visitado se agrega a la lista de visitados, se agrega actual al recorrido y todas sus aristas se agregan a la pila.

Para genera $T(n)$ se toman en consideración la condición de que si el vértice esta visitado o no, por esa razón en la segunda sumatoria donde solo se hace si el vértice no esta visitado se coloca un medio, se esta tomando en cuenta de que solo son la mitad de los elementos que tenga la pila se va a realizar la agregación de aristas ya que como varios vértices tiene conexión a un solo vértice ese vértice puede ya estar visitado en alguna iteración parada

$$T(n) = \sum_{i=1}^N \left(\frac{1}{2} \sum_{j=1}^i C \right)$$

Se toma N como el número de vértices del grafo agregados a la pila e 'i' como las aristas de cada vértice.

Lo cual hace que O grande quede de la siguiente forma:

$$O(n^2)$$

II. Recorrido en Amplitud (Ilustración 4):

Para el caso del recorrido en amplitud la explicación de cómo funciona es la misma que en profundidad. Sin embargo, para el recorrido en amplitud los elementos que se van colocando como actual que es sobre el cual se van agrandando aristas se hace con una cola.

Al cambiar la pila por una cola es que hace que los recorridos salgan de diferente manera, ya que como se vio en el marco teórico a la hora de extraer elementos de dichas listas se extraen de maneras diferentes.

Por lo tanto, al contener las misma iteraciones que de manera similar T(n)

$$T(n) = \sum_{i=1}^N \left(\frac{1}{2} \sum_{j=1}^i c \right)$$

Se toma N como el número de vértices del grafo agregados a la pila e 'i' como las aristas de cada vértice.

Lo cual hace que O grande quede de la siguiente forma:

$$O(n^2)$$

Etapa 4:

I. Kruskal (Ilustración 1):

El algoritmo comienza con la inicialización de todos los candidatos, es decir agregar todas las aristas del grafo a una lista lo cual tiene una complejidad de N^2 siendo dependiente al número de aristas de cada vértice del grafo (Ilustración 2).

Después se ordena la lista con las aristas por peso de menor peso a mayor peso, lo cual tiene una complejidad de N^2 por ser dos ciclos anidados (Ilustración 3).

Después se inicializa el ARM, en este solo se agrega cada vértice en una lista (Ilustración 4), tiene una complejidad de N .

Una vez que ya se tienen las listas con la que va a trabajar para generar el ARM pasamos a un ciclo el cual se va a iterar de acuerdo a la cantidad de aristas que tengamos como candidatas para el ARM, se selecciona la arista con menor peso y se busca si el origen y el destino de esa arista generan un ciclo, sino se agregan al ARM; Todo esto genera una complejidad de $N(N^*N)$.

$$T(n) = N^2 + N^2 + N + N^2$$

Se toma la N con I .

Lo cual hace que O grande quede de la siguiente forma:

$$O(n^2)$$

II. PRIM (Ilustración 5):

Para Prim tenemos la misma inicialización de los candidatos y del ordenamiento de estos, después ya que Prim cuenta con un origen, ese origen se agrega al ARM.

Después iteramos hasta que el ARM tenga el mismo número de vértices que el grafo, dentro se busca cual es la arista con menor peso y que este conectada con el origen o el destino al ARM y esa arista se toma y se agrega al ARM.

Esta iteración tiene una complejidad de N^2

$$T(n) = N^2 + N^2 + N^2$$

Este tiempo nos genera una O grande que queda de la siguiente forma:

$$O(n^2)$$

Etapa 5:

I. Dijkstra (Ilustración 1):

El algoritmo inicia con la creación de una variable de tipo Dijkstra y una lista de elementos Dijkstra.

Después inicializamos la lista con una función que agrega los elementos Dijkstra (Ilustración 2) esta función tiene una complejidad de N es un Foreach que itera sobre el número de vértices del grafo.

Después tenemos el ciclo que nos actualiza los valores de la lista de elementos Dijkstra, la condición de paro es una función con una complejidad “constante” (Ilustración 3) solo es un return para saber si el vértice destino ya es definitivo y se puede hacer la regresión para generar el camino.

Después se asigna valor a la Dijkstra generada con anterioridad con una función que selecciona el elemento Dijkstra con el menor peso y que no sea definitivo (Ilustración 4) como solo se itera sobre la lista genera un N como complejidad.

Después de tener el elemento que se selecciona como definitivo Pasamos a actualizar nuestra lista con elementos Dijkstra (Ilustración 5) siempre y cuando el nuevo paso que genera el elemento actual sea menor que el que ya tiene, este proceso tiene una complejidad de N.

Por último, una vez que ciclo haya terminado se hacer la regresión sobre la lista con elementos Dijkstra para obtener el recorrido de la partícula (Ilustración 6).

$$T(n) = N + N(C + N + N)$$

El T(n) anterior genera una O grande que queda de la siguiente forma:

$$O(n^2)$$

Pruebas y Resultados:

Etapa 1:

1. Cargamos la primera imagen que contiene solo un punto y analizamos, se muestra como el círculo se pinta de color azul y su centro queda de color

amarillo, mientras que del lado derecho en la listBox muestra su centro (*ilustración 11, 12*).

2. Cargamos la segunda imagen que contiene varios puntos del mismo tamaño y analizamos, se muestra como poco a poco los círculos se pinta de color azul y su centro queda de color amarillo y del lado derecho en la listBox muestran los centros (*ilustración 13, 14, 15*).
3. Cargamos la tercera imagen que no contiene puntos y analizamos, en este caso no pinta nada por no contener circulos, mientras que del lado derecho en la listBox muestra vacío al no contener círculos (*ilustración 16, 17*).
4. Cargamos la cuarta imagen que contiene varios puntos de distintos tamaños y analizamos, se muestra como el circulo se pinta de color azul y su centro queda de color amarillo de manera proporcional al 5% del tamaño del circulo que se está pintando en ese momento, mientras que del lado derecho en la listBox muestran los centros (*ilustración 18, 19, 20*).
5. Por ultimo cargamos la 5ta imagen que contiene círculos de un tamaño tan grande que parece que se están tocando y analizamos, en este caso podía llegar a darse en error de que al estar muy grande se tomen dos círculos como si fueran uno y para solucionar esto se tenia que bajar la imagen en buena resolución y tener las condiciones de paro de los for de búsqueda mencionados en el desarrollo con diferente del color negro, se muestra como se van pintando los círculos y el lado derecho sus centros (*ilustración 21, 22, 23*).

Etapa 2:

1. Imagen 1 (Ilustración 5 y 6):

Primero cargamos la imagen, después damos click sobre el botón analizar. Una vez analizada la imagen nos muestra dentro de la imagen enumerados los vértices, las aristas de color verde, los dos puntos más cercanos de color naranja, y de color morado el camino.

Del lado derecho vemos que muestra los centros y debajo de los centros el grafo en forma de árbol con los adyacentes de cada vértice y dichos adyacentes con las distancias entre el origen y el destino.

2. Imagen 2 (Ilustración 7 y 8):

Al dar en analizar pinta lo mismo que en caso anterior y con los mismos colores.

En este caso se ve por primera vez de manera clara cómo funciona la localización de adyacentes para un vértice y como si hay una obstrucción no pinta la línea entre estos dos vértices y tampoco los coloca como

adyacentes del vértice, por esto mismo todos tiene diferentes números de adyacentes.

3. Imagen 3 (Ilustración 9 y 10):

Al analizar esta imagen se muestra lo que paso en el caso anterior, ya que 3 vértices entran encerrados en un cuadro en medio estos no se conectan con nadie salvo ellos 3

4. Imagen 4 (Ilustración 11 y 12):

Esta imagen muestra que se separo en 4 secciones los vértices en los cuales solo los que perteneces a la misma sección de imagen se conocen.

5. Imagen 5 (Ilustración 13, 14 y 15):

Esta imagen es la primera que al analizar no muestra un camino y por eso al analizar te manda un mensaje que dice que no existe el camino ya que no hay 4 vértices que se conecten en línea sin repetir vértices, todos los demás puntos se cumplen y de dibujan en la imagen.

6. Imagen 6 (Ilustración 16, 17, 18, 19):

Esta imagen tiene todos sus vértices aislados por eso al analizar manda el mensaje de que no hay camino y tampoco existen 2 puntos cercanos ya que nadie se conecta con nadie y esto se ve en la parte de grafo en la captura.

7. Imagen 7 (Ilustración 20, 21, 22):

Esta imagen al se un solo punto o vértice no tiene camino ni 2 puntos cercanos y nada por eso los dos mensajes de inexistencia.

8. Imagen 8 (Ilustración 23 y 24):

Esta imagen fue diseñada con el objetivo de hacer que el color rojo afectara la validación de obstrucción, sin embargo, al tener la condición como Color diferente de blanco en lugar de separar el RGB no ocurre dicha falla y realiza el análisis de forma normal.

9. Imagen 9 (Ilustración 25, 26, 27):

Esta imagen a pesar de ser 2 vértices no cuenta con 2 puntos cercanos ya que hay una obstrucción en medio.

10. Imagen 10 (Ilustración 28):

Esta imagen tiene los mismos elementos que la imagen anterior sin embargo esta esta echa para que la validación de obstrucción falle ya los puntos de análisis pasan por un lado del color de la línea, pero esto se solucionaba analizando un pixel arriba y abajo si analizas en X y en izquierda y derecha si analizas en Y

11. Imagen 11 (Ilustración 29 y 30):

La última imagen tiene las validaciones de obstrucción que se vieron en las imágenes anteriores, pero ya que si se valido de manera correcta muestra el análisis sin fallas.

Etapa 3:

1. Prueba 1 (Ilustración 5, 6, 7 y 8):

En las 2 primeras imágenes se muestra los recorridos ordinarios.
En las últimas 2 se muestra el recorrido que hacen las partículas.

2. Prueba 2 (Ilustración 9, 10, 11 y 12):

En las 2 primeras imágenes se muestra los recorridos ordinarios.
En las últimas 2 se muestra el recorrido que hacen las partículas.

3. Prueba 3 (Ilustración 13, 14, 15 y 16):

En las 2 primeras imágenes se muestra los recorridos ordinarios.
En las últimas 2 se muestra el recorrido que hacen las partículas.

4. Prueba 4 (Ilustración 17, 18, 19 y 20):

En esta prueba lo que se muestra es el comportamiento de las partículas depredadoras (verdes) y las partículas presas (moradas) cuando se encuentra en el mismo espacio.

Después de que se encuentran las depredadoras de comen a las presas y cambian de color, en este ejemplo se muestra que ya quedan menos presas y las depredadoras pasaron de verde a amarillo

Etapa 4:

1. Prueba 1 (Ilustración 6 y 7):

En la primera imagen vemos el camino que genera el algoritmo de Kruskal (Color Rojo) y como en el lado derecho superior se muestran las aristas que conforma el ARM junto con el peso total del ARM.

En la segunda imagen se muestra el camino que genero el algoritmo de Prim, como se ve es el mismo camino, sin embargo, a la hora de ver las aristas que conforman el ARM de Prim vemos como este tiene como primera arista una diferente a Kruskal, por lo tanto, su ARM se creó de manera diferente.

2. Prueba 2 (Ilustración 8 y 9):

El grafo que se genera de esta prueba ya tiene forma de árbol y dado a que los dos algoritmos generan arboles ambos algoritmos recubren todas las aristas del grafo, lo único que cambia es el origen de donde se generan los ARM, también tiene el mismo peso

3. Prueba 3 (Ilustración 10 y 11):

En esta prueba se ve que se selecciona de manera arbitraria las aristas horizontales ya que tienen el mismo peso y no importa cual se agarra para el ARM

4. Prueba 4 (Ilustración 12 y 13):

En esta prueba se analiza un grafo no conexo, entonces se modificó el algoritmo para que funcione con grafos no conexos y esto se da con las condiciones de paro en los ciclos que agregar las aristas al ARM.

Para Kruskal es que ya no haya aristas para agregar y en Prim es que todos los vértices estén conexos, con estas condiciones los algoritmos ya pueden funcionar en grafos no conexos.

5. Prueba 5 (Ilustración 14 y 15):

Esta prueba funciona igual que la anterior solo que esta vez el grafo es aún más no conexo.

6. Prueba 6 (Ilustración 16):

En esta última prueba como el grafo no contiene aristas no se crea ningún ARM, por eso en Prim y Kruskal se muestra un peso total de 0.

Etapa 5:

1. Prueba 1 (Ilustración 7, 8 y 9):

Se toma la primera imagen, se analiza y se crea una partícula con inicio en 0 y fin en 80 y vemos que como origen se coloca 80 por que llegó a su destino.

2. Prueba 2 (Ilustración 10, 11 y 12):

Se intentará hacer que la partícula presa llegue del vértice 1 a 15 y agregando 2 depredadoras., lamentablemente no llegue y es devorada y se muestra un mensaje de que se murió.

3. Prueba 3 (Ilustración 13, 14):

Para este caso también se agregan depredadores solo que esta vez es solo uno para intentar ver que la partícula presa llega a su destino y lo logra y se actualiza su origen.

4. Prueba 4 – Sin conexión entre vértices (Ilustración 15):

Esta prueba se realiza con un grafo no conexo y se intenta llegar de un subgrafo a otro, dado a que no hay conexión se muestra en pantalla un mensaje emergente.

Conclusiones:

Etapa 1:

C# es un lenguaje en el cual desarrollar software se hace de manera sencilla ya que ofrece muchos elementos que facilitan el desarrollo.

Existe un apartado de diseño en el cual puedes ver como queda se ve de manera visual lo que se está desarrollando, este apartado de diseño es muy importante ya que no solo te deja visualizar tu trabajo, sino que te permite agregar elementos diversos elementos a la ventana sobre la que están trabajando sin necesidad de hacer la declaración en el código fuente ya que C# lo agrega de manera automática

El otro punto que también es muy importante es que en C# las distintas librerías contienen una gran cantidad de tipos de plantillas de clases que contiene c#, las cuales te permiten almacenar distinta información de manera sencilla, puedes guardar desde imágenes, colores, listas, eventos.

Etapa 2:

Al realizar el análisis de la complejidad algorítmica de lo que es la construcción del grafo y los algoritmos de fuerza bruta que se utilizan para encontrar el par de puntos mas cercanos y el amino de cuatro vértices te das cuenta de como es que se ve en tiempo de ejecución el programa ya sea que tenga una complejidad cubica como es en caso de la creación del grafo.

Con esto nos damos cuenta de que los algoritmos de fuerza bruta que al parecer son fáciles de implementar consumen muchos recursos del sistema y tiempo estos algoritmos esta bien simple y cuando no se tengan demasiados datos, sino el tiempo durante el cual una tarea se ejecuta se eleva de manera exponencial.

Etapa 3:

Los recorridos de los gafos lo que generan son arboles si ve de manera grafica parten de un origen que es el que se le considera padre y se van desplazando hacia su hijos, y dependiendo de si es de profundidad o de amplitud es donde vemos si se construye agarrando un nuevo origen cuando pasar al hijo o si primero agregas al árbol a todos sus hijos y de pues a uno de sus hijos lo colocas como origen.

Para la parte de hacer la animación de las partículas se tuvo que tomar como base los recorridos ya que las partículas hacen recorridos, sin embargo, se tuvieron modificar los métodos para hacer que se regresen.

En profundidad se agregó una pila para saber por qué vértices había pasado y una condición de que si el vértice donde se encuentra tiene conexión con el siguiente vértice se agrega al recorrido de manera ordinaria, sino buscas cual vértice tiene

la conexión y la búsqueda se hace desopilando los elementos de la pila extra que se creó.

Para profundidad se tuvo que crear la implementación de caminos, para un camino es el mismo código que un recorrido, solo que un camino tiene un origen y un fin y que la condición paro es que cuando llegue al destino ya no siga agregando vértices al recorrido. Agregando el camino entre cada vértice que se agregaba en la implementación del recorrido se obtiene el desplazamiento de la partícula.

Etapa 4:

Prim y Kruskal son algoritmos que al parecer generan las mismas soluciones ya que ambos van agarrando los vértices con menor peso, sin embargo, con el hecho de que Prim no pueda seleccionar aristas fuera de sus componentes conexos hace que su implementación en la vida sirva para obtener diferentes resultados.

Etapa 5:

Apéndice(s):

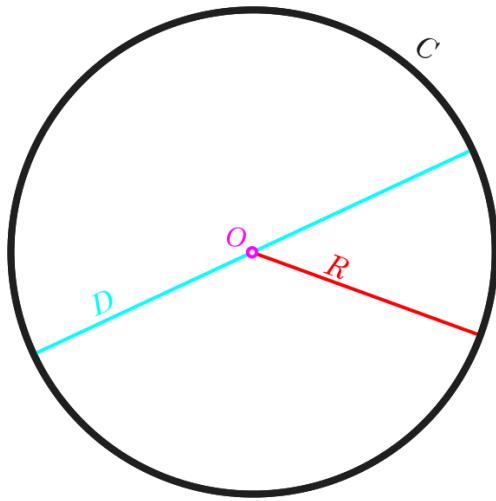


Ilustración 1

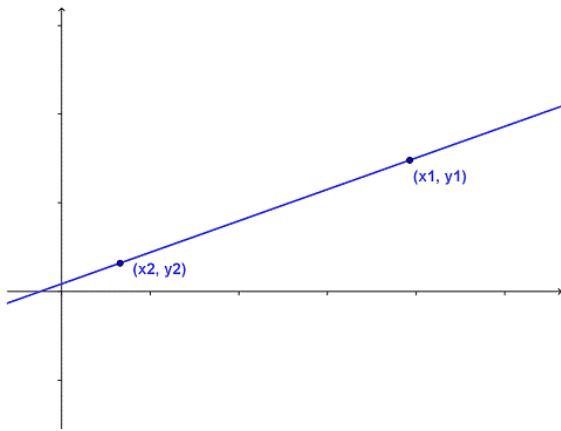


Ilustración 2

$$m = \frac{y - y_1}{x - x_1}$$

$$(x - x_1)m = \frac{(y - y_1)}{(x - x_1)}(x - x_1)$$

$$(x - x_1)m = (y - y_1)$$

$$(y - y_1) = m(x - x_1)$$

Ilustración 3

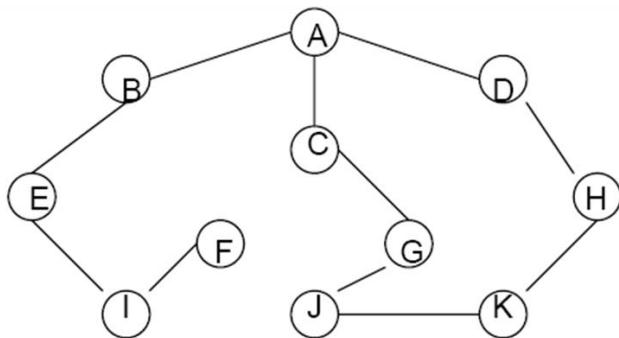


Ilustración 4

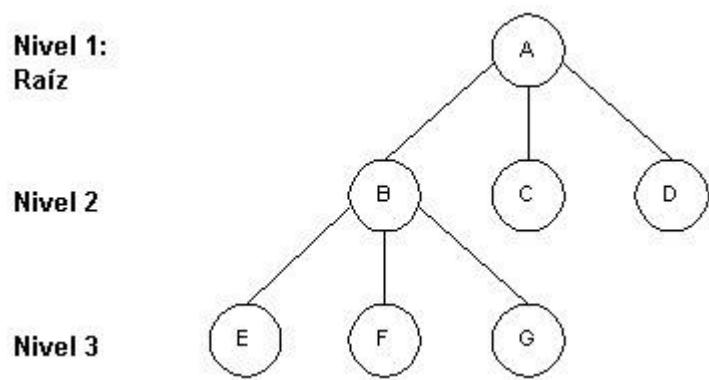


Ilustración 5

Etapa 1:

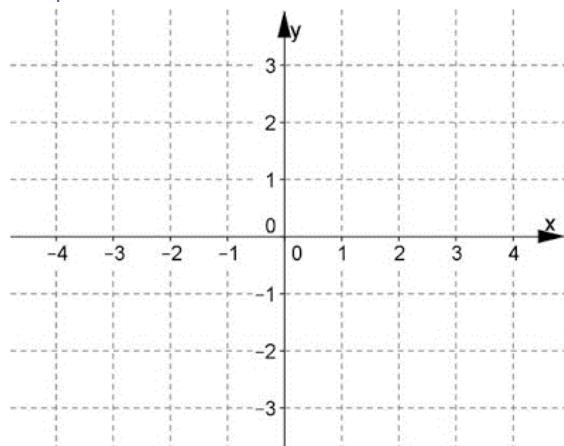


Ilustración 1

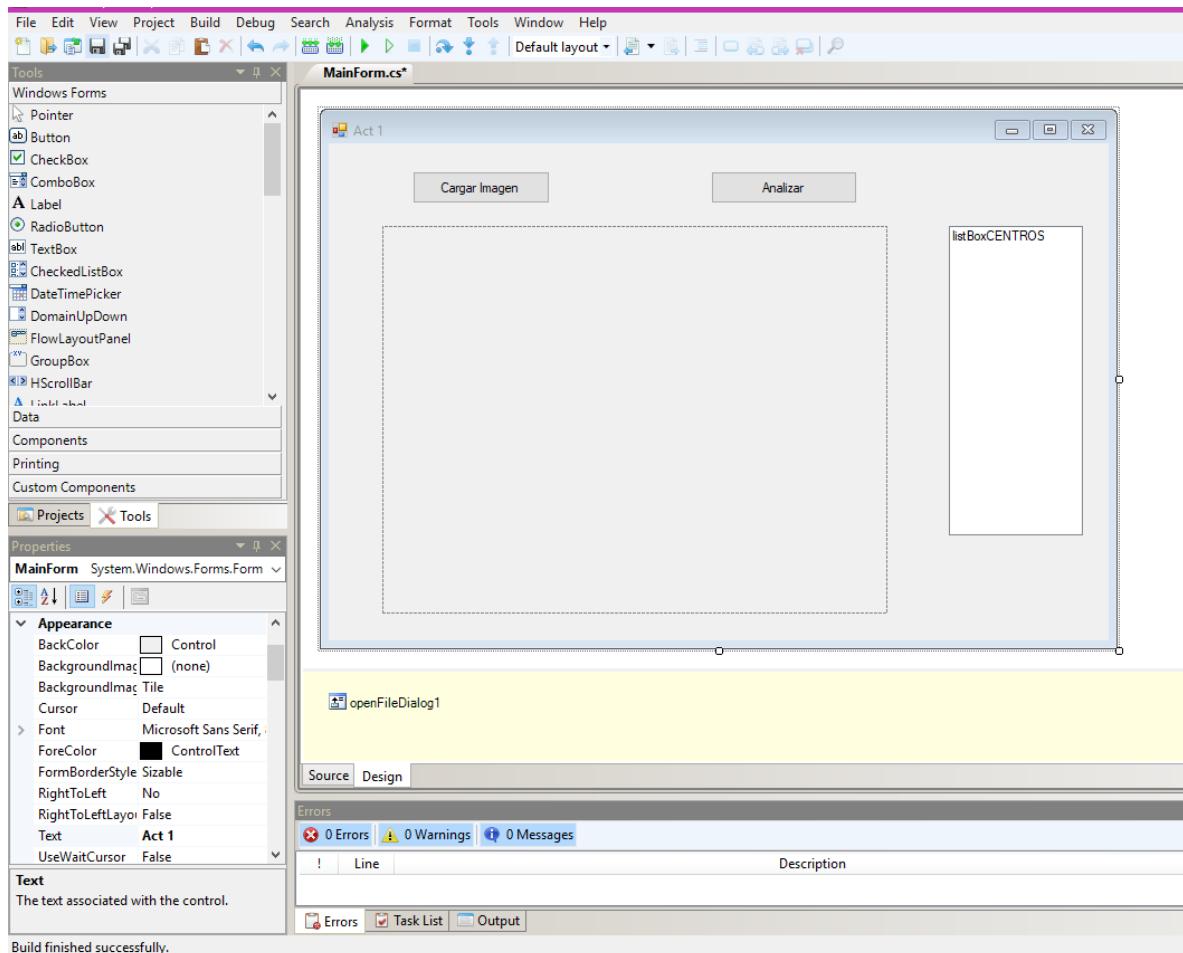


Ilustración 2

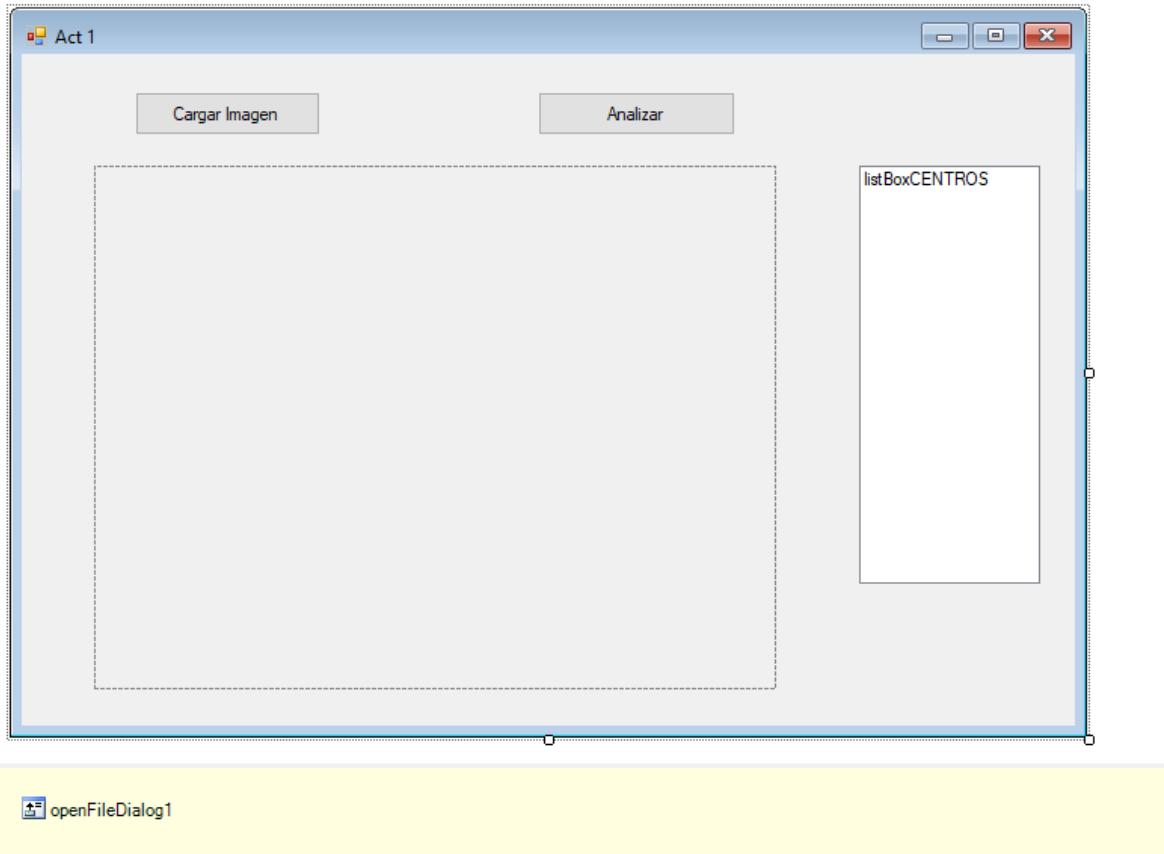


Ilustración 3

```
void ButtonCARGAClick(object sender, EventArgs e)
{
    openFileDialog1.ShowDialog();
    ruta = openFileDialog1.FileName;
    pictureBox1.Image = Image.FromFile(ruta);

}
```

Ilustración 4

```
for (int i = 0; i < bmpImagen.Height; i++) {      // Height = alto      i = y
    for (int j = 0; j < bmpImagen.Width; j++) {      // Width   = ancho      j = x
        Color c = bmpImagen.GetPixel(j, i);
```

Ilustración 5

```
if ((c.R == 0 && c.G == 0 && c.B == 0) && c.A == 255) {
    int y_i = i, y_f = i,
        x_izq = j, x_der = j,
        y_m, x_m;
```

Ilustración 6

```

for (; y_f < bmpImagen.Height; y_f++) { // Movimiento de Y hacia abajo
    c = bmpImagen.GetPixel(j, y_f);
    if (!(c.R == 0 && c.G == 0 && c.B == 0) || c.A != 255) {
        break;
    }
}

y_m = (y_f + y_i) / 2; // Calculo de Y medio

for (; x_izq >= 0; x_izq--) { // Movimiento de X hacia la izquierda
    c = bmpImagen.GetPixel(x_izq, y_m);
    if (!(c.R == 0 && c.G == 0 && c.B == 0) || c.A != 255) {
        break;
    }
}

for (; x_der < bmpImagen.Width; x_der++) { // Movimiento de X hacia la derecha
    c = bmpImagen.GetPixel(x_der, y_m);
    if (!(c.R == 0 && c.G == 0 && c.B == 0) || c.A != 255) {
        break;
    }
}

x_m = (x_der + x_izq) / 2; // Calculo de X medio

```

Ilustración 7

```

// Calculo del 5% del diametro para dibujar el centro
Y5 = y_m - ((y_m - y_i) / 10);
DY = y_m + ((y_m - y_i) / 10);
X5 = x_m - ((x_m - x_izq) / 10);
DX = x_m + ((x_m - x_izq) / 10);

bmpImagen = Colorear(y_i, y_f, x_der, x_izq, bmpImagen, Color.Cyan); // Colorear el circulo
bmpImagen = Colorear((int)Y5, (int)DY, (int)DX, (int)X5, bmpImagen, Color.Yellow); // Colorear el centro

pictureBox1.Image = bmpImagen; // Actualizacion del PictureBox

Point centro = new Point(x_m, y_m); // Punto que contiene el centro del circulo
lista_centros.Add(centro); // Adicion del punto a una lista

```

Ilustración 8

```

Bitmap Colorear(int Sup_Y, int Inf_Y, int Der_X, int Izq_X, Bitmap Original ,Color Relleno)
{ //Funcion para colorear un circulo encontrado

    float win = (Der_X - Izq_X) + 2;
    float hei = (Inf_Y - Sup_Y) + 2;

    Graphics Circle = Graphics.FromImage(Original);
    Brush randomBrush = new SolidBrush(Relleno);

    Circle.FillEllipse(randomBrush, Izq_X - 1 , Sup_Y - 1, win, hei);

    return Original;
}

```

Ilustración 9

```
listBoxCENTROS.DataSource = Lista_centros; // Imprimir lista de puntos
```

Ilustración 10

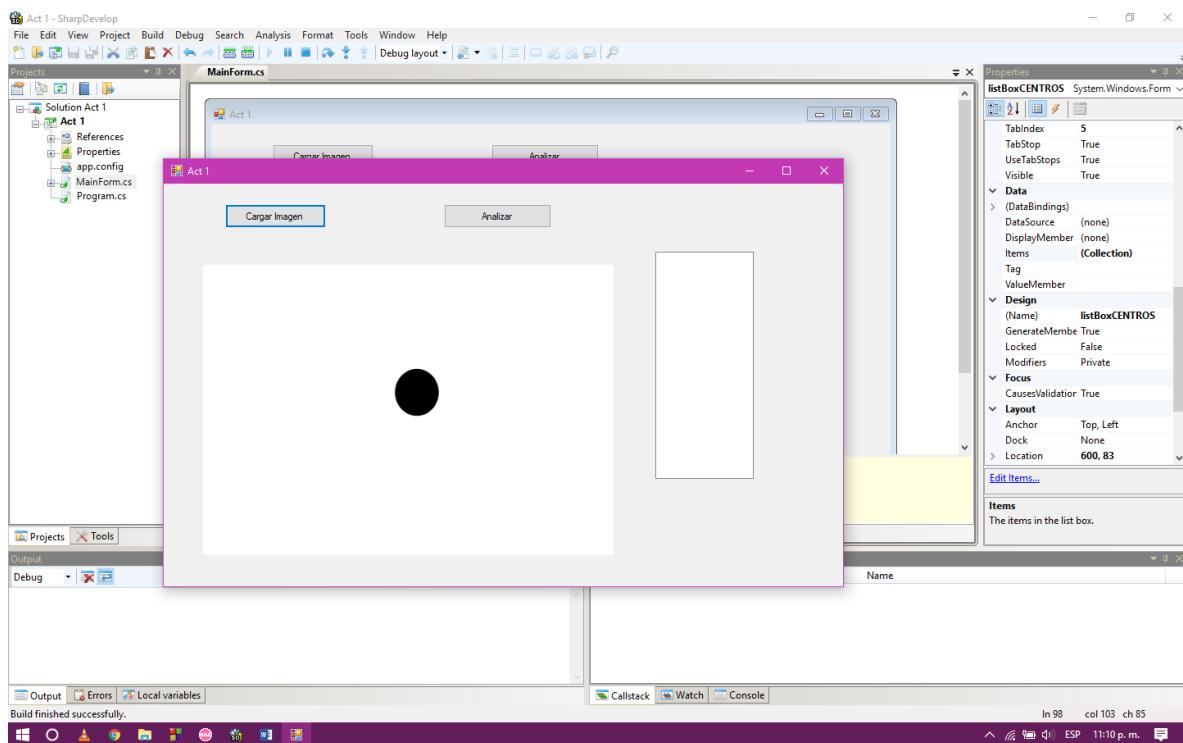


Ilustración 11

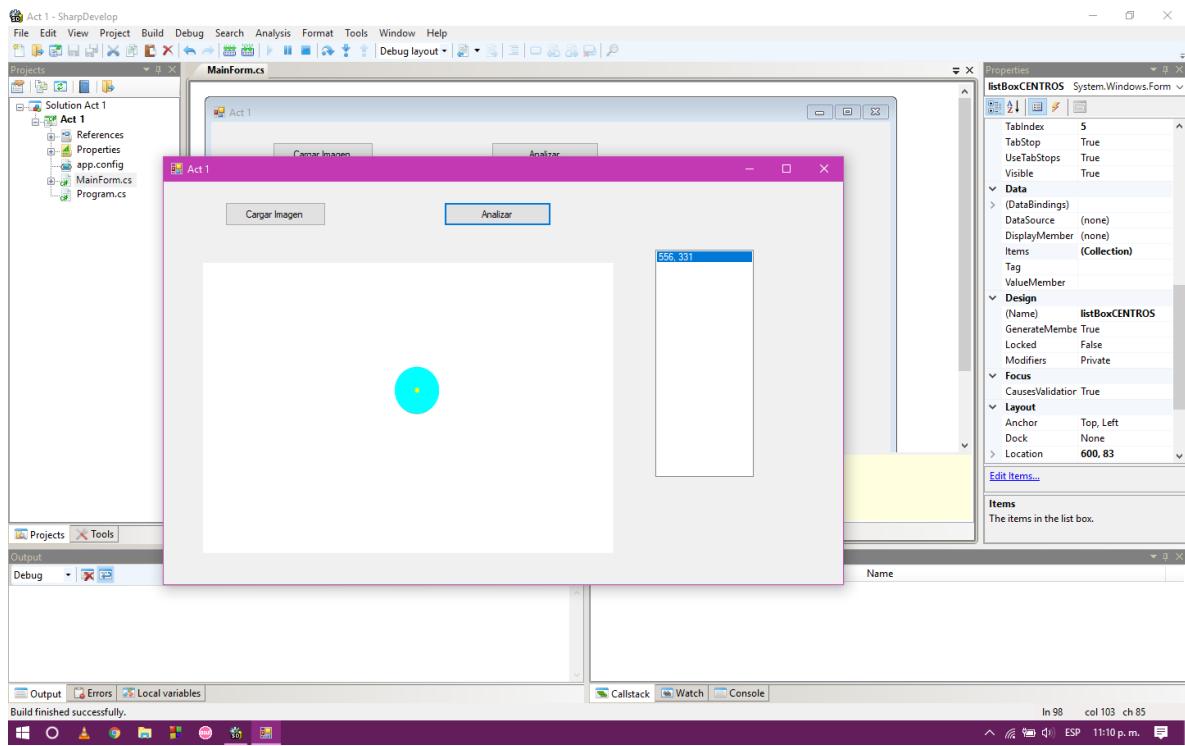


Ilustración 12

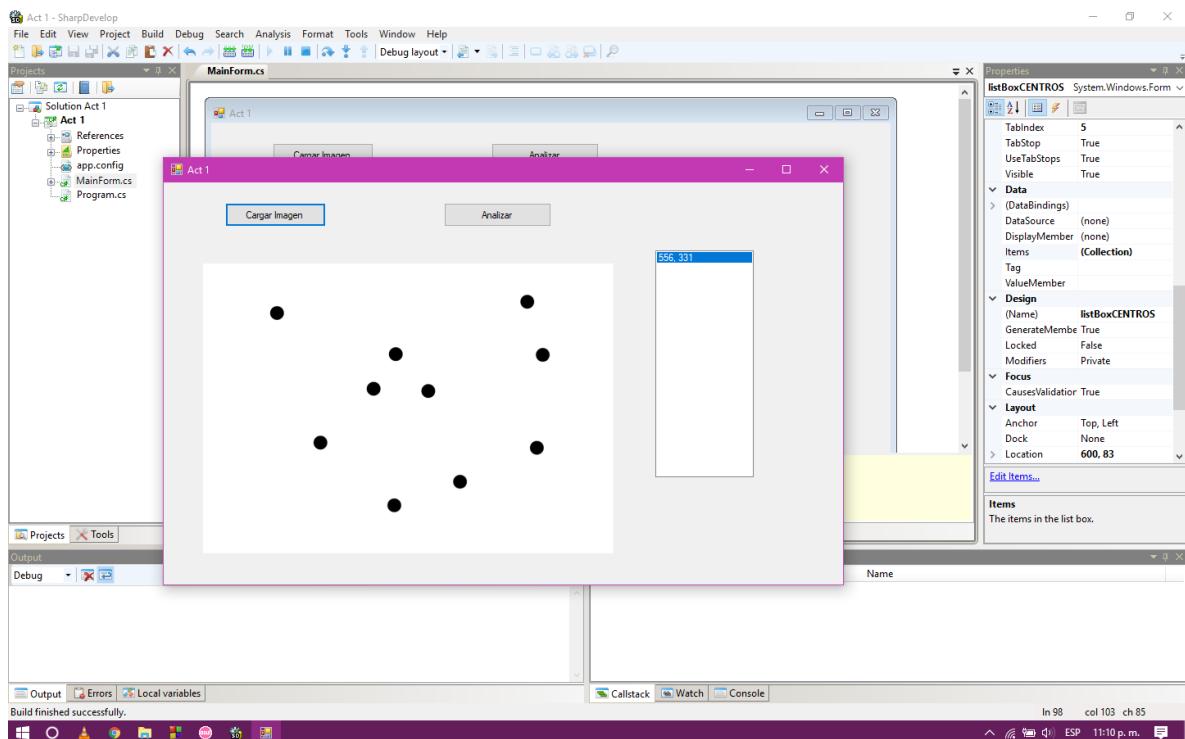


Ilustración 13

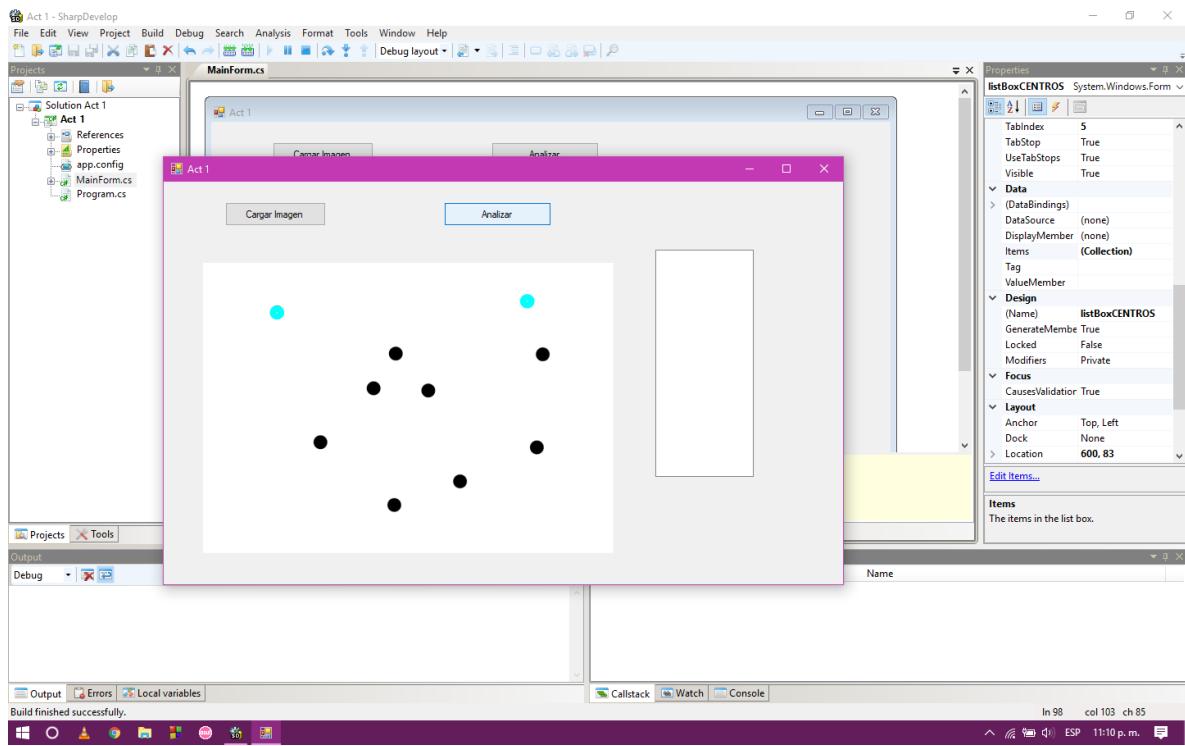


Ilustración 14

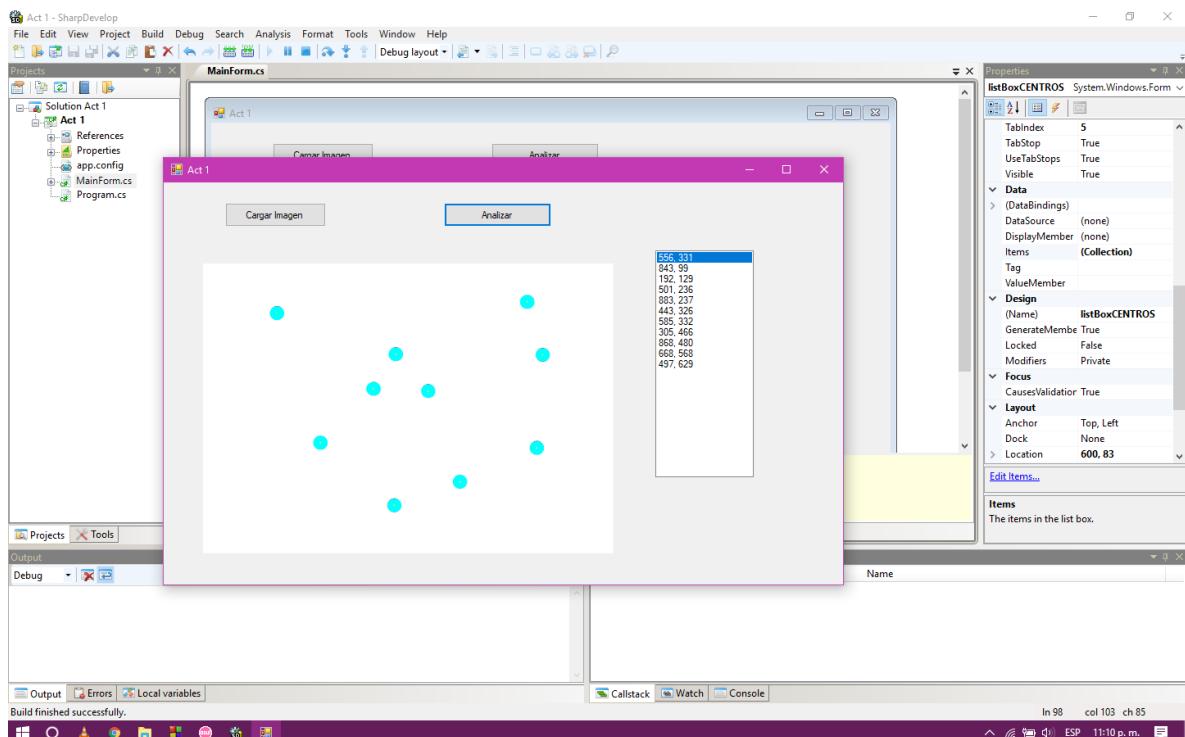


Ilustración 15

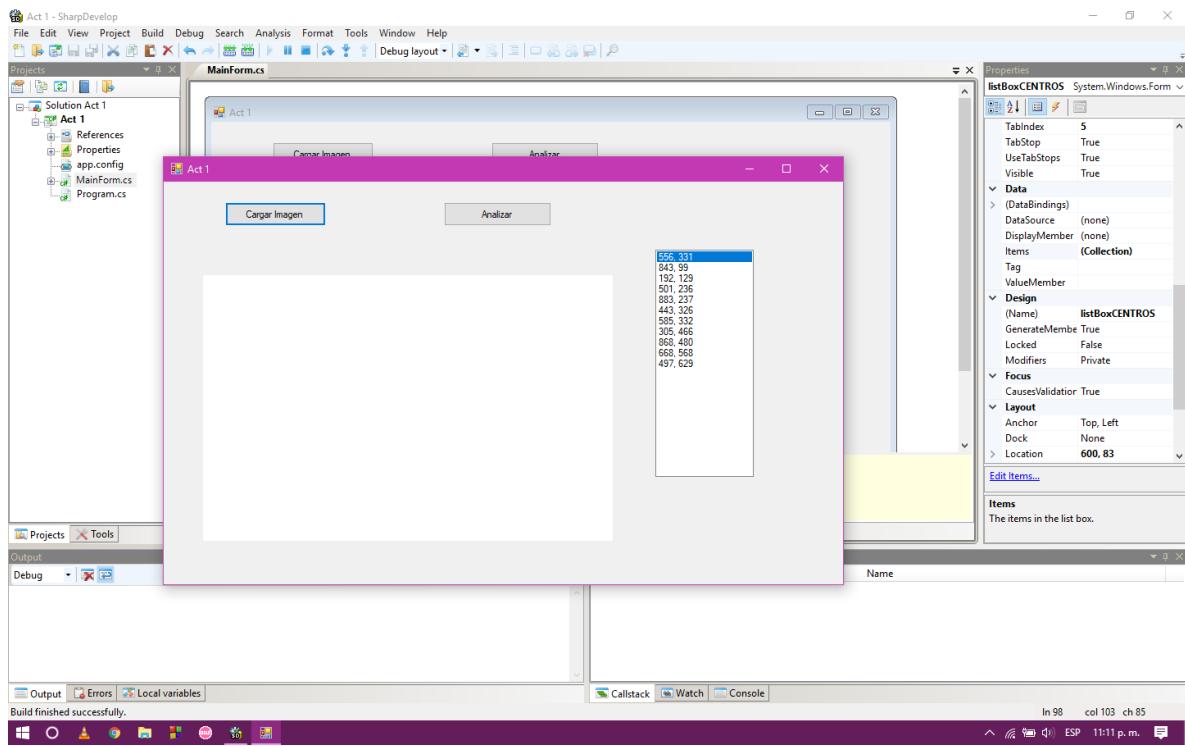


Ilustración 16

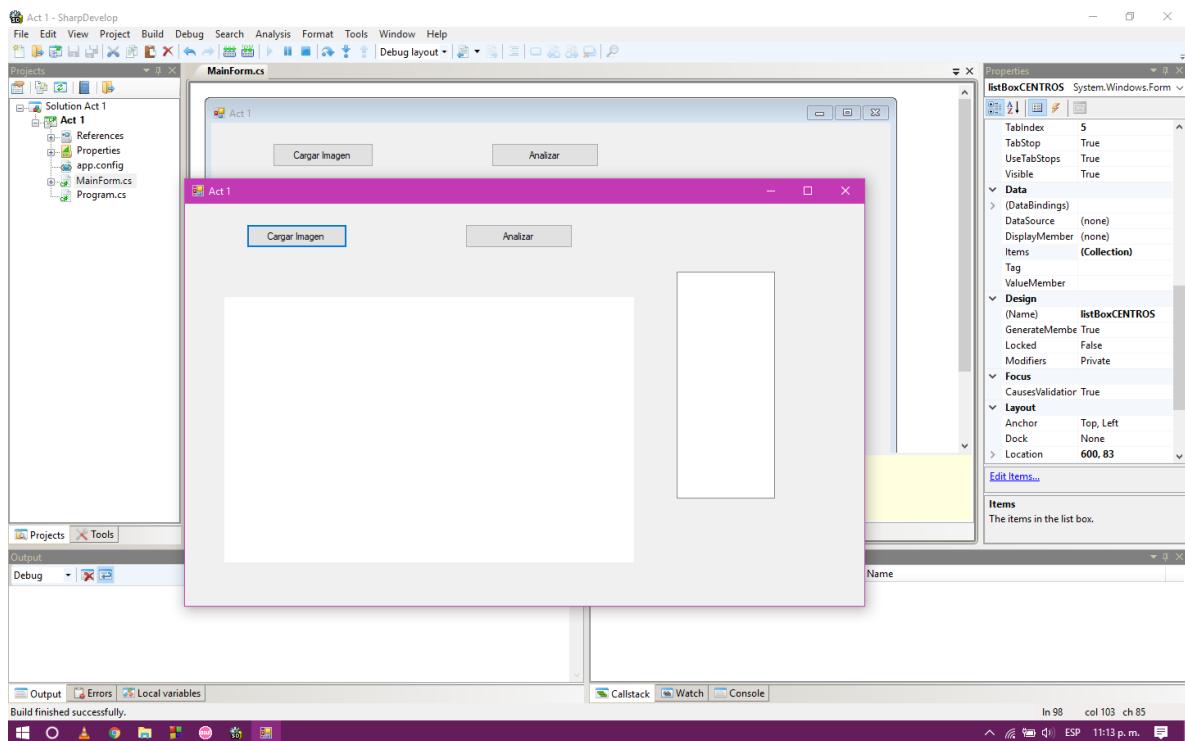


Ilustración 17

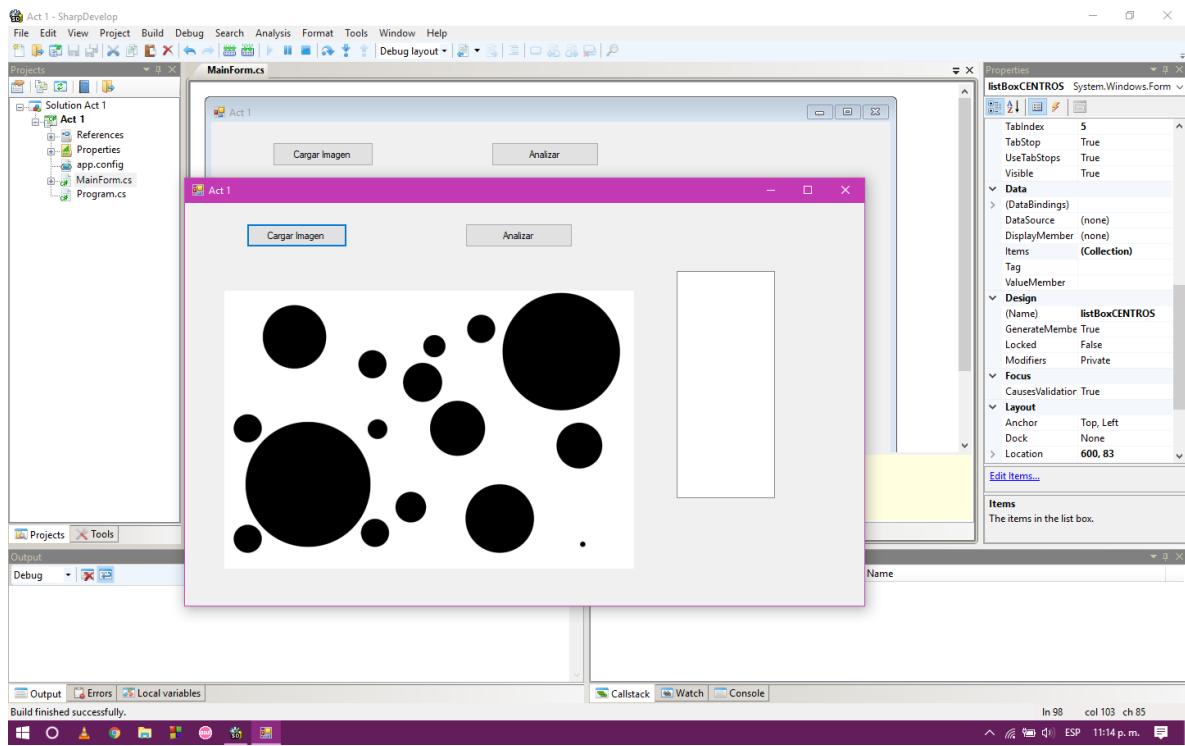


Ilustración 18

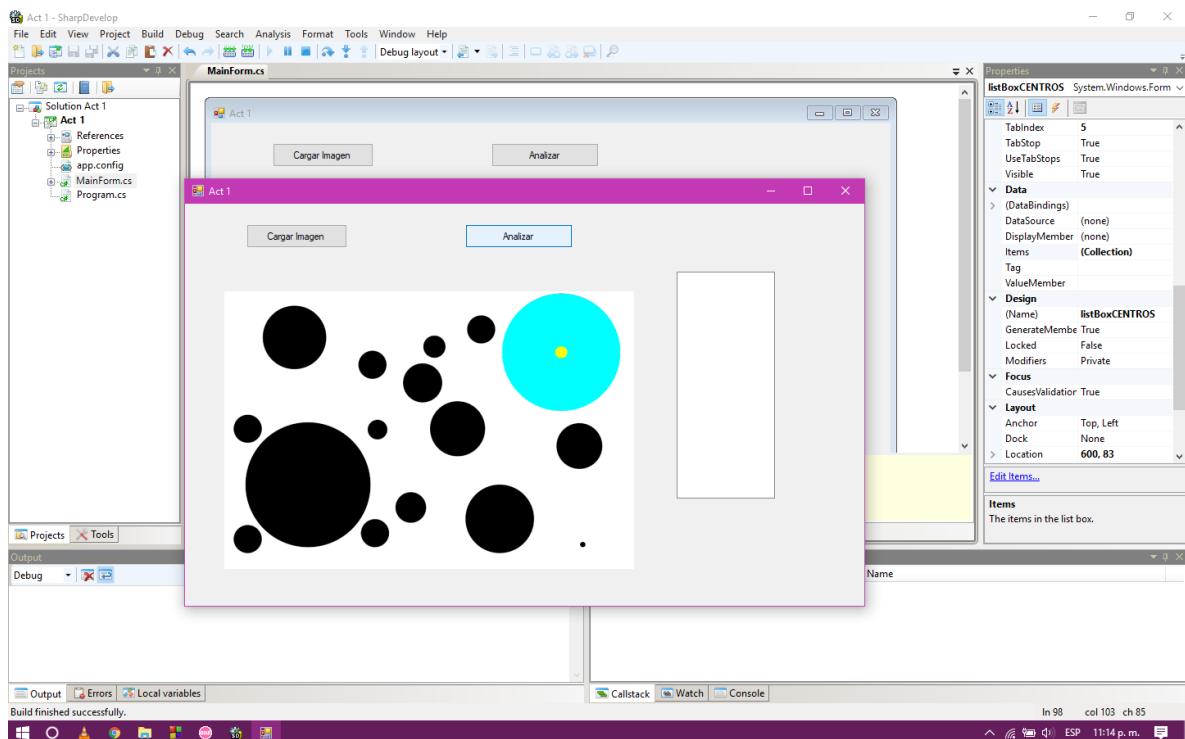


Ilustración 19

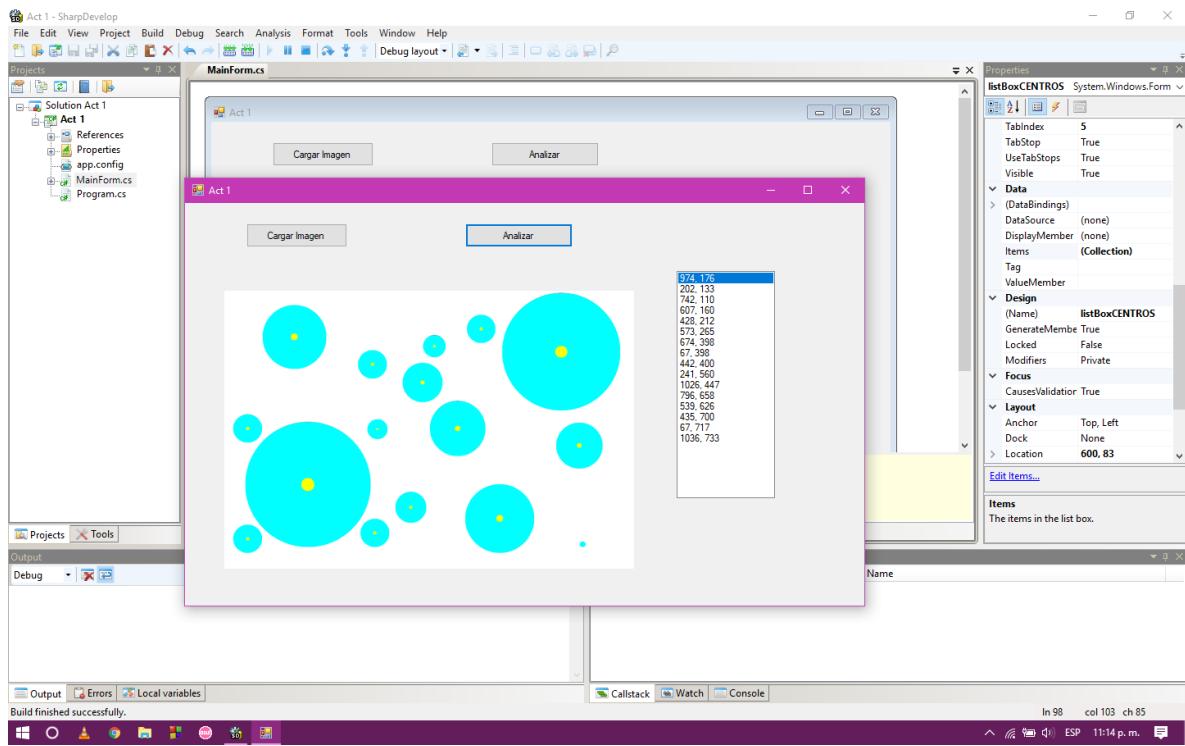


Ilustración 20

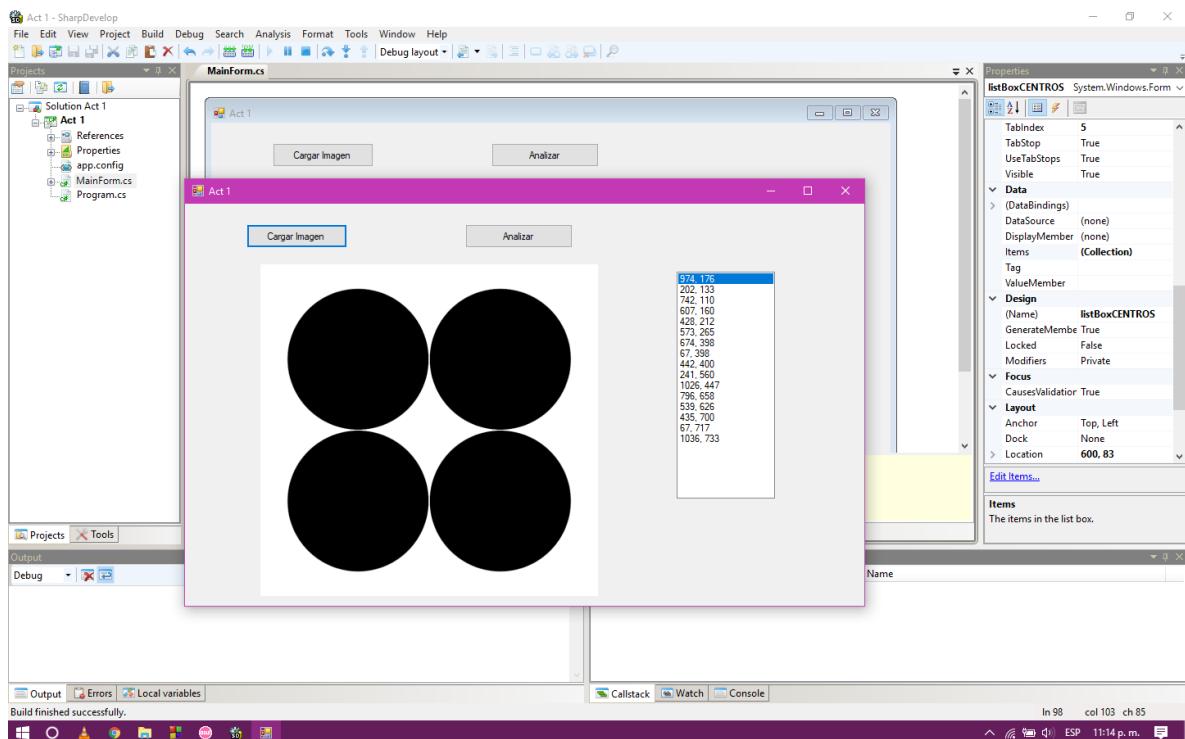


Ilustración 21

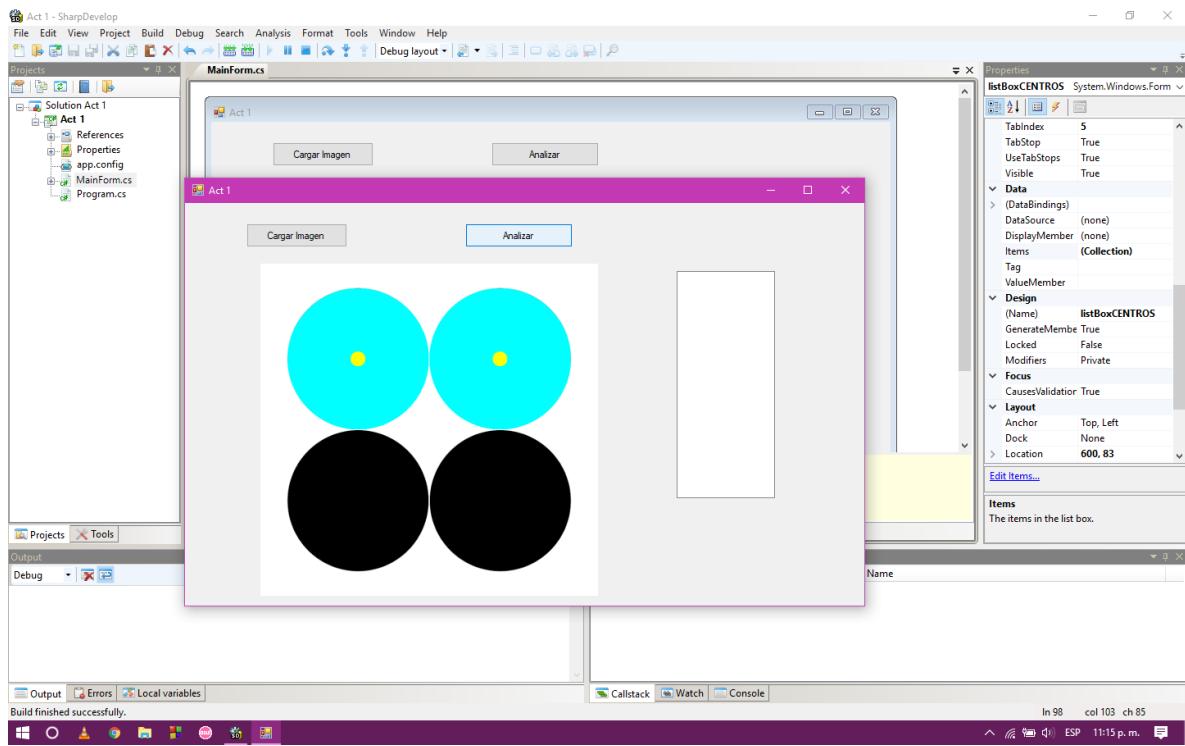


Ilustración 22

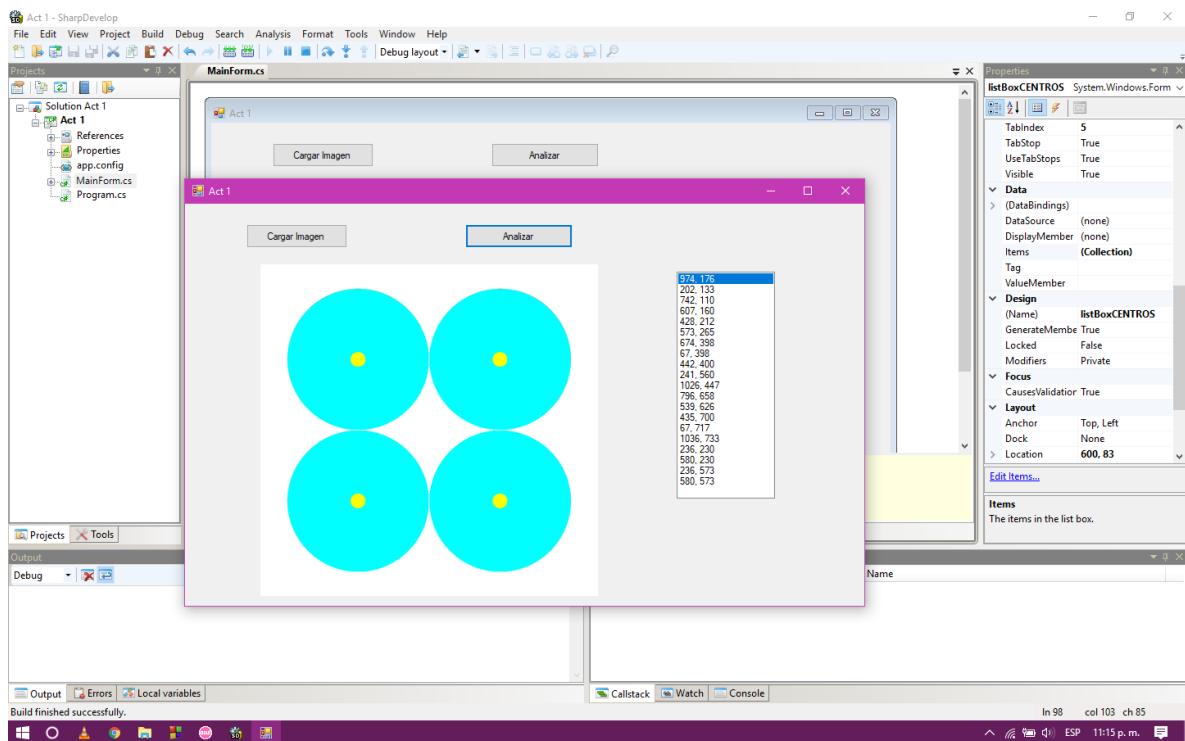


Ilustración 23

Etapa 2:

```
..  
public class Grafo  
{  
    // disable once FieldCanBeMadeReadOnly.Local  
    List<Vertice> listaVertices;  
  
    public Grafo(List<Circulo> listaCirculos)  
    {  
        listaVertices = new List<Vertice>();  
  
        foreach (Circulo auxCirculo in listaCirculos) {  
            // disable once SuggestUseVarKeywordEvident  
            Vertice nuevoVertice = new Vertice(auxCirculo);  
            listaVertices.Add(nuevoVertice);  
        }  
    }  
}
```

Ilustración 1

```
foreach (Vertice AuxVerticeOrigen in originalGrafo.GetListaVertices()) {  
    List<Arista> Adyacentes = new List<Arista>(); // Lista para insertar en Vertice origen  
  
    foreach (Vertice AuxVerticeDestino in originalGrafo.GetListaVertices()) {  
        List<Point> Pixelles = new List<Point>();  
  
        if (AuxVerticeOrigen.GetOrigen() != AuxVerticeDestino.GetOrigen()) {  
            Y0 = AuxVerticeOrigen.GetOrigen().GetY(); // Y vertice origen  
            X0 = AuxVerticeOrigen.GetOrigen().GetX(); // X vertice origen  
            Y1 = AuxVerticeDestino.GetOrigen().GetY(); // Y vertice destino  
            X1 = AuxVerticeDestino.GetOrigen().GetX(); // X vertice destino  
            RadioOrigen = AuxVerticeOrigen.GetOrigen().GetRadio() + 4; // Radio del circulo A  
            RadioDestino = AuxVerticeDestino.GetOrigen().GetRadio() + 4; // Radio del circulo B  
            Pendiente = (Y1 - Y0) / (X1 - X0); // Calculo de pendiente  
  
            if ((-1 < Pendiente) && (Pendiente < 1)) { // ***** If para dibujo en los cuadrantes de X  
                if (X1 > X0) {  
                    RadioDestino = X1 - RadioDestino;  
                    Pintar = true;  
                    for (int i = (int)X0; i < X1; i++)...  
                }
```

Ilustración 2

```

int Distancia = 0;
Boolean Primero = true;
Circulo[] Par = new Circulo[2];
Draw drawPar = new Draw();

foreach (Vertice AuxVertice in originalGrafo.GetListaVertices()) {

    foreach (Arista AuxArista in AuxVertice.GetListaAristas()) {

        if (AuxVertice != AuxArista.GetDestino()) {
            if (Primero) {
                Distancia = AuxArista.GetDistancia();
                Par[0] = AuxVertice.GetOrigen();
                Par[1] = AuxArista.GetDestino().GetOrigen();
                Primero = false;
            } else if (Distancia > AuxArista.GetDistancia()) {
                Distancia = AuxArista.GetDistancia();
                Par[0] = AuxVertice.GetOrigen();
                Par[1] = AuxArista.GetDestino().GetOrigen();
            }
        }
    }
}

```

Ilustración 3

```

foreach (Vertice AuxVertice in originalGrafo.GetListaVertices()) {
    AuxPuntos[0] = AuxVertice;

    foreach (Arista AuxArista_1 in AuxVertice.GetListaAristas()) {
        AuxPuntos[1] = AuxArista_1.GetDestino();
        AuxCamino[0] = AuxArista_1;
        AuxDistancia += AuxArista_1.GetDistancia();

        foreach (Arista AuxArista_2 in AuxArista_1.GetDestino().GetListaAristas()) {
            if (AuxArista_2.GetDestino() != AuxPuntos[0]) {

                AuxPuntos[2] = AuxArista_2.GetDestino();
                AuxCamino[1] = AuxArista_2;
                AuxDistancia += AuxArista_2.GetDistancia();

                foreach (Arista AuxArista_3 in AuxArista_2.GetDestino().GetListaAristas()) {
                    if (AuxArista_3.GetDestino() != AuxPuntos[0] && AuxArista_3.GetDestino() != AuxPuntos[1]) {

                        AuxPuntos[3] = AuxArista_3.GetDestino();
                        AuxCamino[2] = AuxArista_3;
                        AuxDistancia += AuxArista_3.GetDistancia();

                        if (Primero)... else if (Distancia > AuxDistancia)...}

                        AuxPuntos[3] = null;
                        AuxCamino[2] = null;
                        AuxDistancia -= AuxArista_3.GetDistancia();
                    }
                }
            }
        }
    }
}

```

Ilustración 4

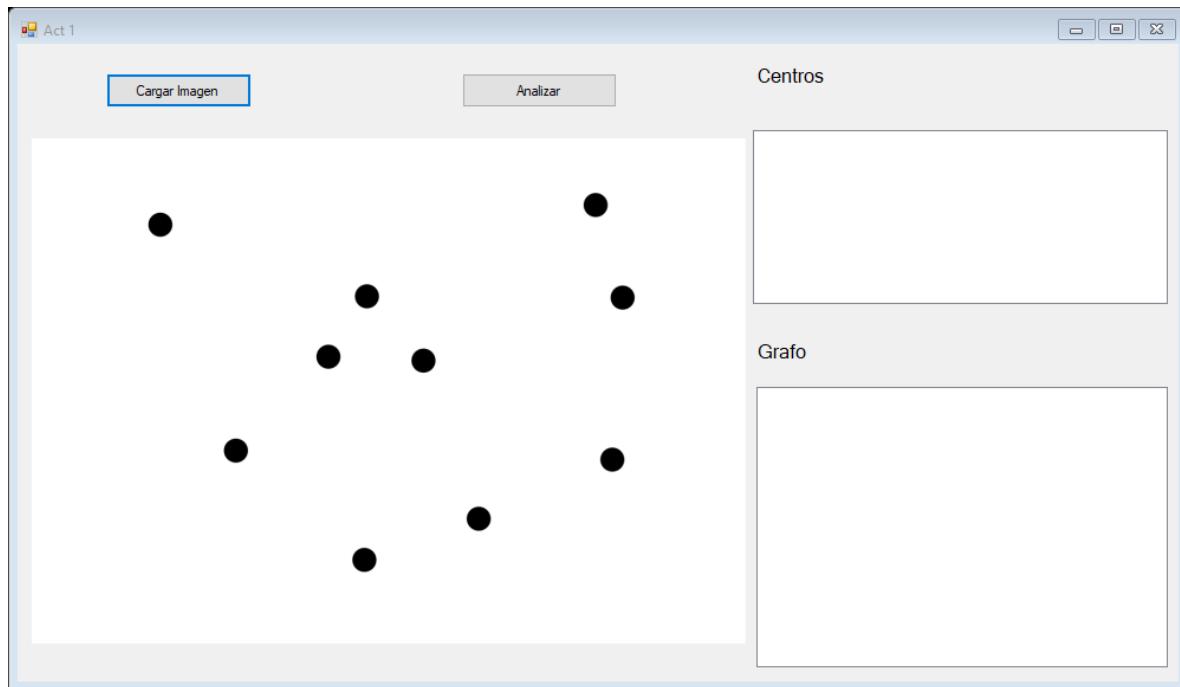


Ilustración 5

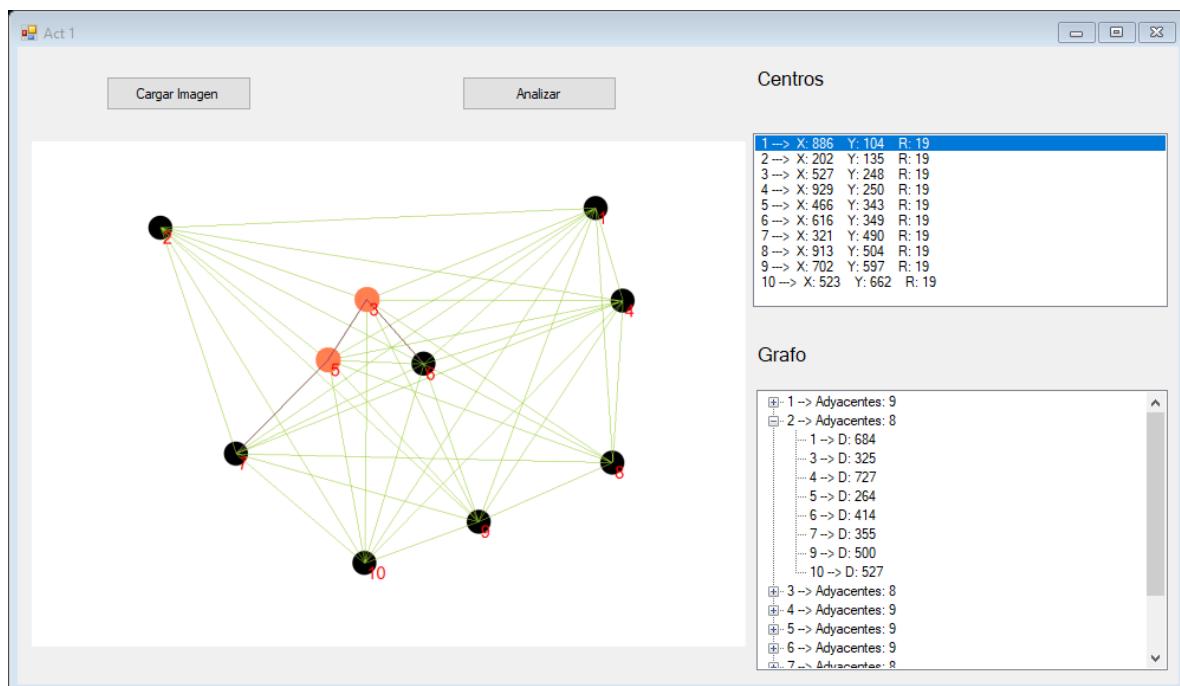


Ilustración 6

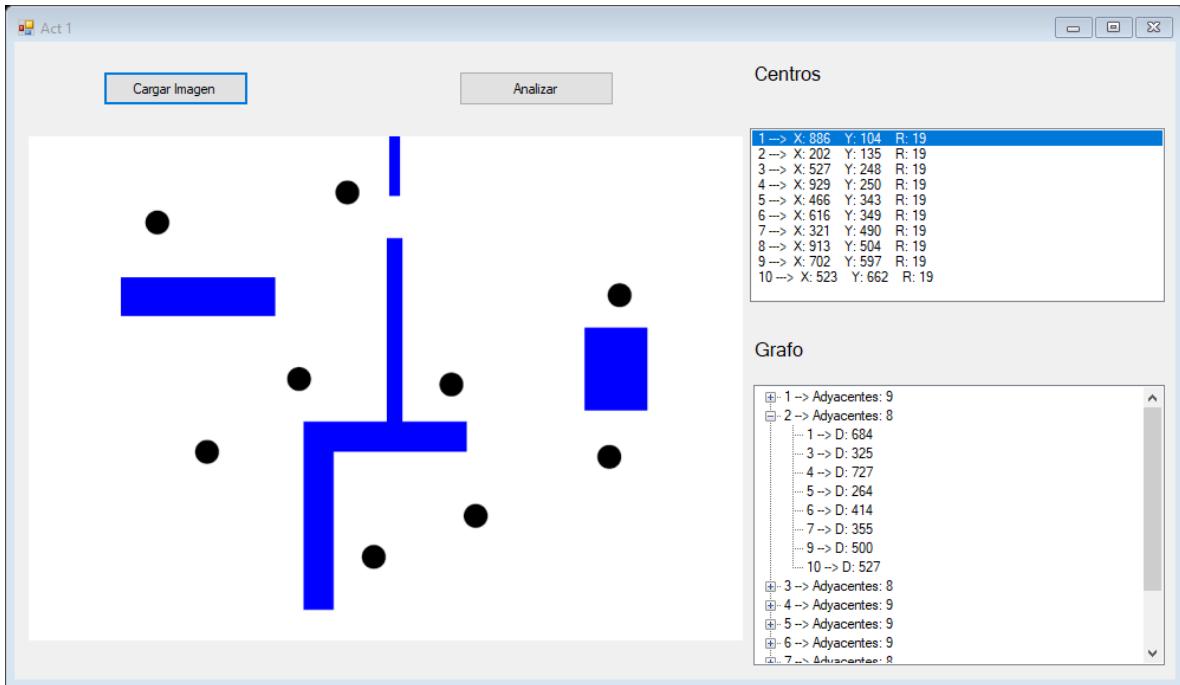


Ilustración 7

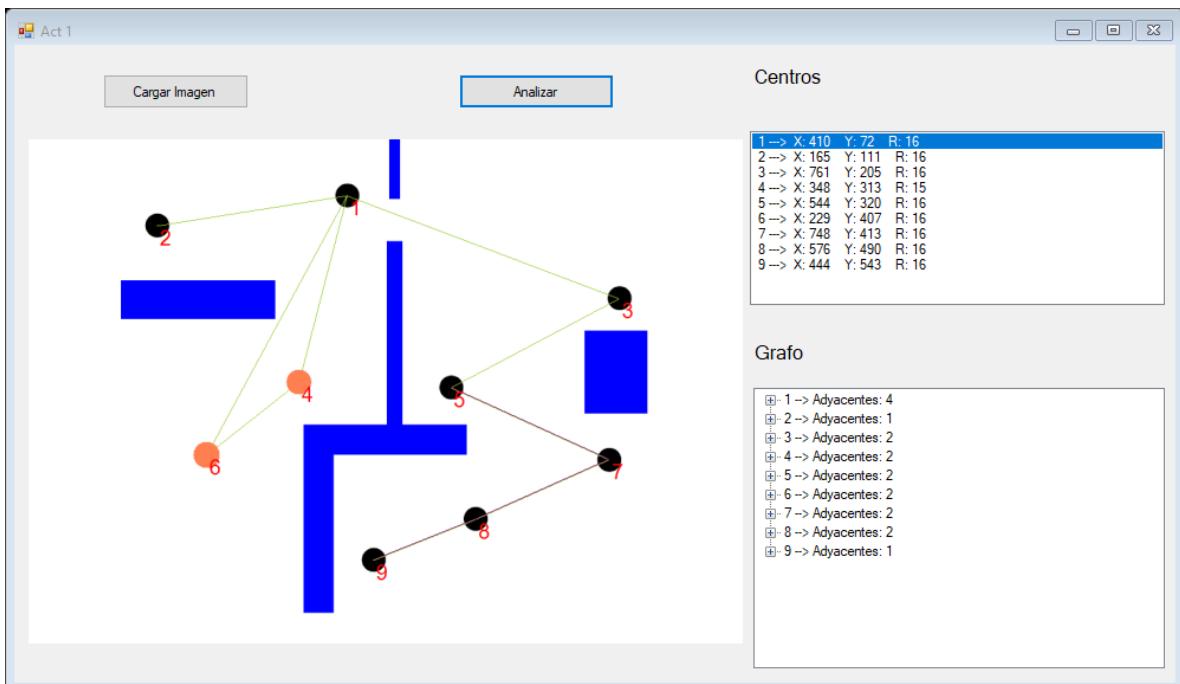


Ilustración 8

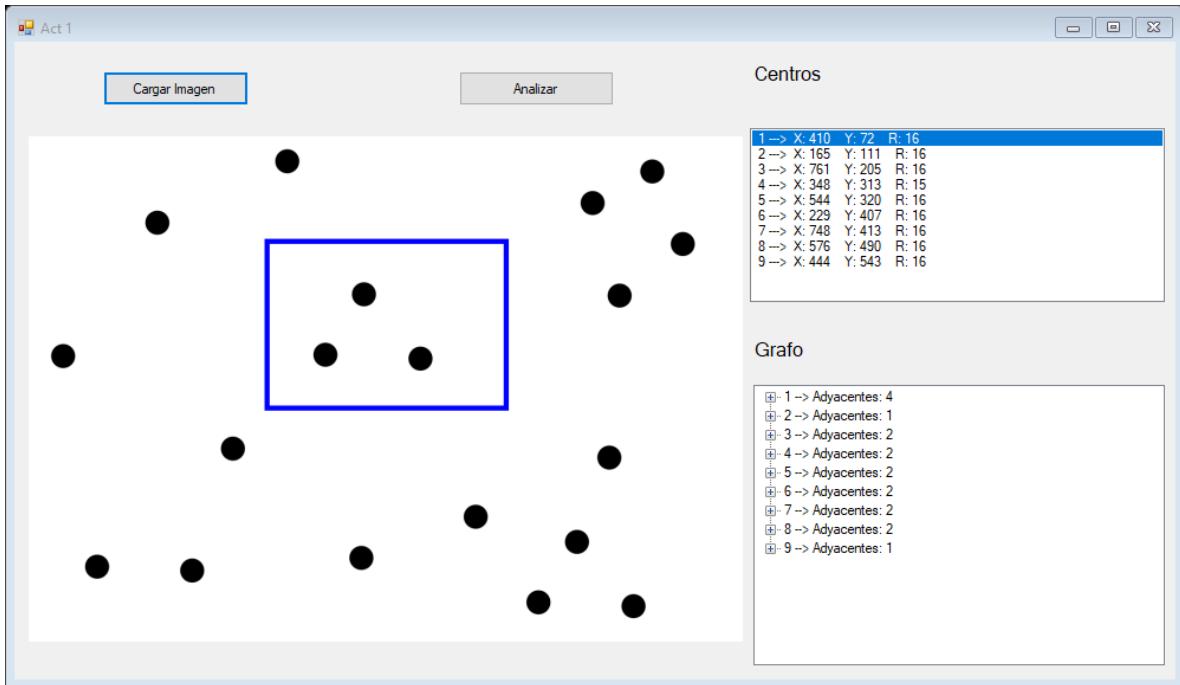


Ilustración 9

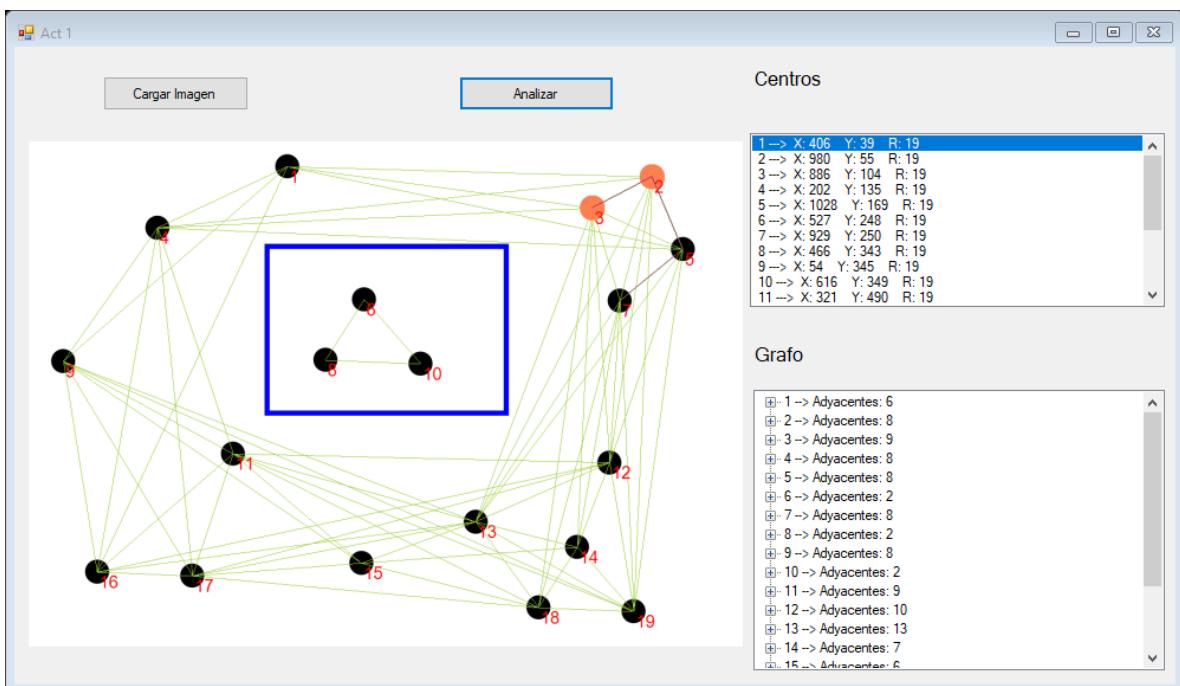


Ilustración 10

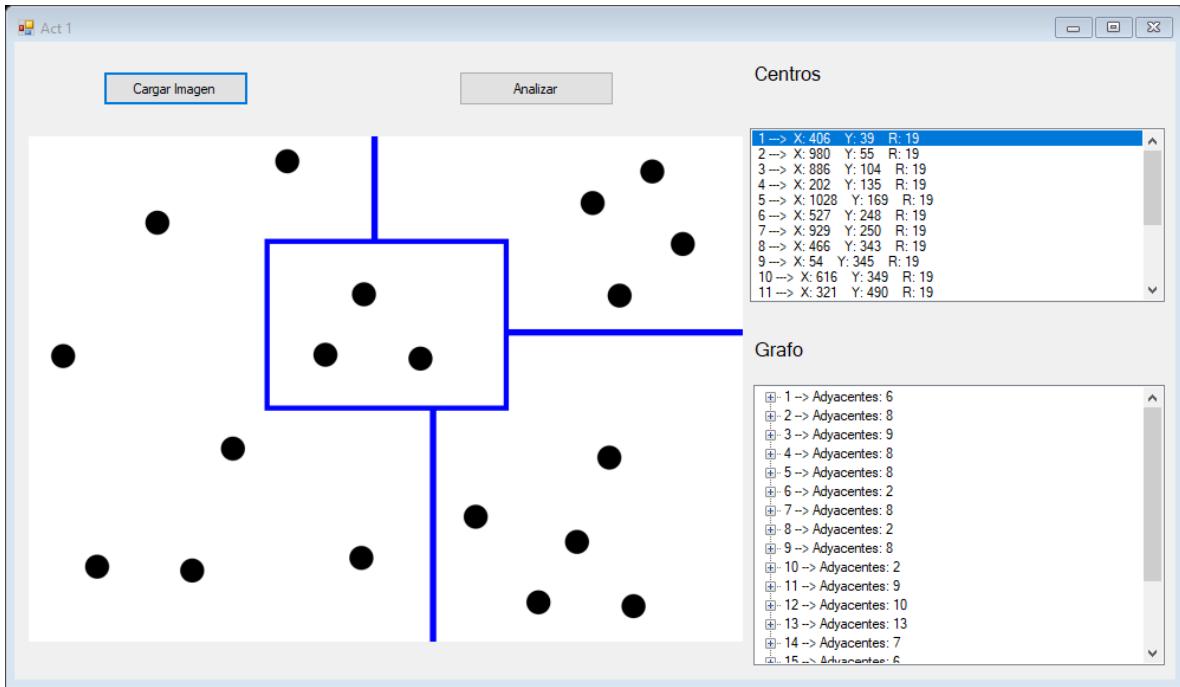


Ilustración 11

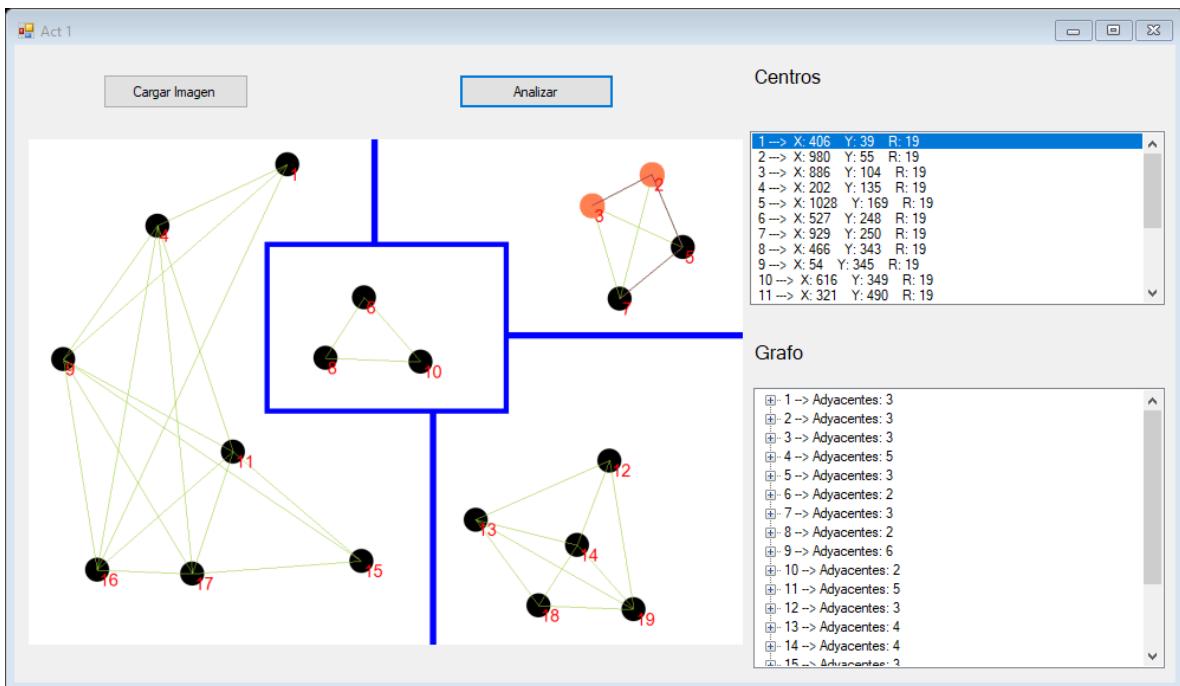


Ilustración 12

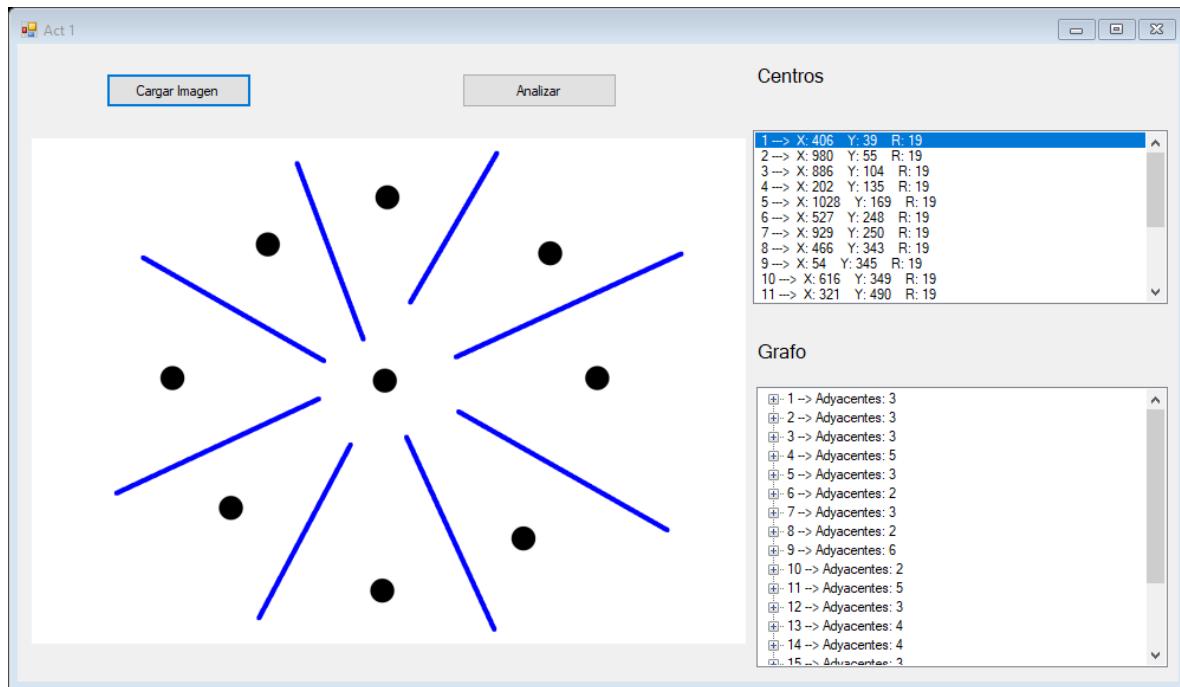


Ilustración 13

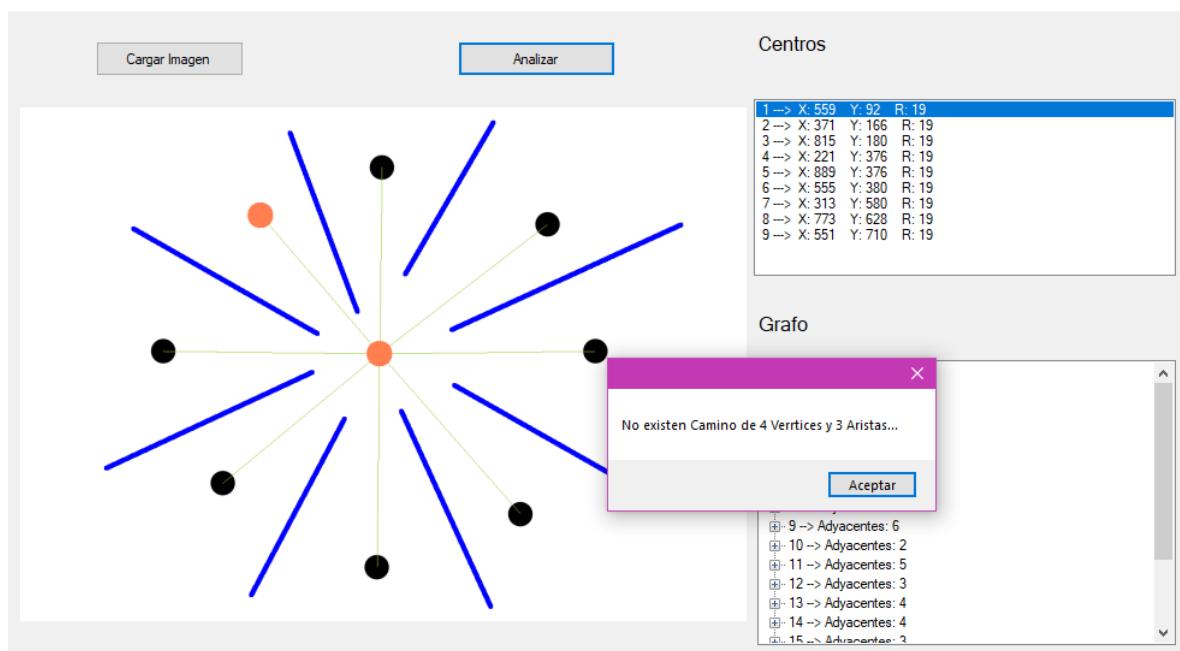


Ilustración 14

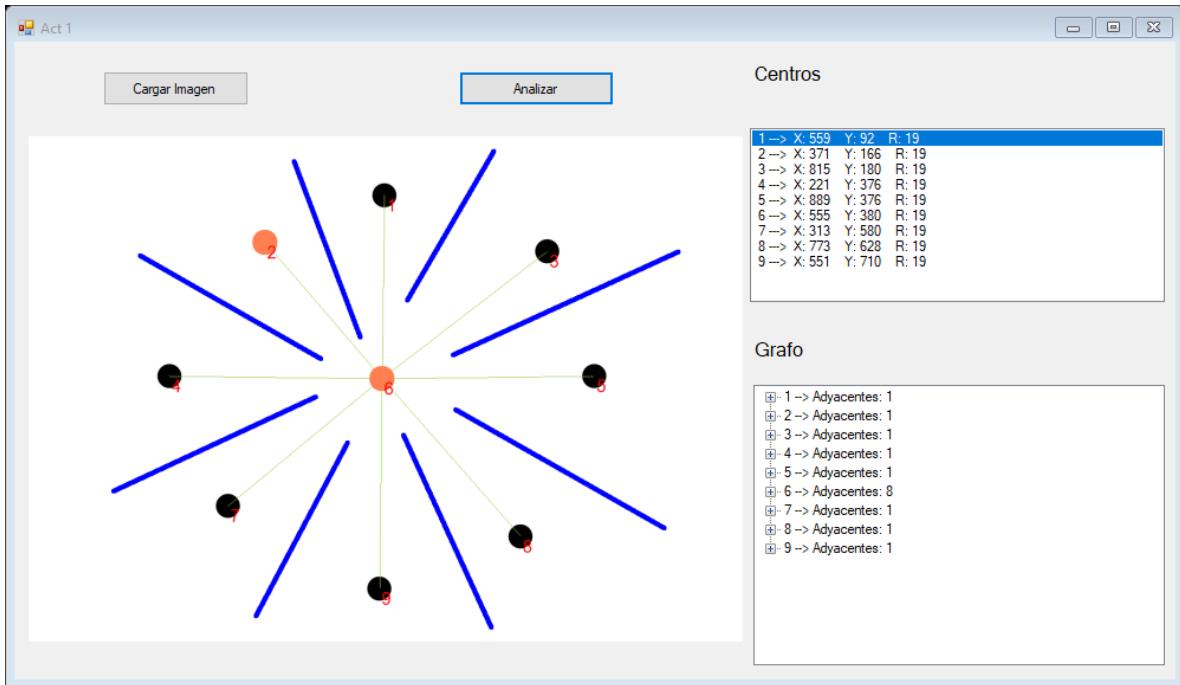


Ilustración 15

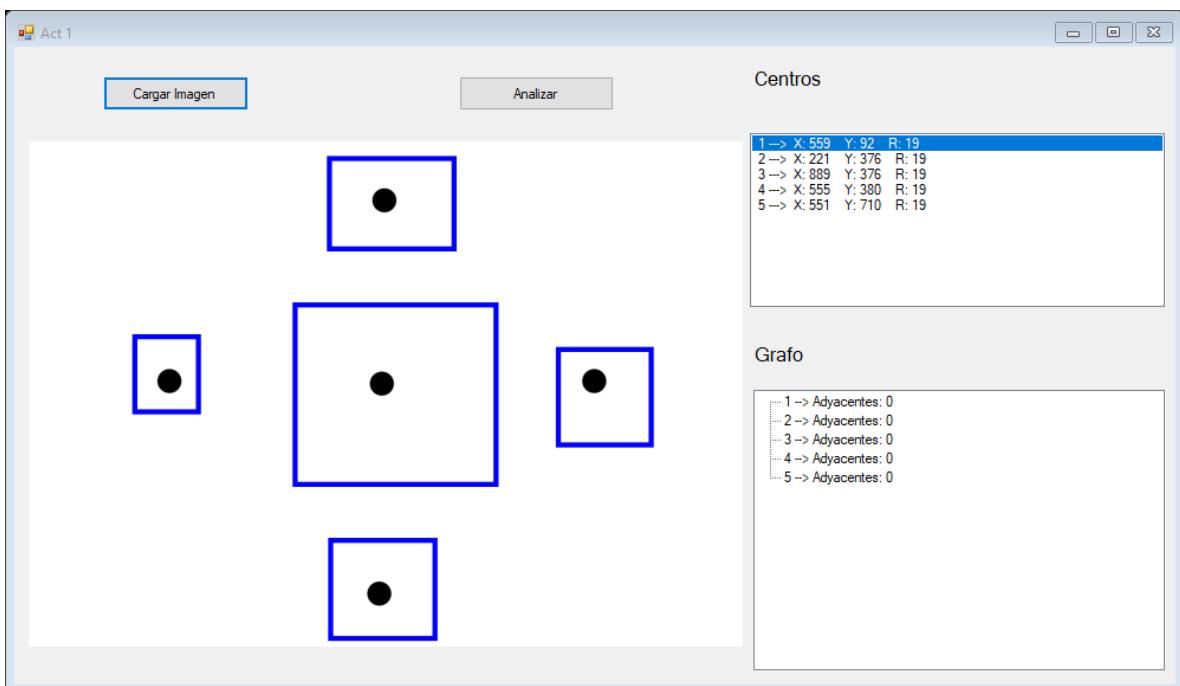


Ilustración 16

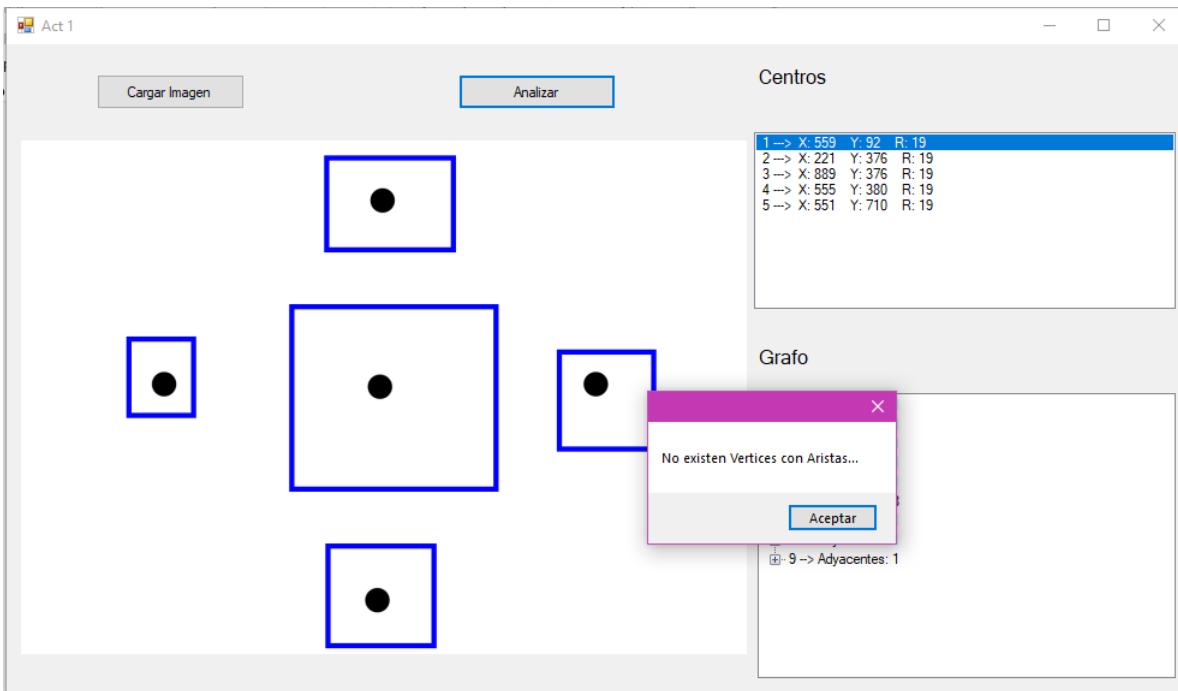


Ilustración 17

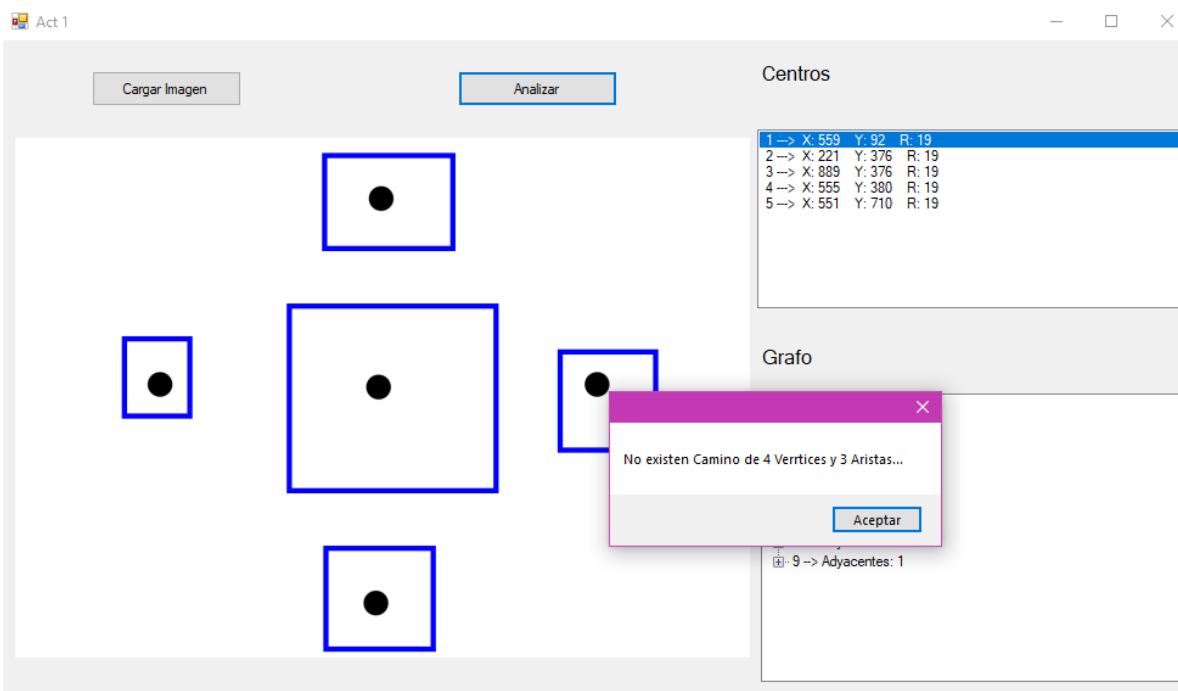


Ilustración 18

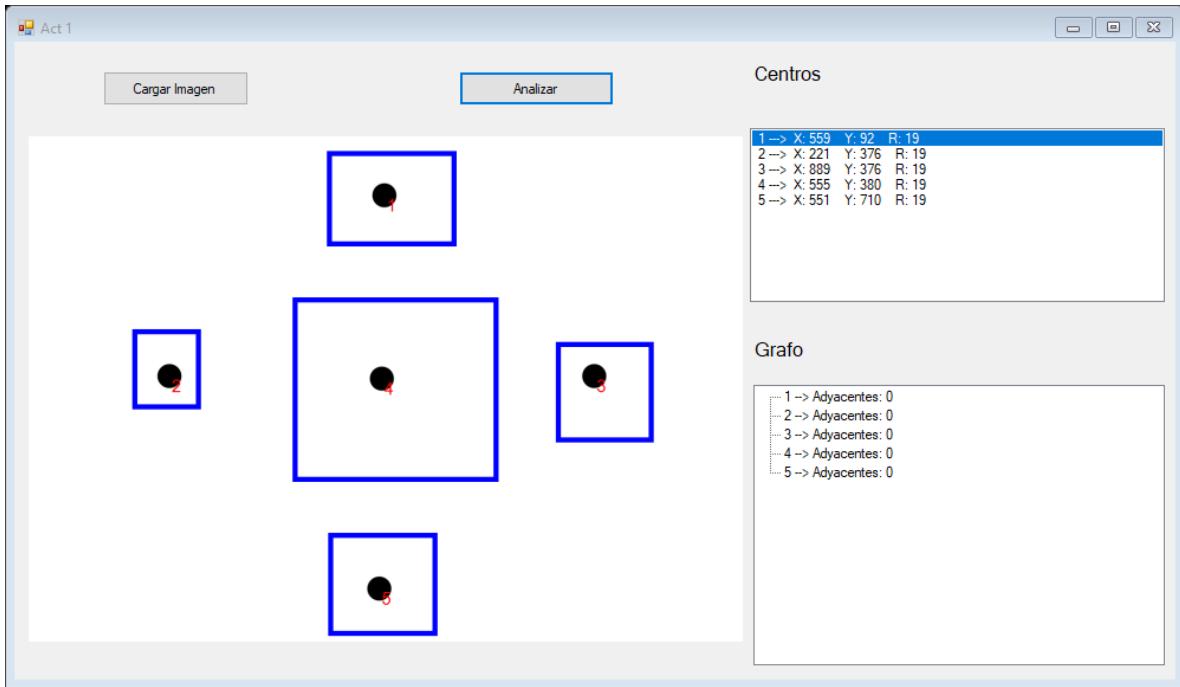


Ilustración 18

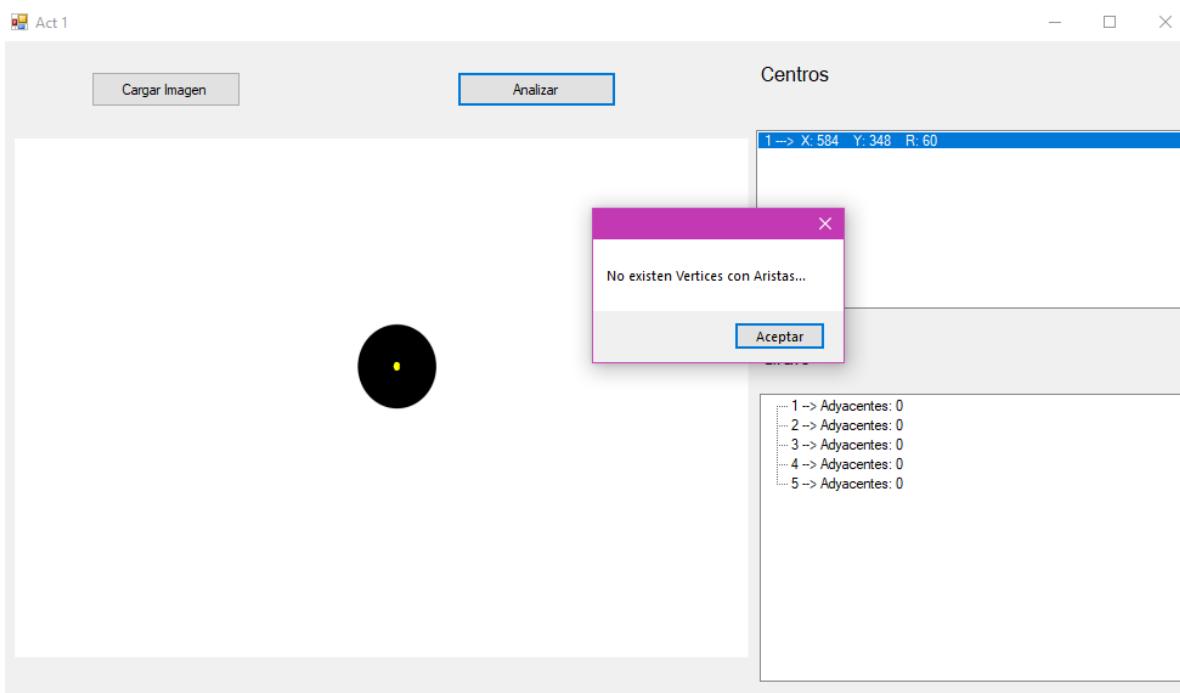


Ilustración 20

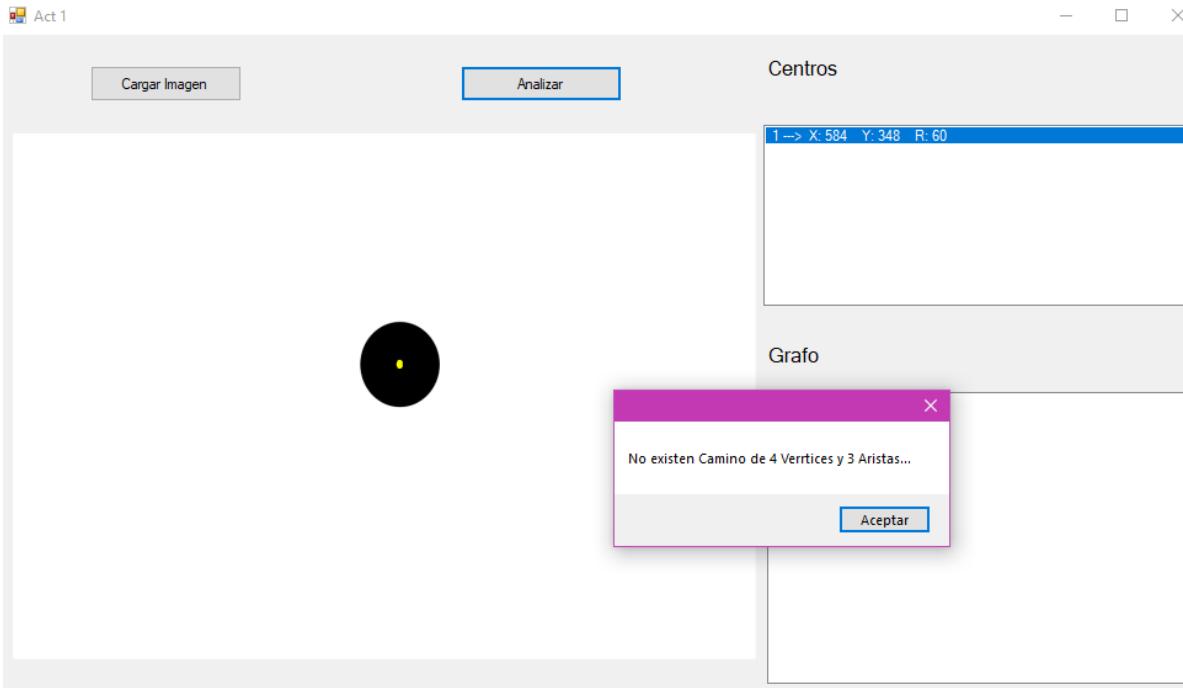


Ilustración 21

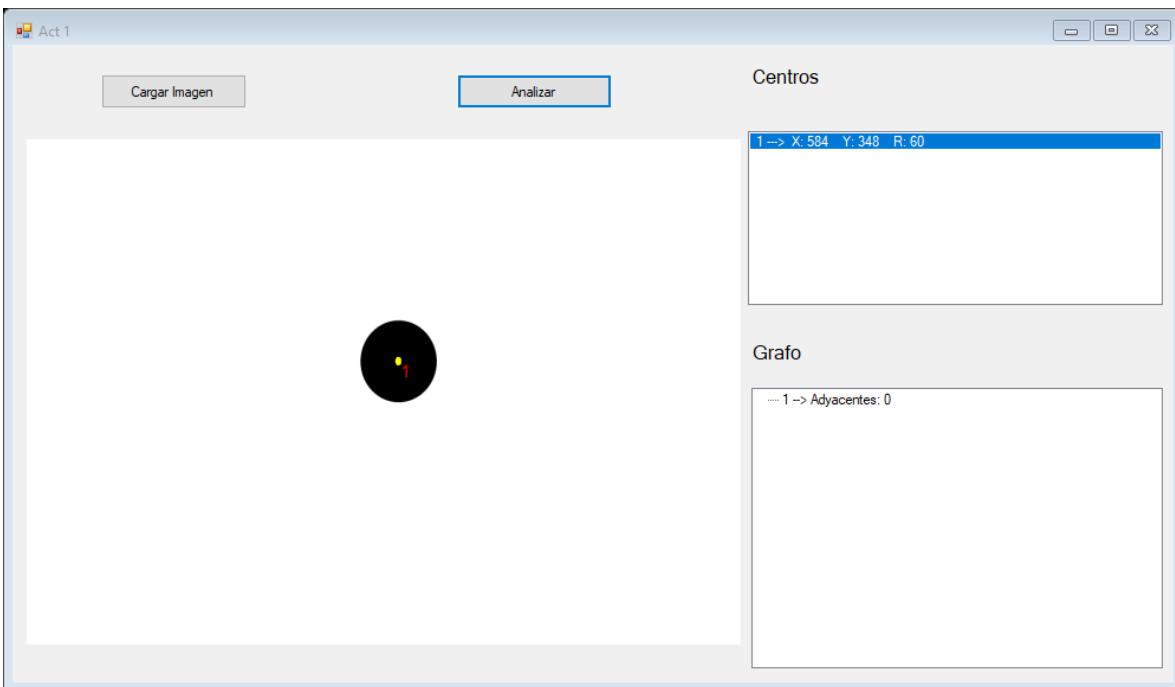


Ilustración 22

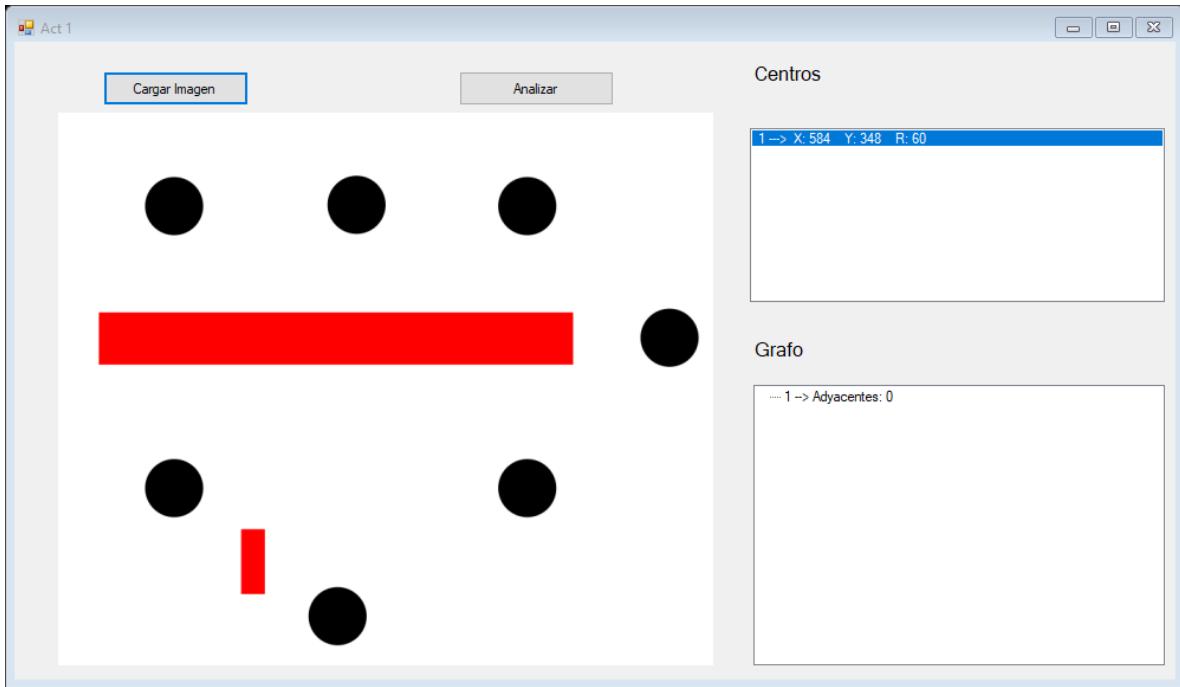


Ilustración 23

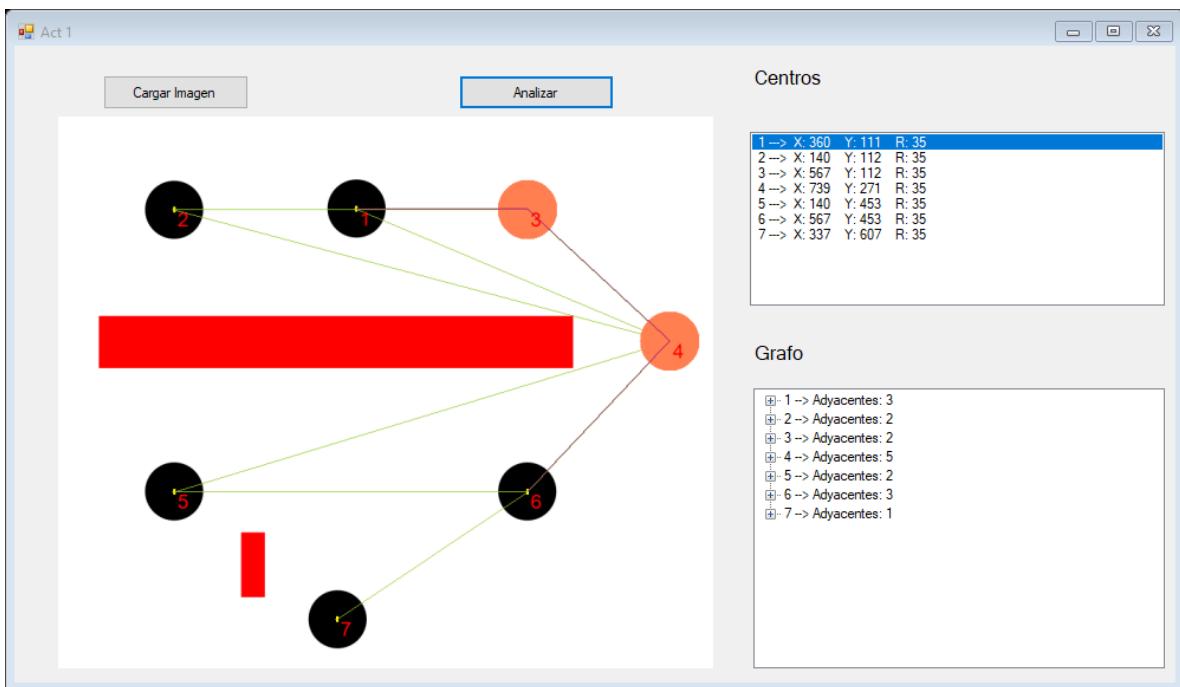


Ilustración 24

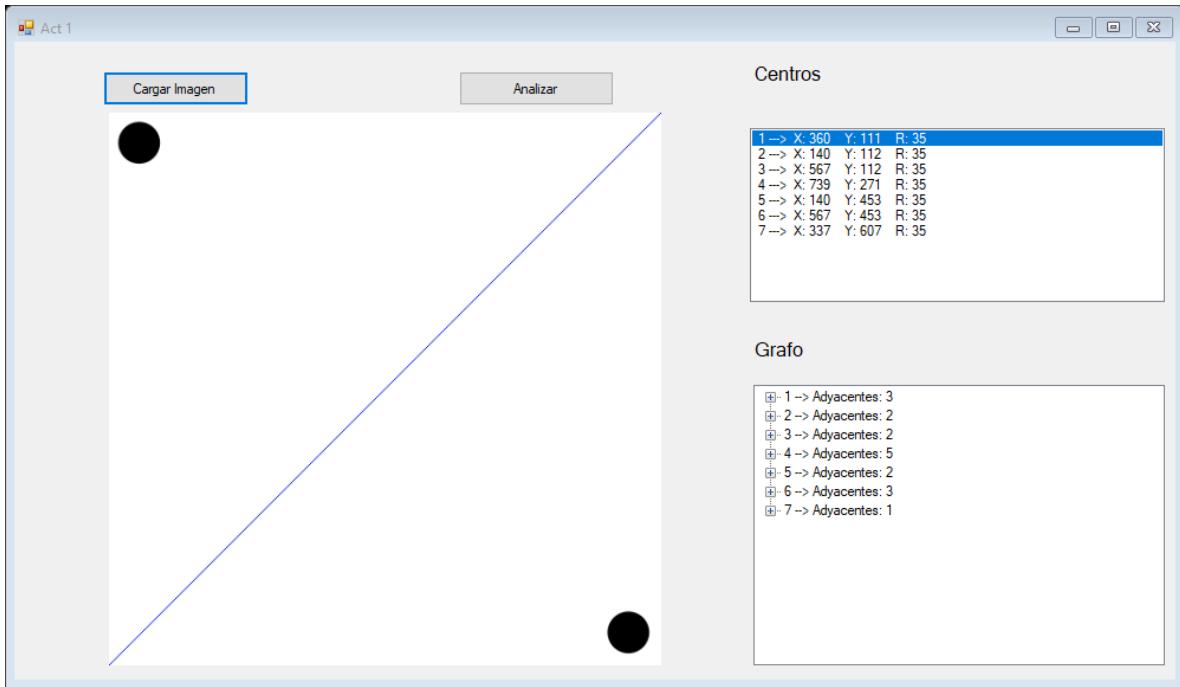


Ilustración 35

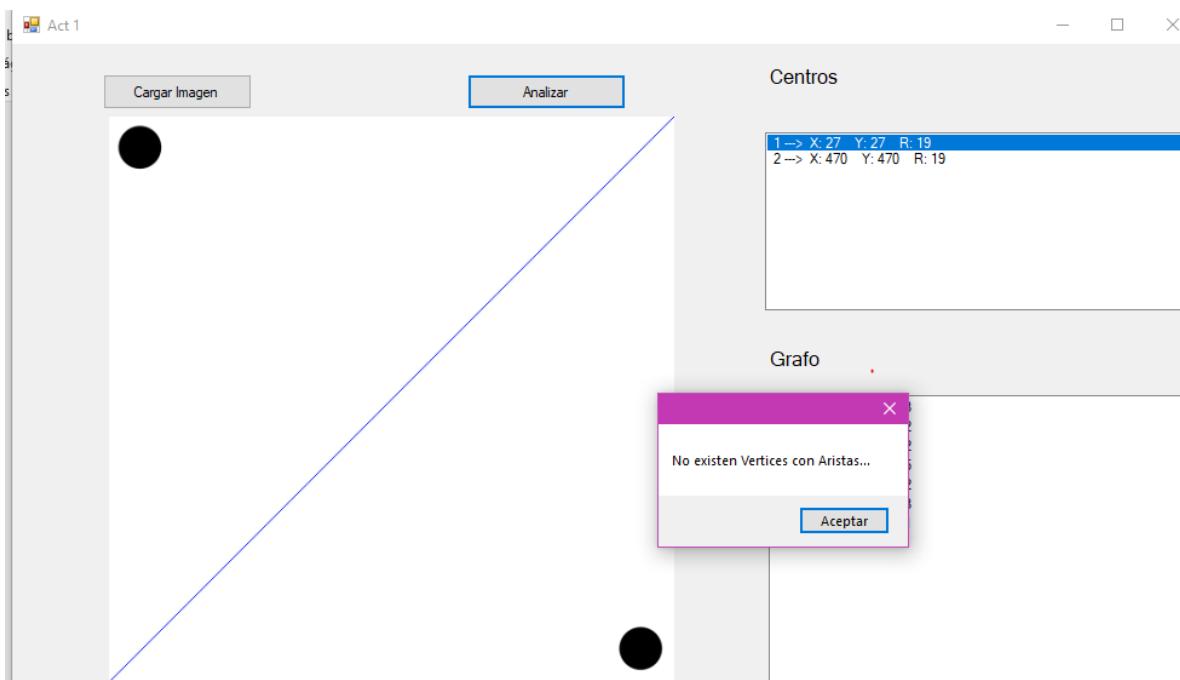


Ilustración 4

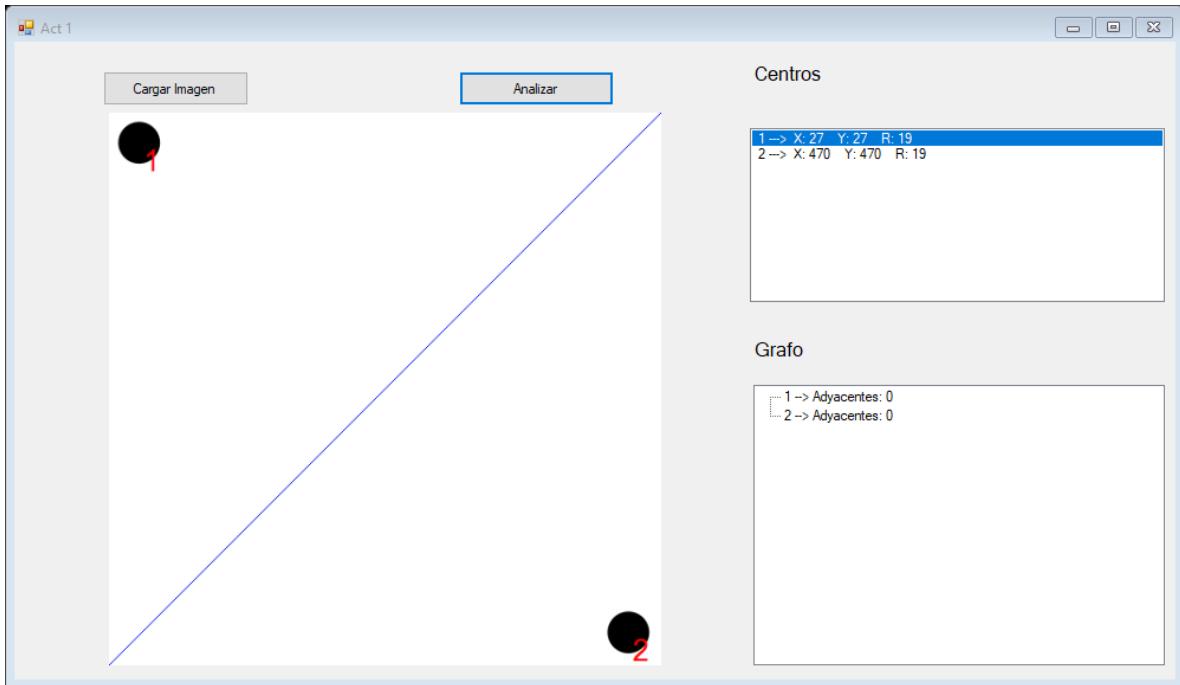


Ilustración 5

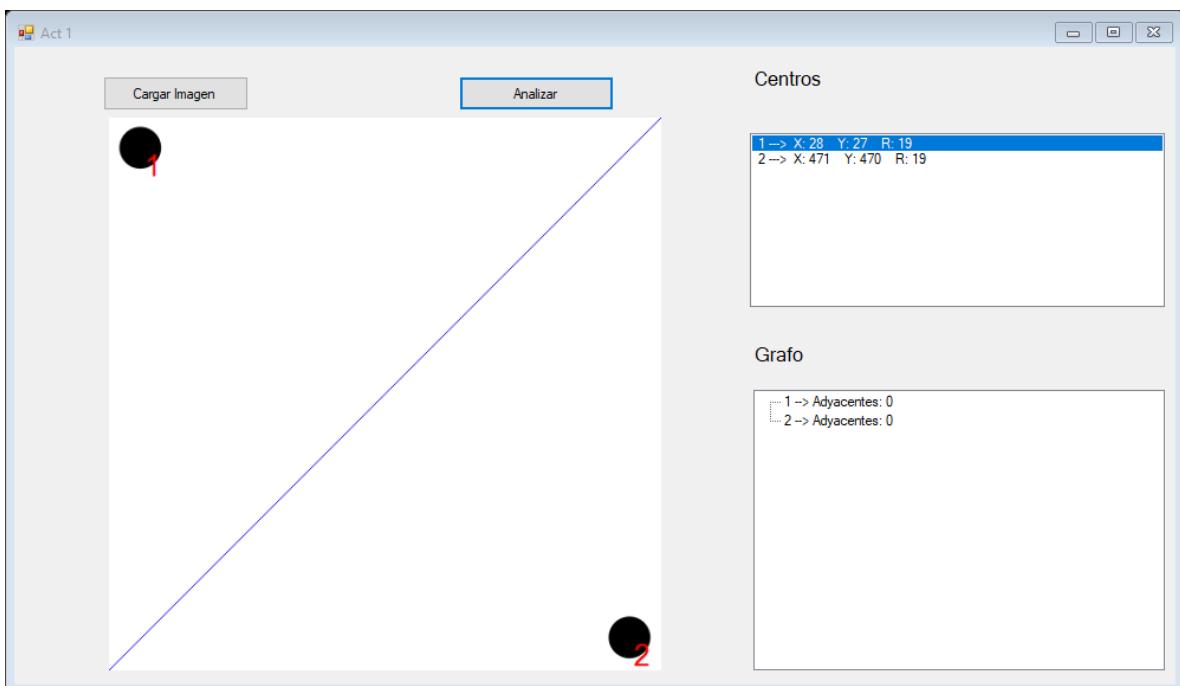


Ilustración 28

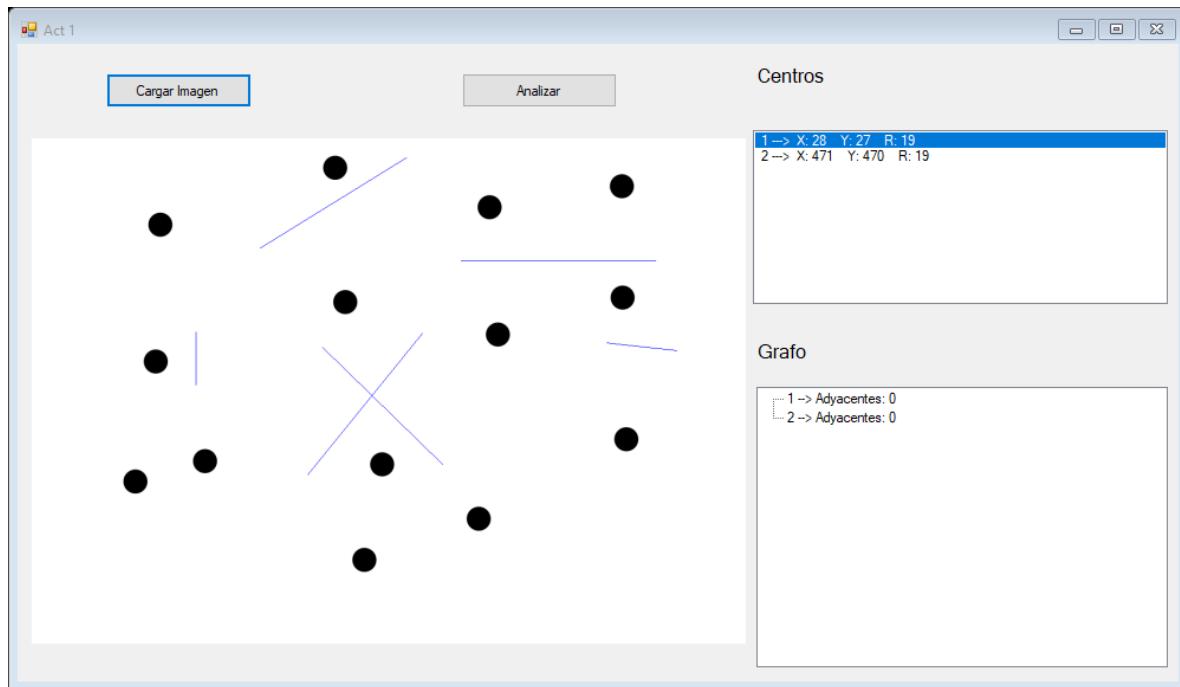


Ilustración 29

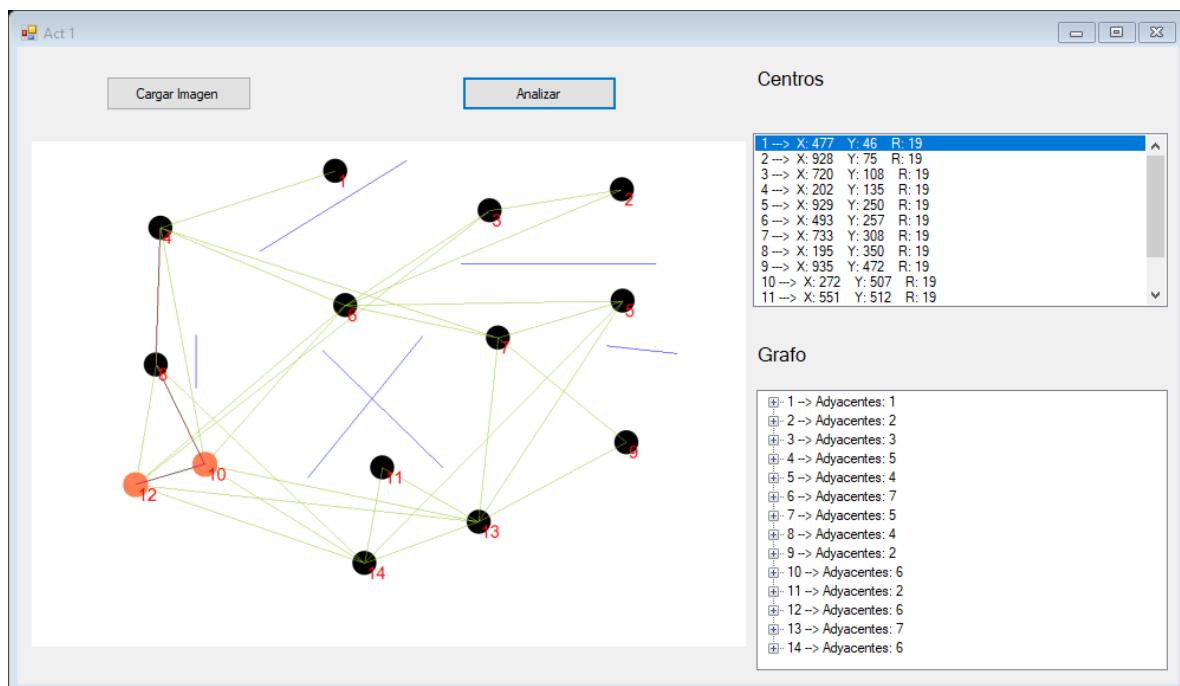


Ilustración 30

Etapa 3:

```
public void TravelDepth(List<int> Recorrido, int Origen)
{
    int Actual;
    Stack <int> Pila = new Stack<int>();
    List <int> Visitados = new List<int>();

    Pila.Push(Origen);

    while (Pila.Count > 0) {

        Actual = Pila.Peek();
        Pila.Pop();

        if (VertexVisited(Visitados, Actual)) {
            Recorrido.Add(Actual);
            Visitados.Add(Actual);

            foreach (Arista Sig in ListaVertices[Actual].GetListaAristas()) {
                if (VertexVisited(Visitados, Sig.GetDestino().GetOrigen().GetID())) {
                    Pila.Push(Sig.GetDestino().GetOrigen().GetID());
                }
            }
        }
    }
}
```

Ilustración 1

```
public void TravelWidth(List<int> Recorrido, int Origen)
{
    int Actual;
    Queue <int> Cola = new Queue<int>();
    List <int> Visitados = new List<int>();

    Cola.Enqueue(Origen);

    while (Cola.Count > 0) {

        Actual = Cola.Peek();
        Cola.Dequeue();

        if (VertexVisited(Visitados, Actual)) {
            Recorrido.Add(Actual);
            Visitados.Add(Actual);

            foreach (Arista Sig in ListaVertices[Actual].GetListaAristas()) {

                if (VertexVisited(Visitados, Sig.GetDestino().GetOrigen().GetID())) {
                    Cola.Enqueue(Sig.GetDestino().GetOrigen().GetID());
                }
            }
        }
    }
}
```

Ilustración 2

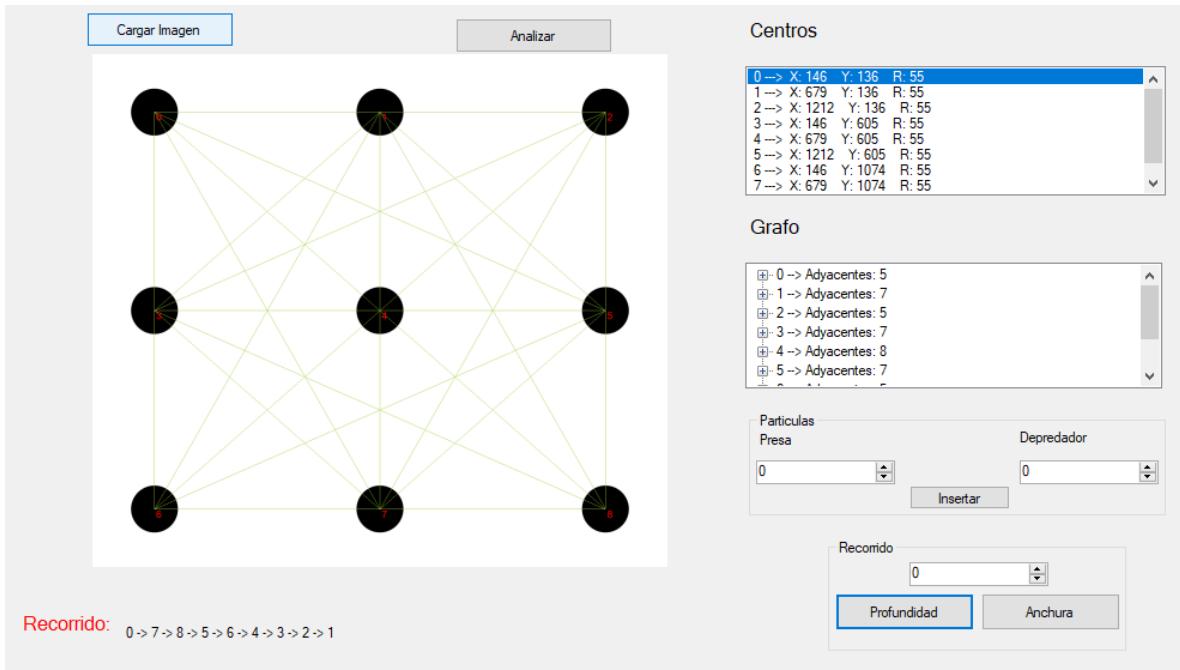


Ilustración 3

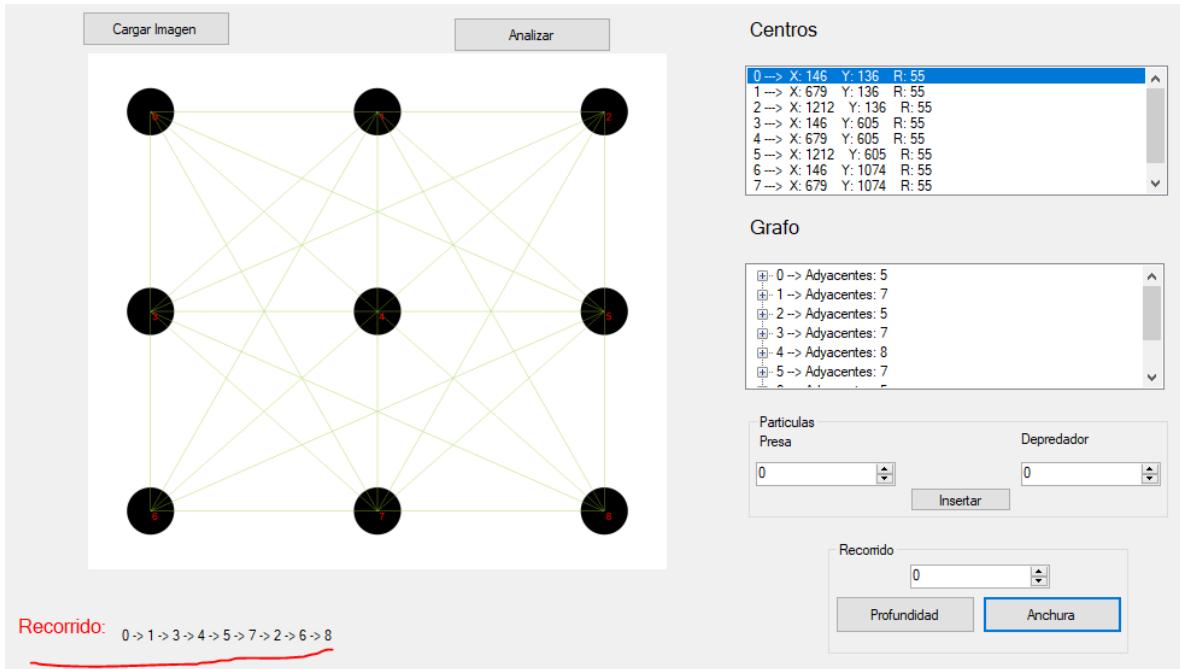


Ilustración 4

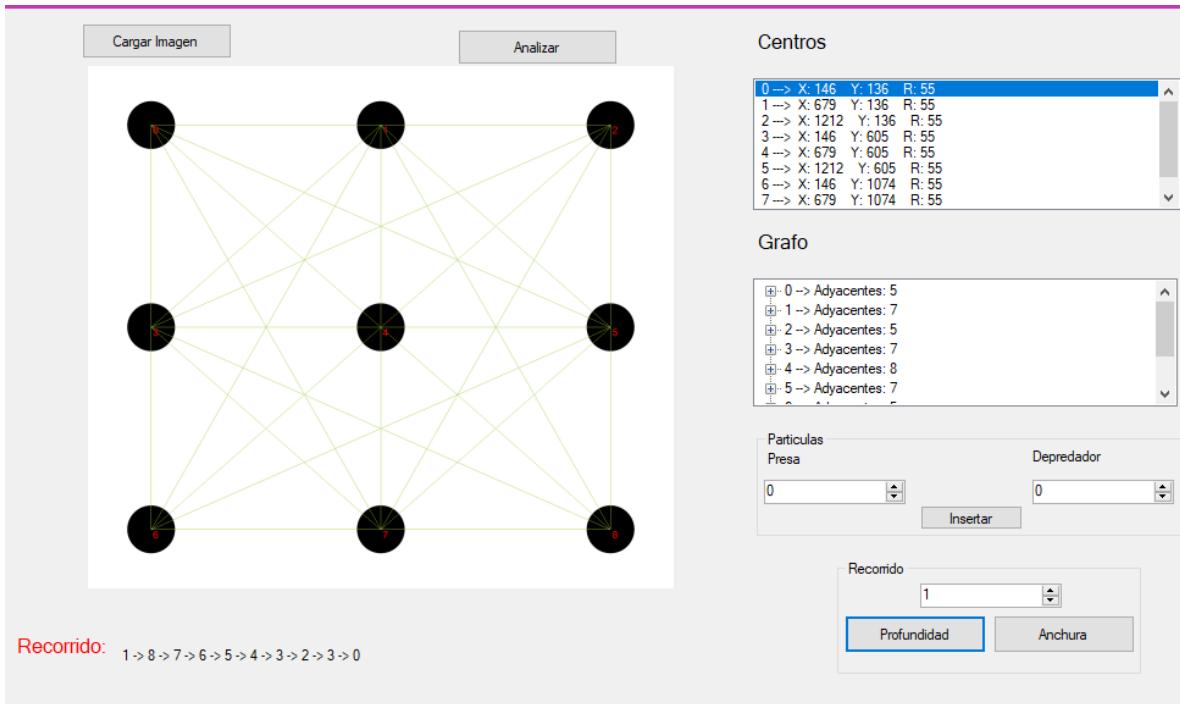


Ilustración 5

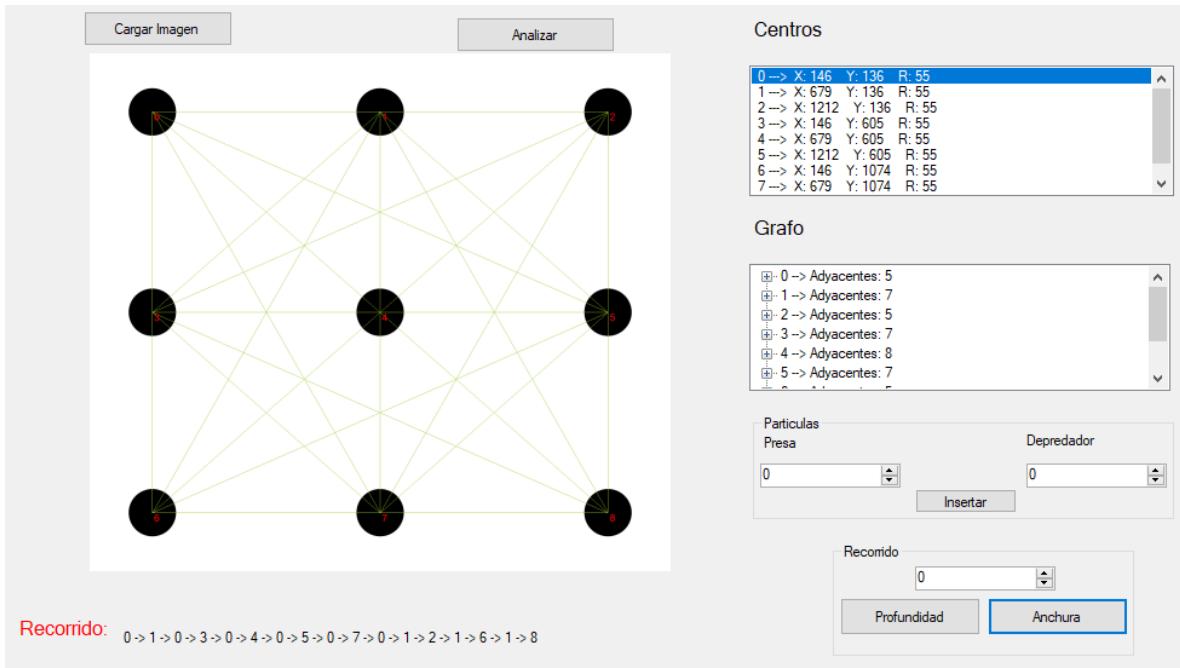


Ilustración 6

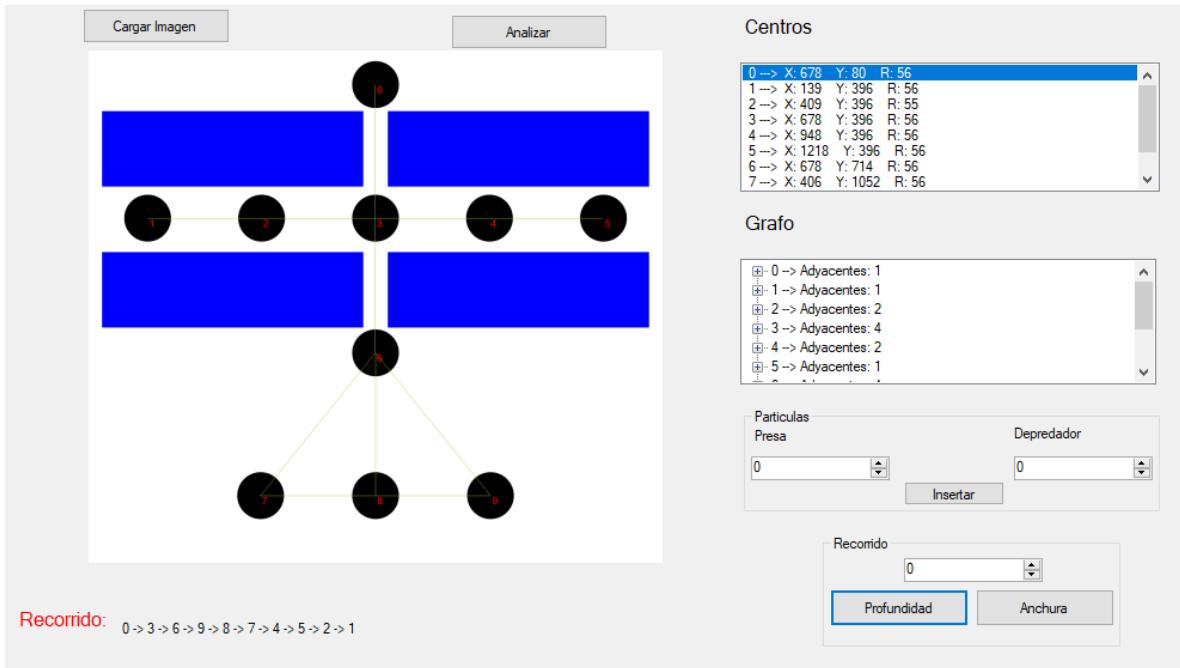


Ilustración 7

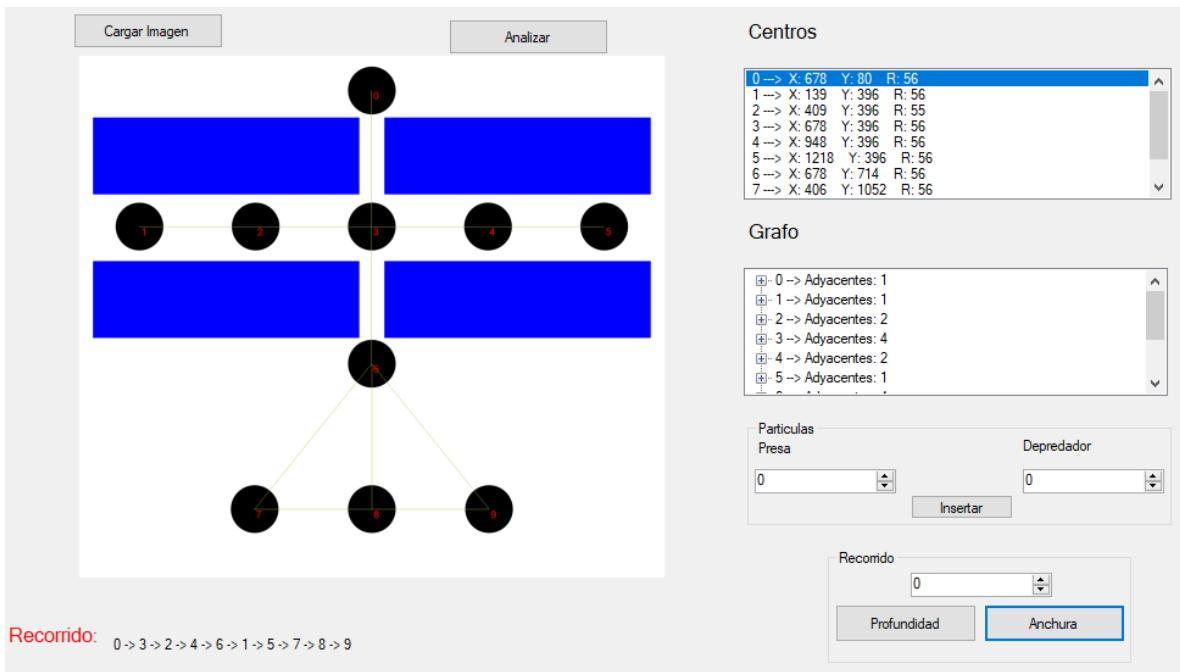


Ilustración 8

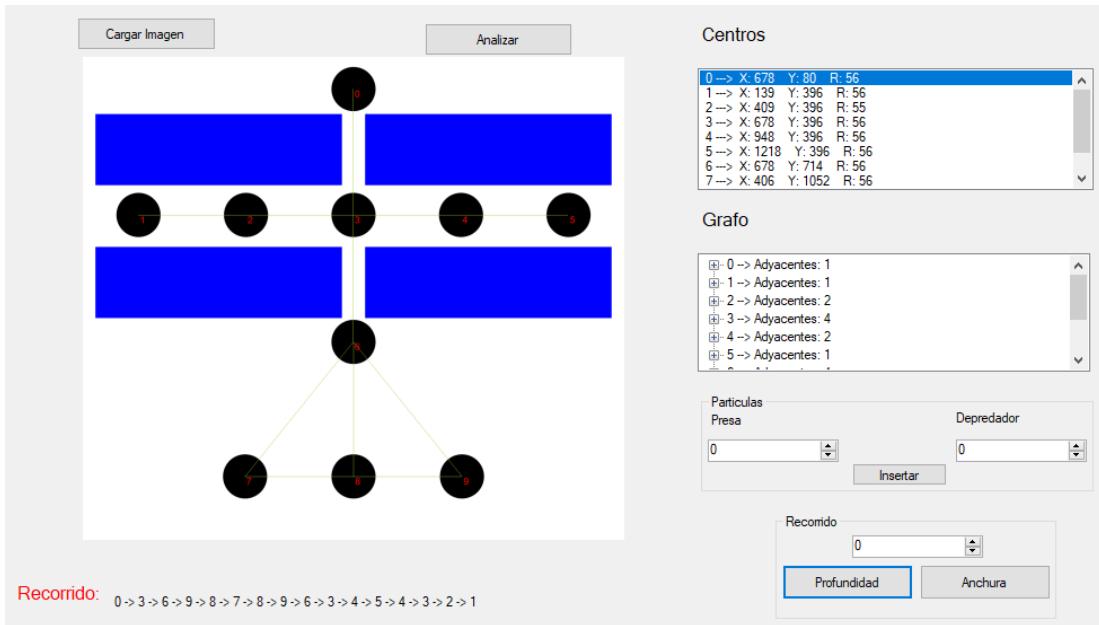


Ilustración 9

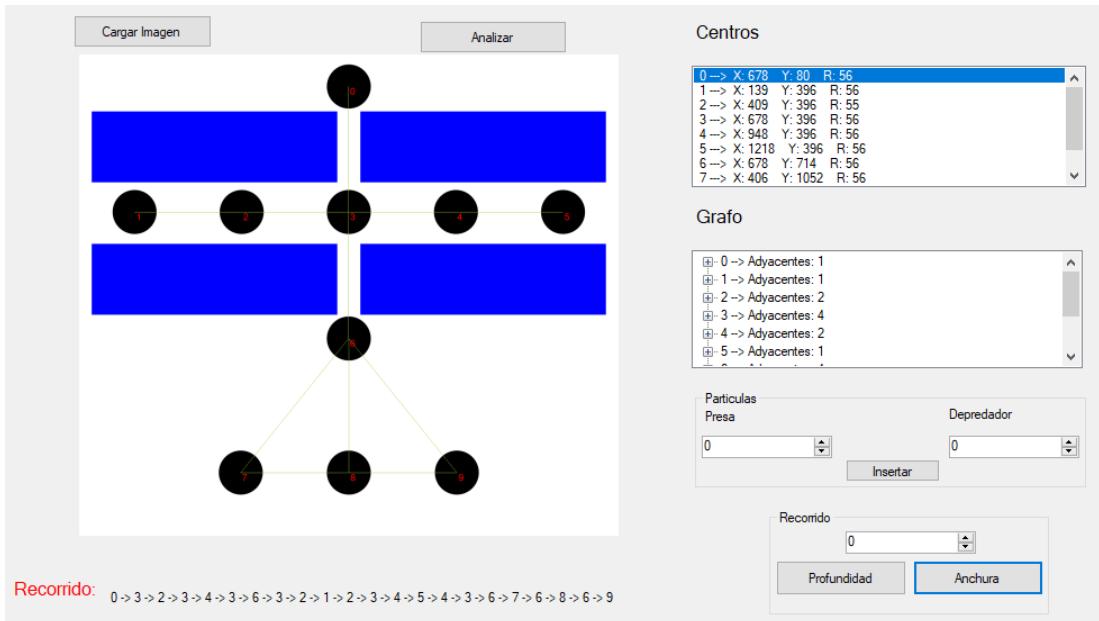


Ilustración 10

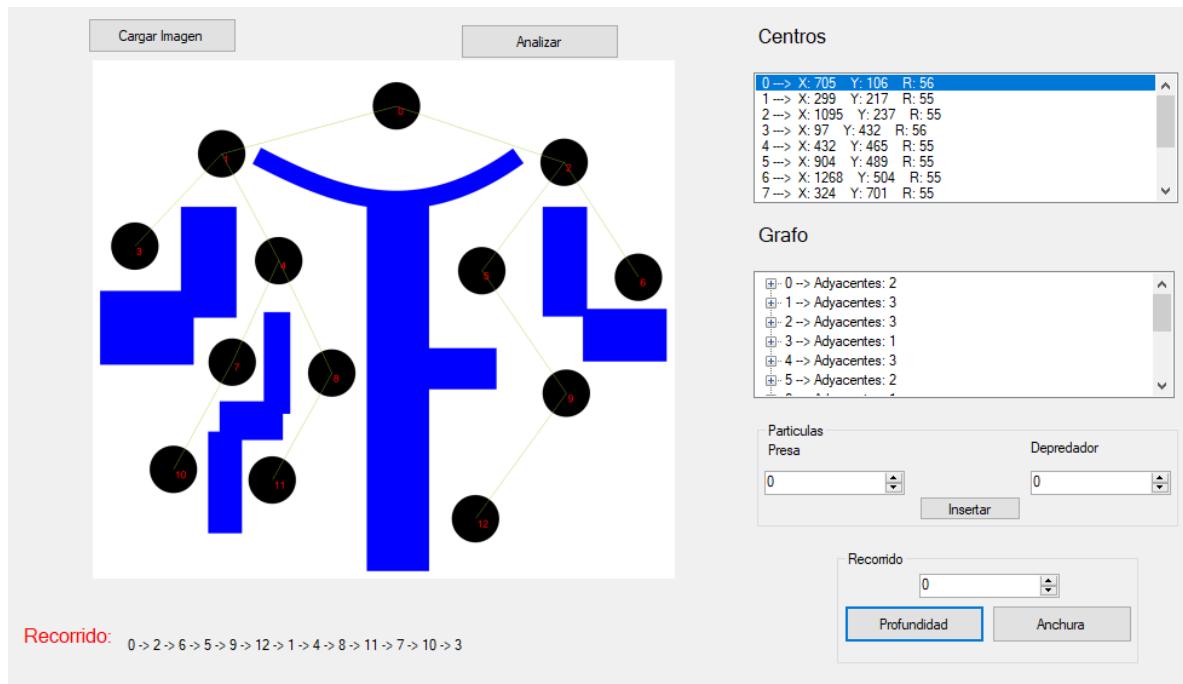


Ilustración 11

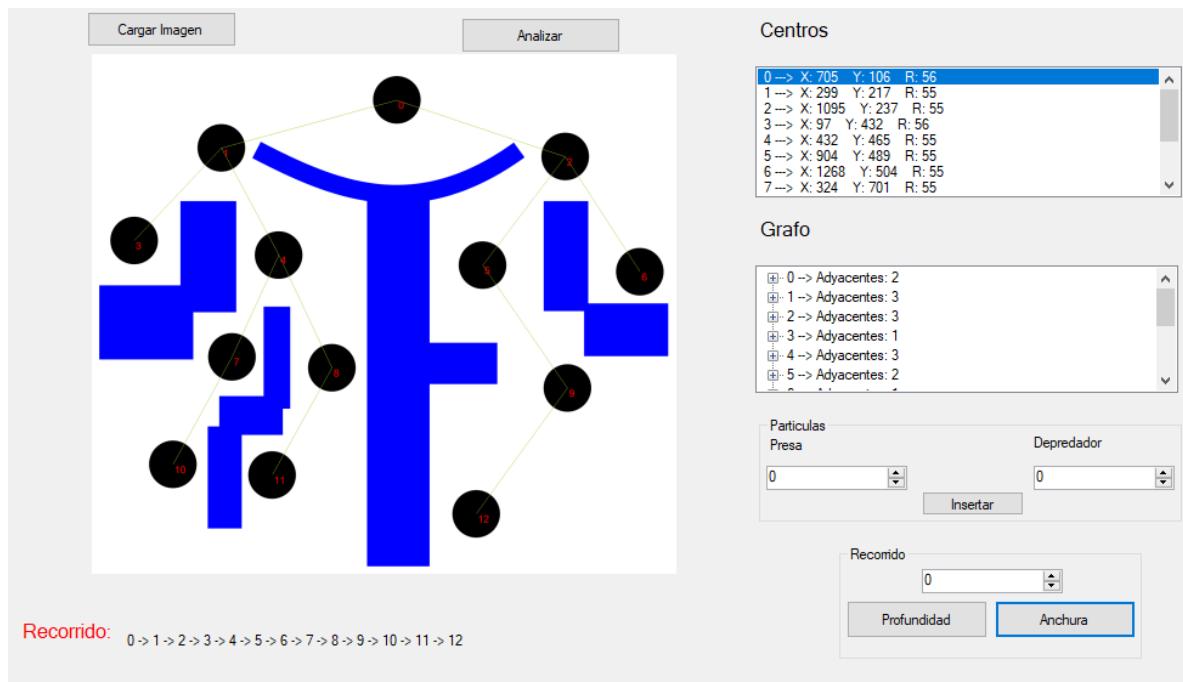


Ilustración 12

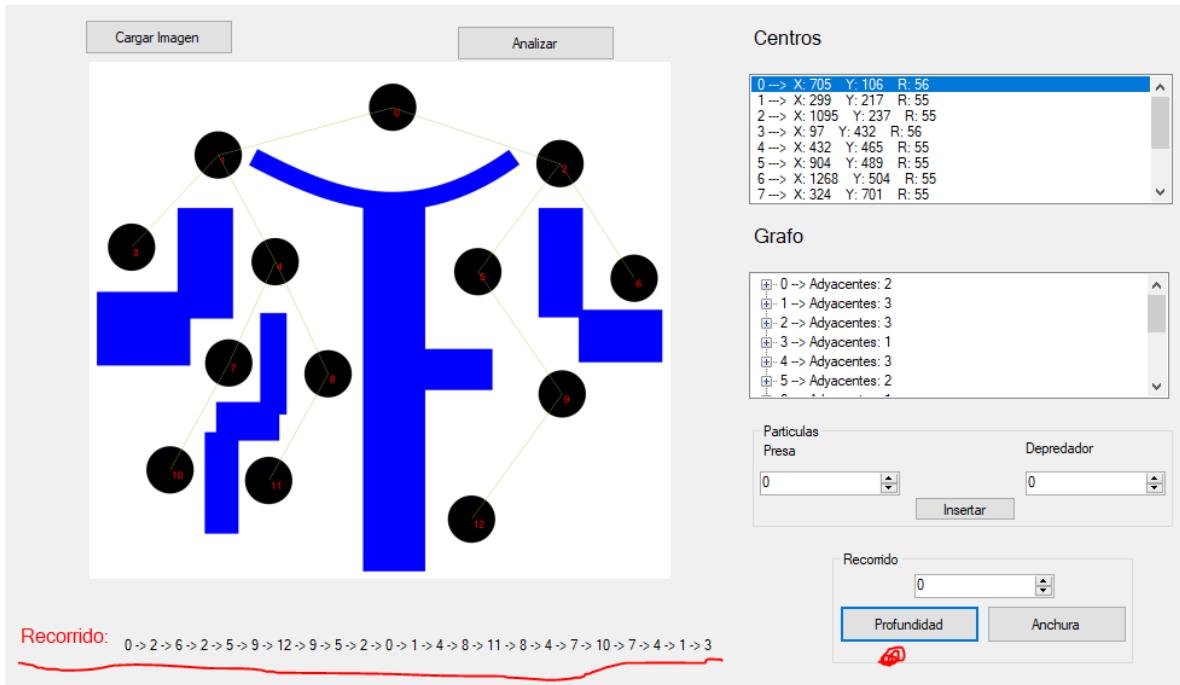


Ilustración 13

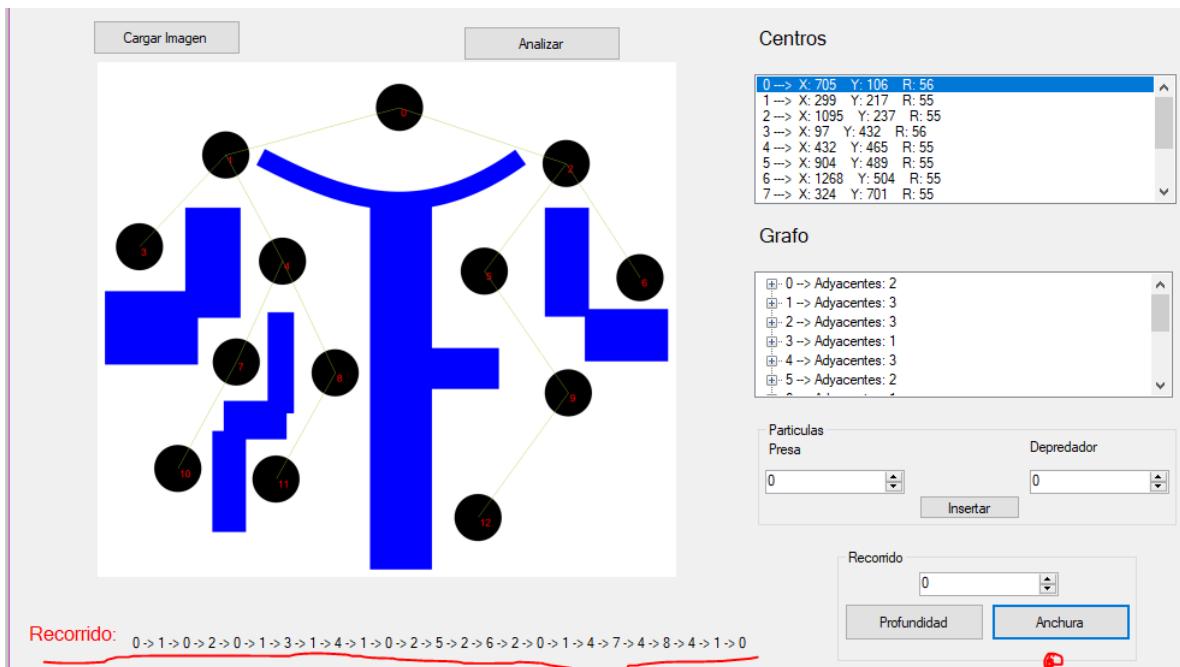


Ilustración 14

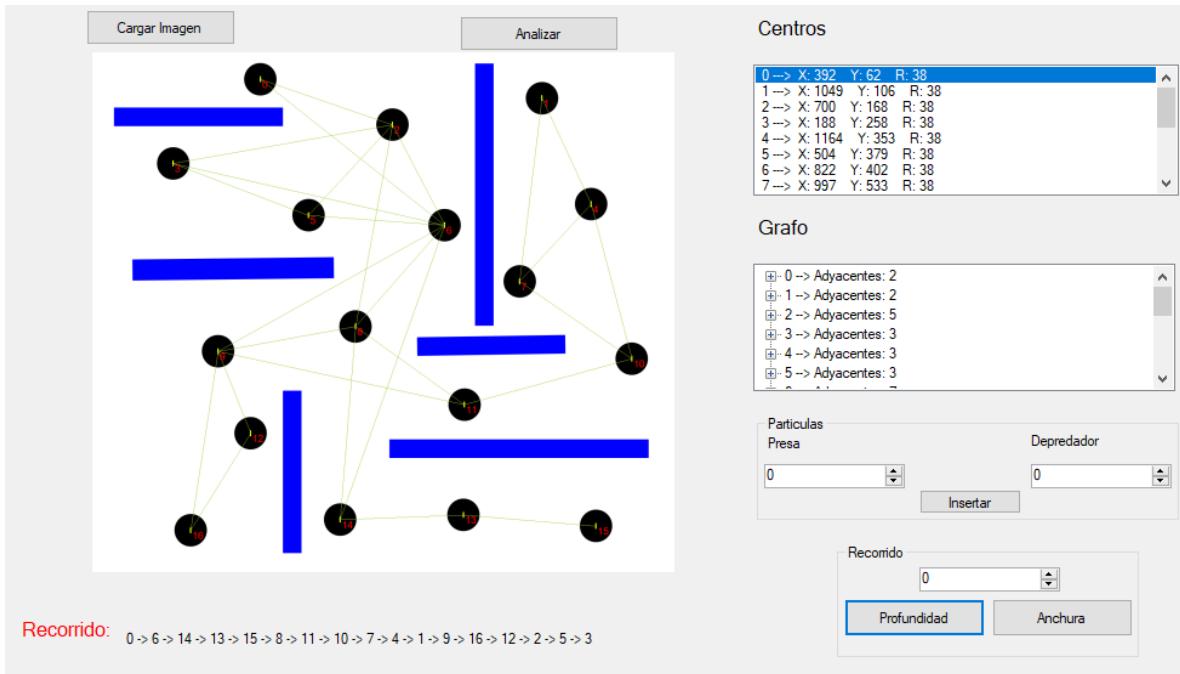


Ilustración 15

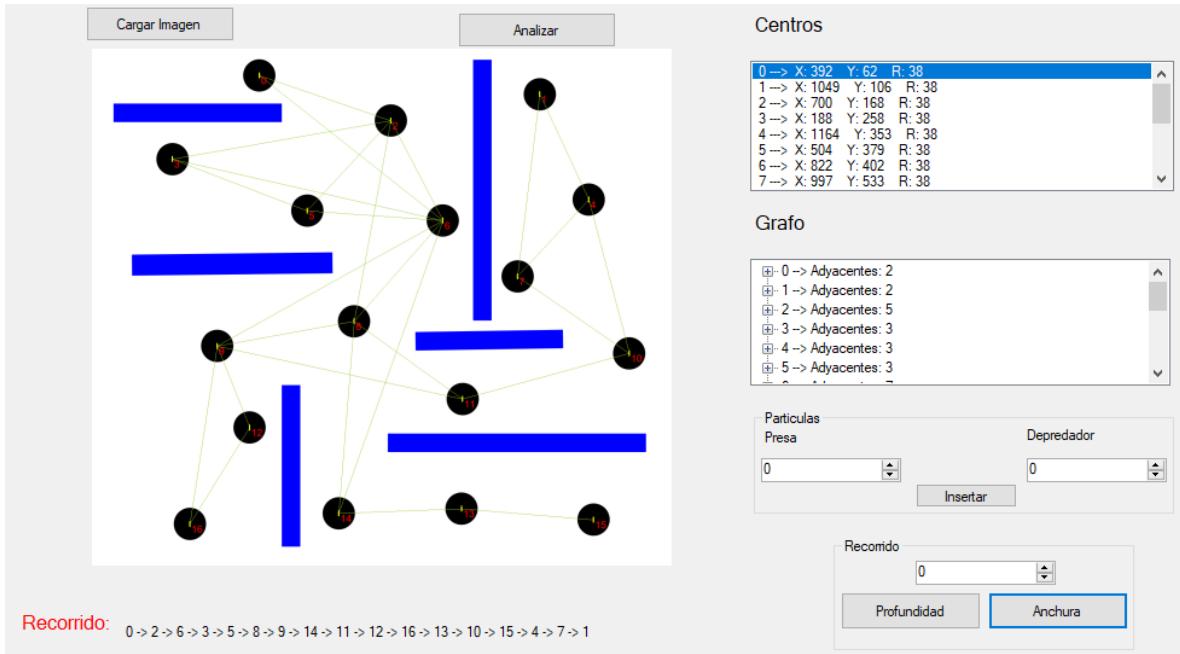


Ilustración 15

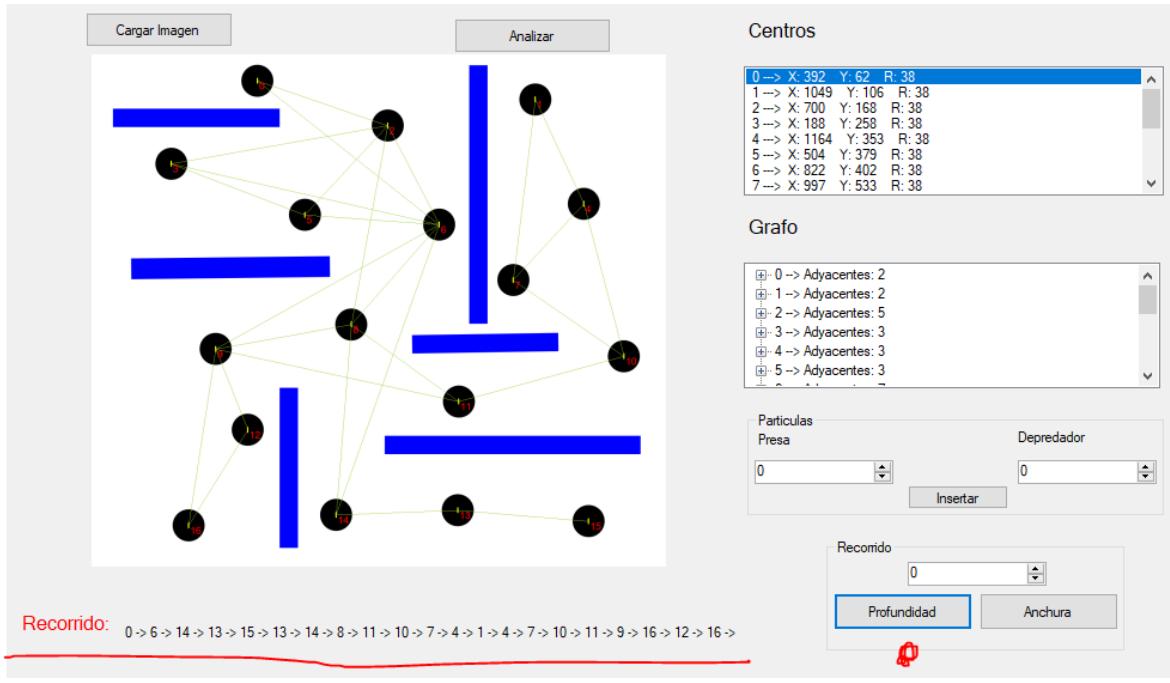


Ilustración 16

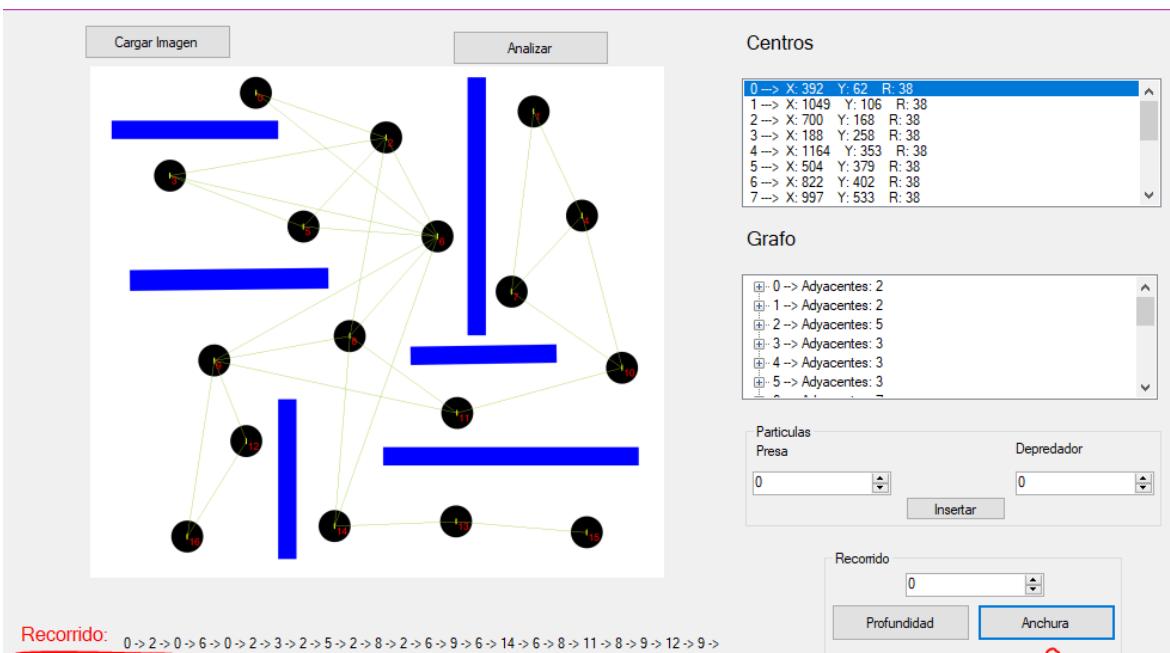


Ilustración 17

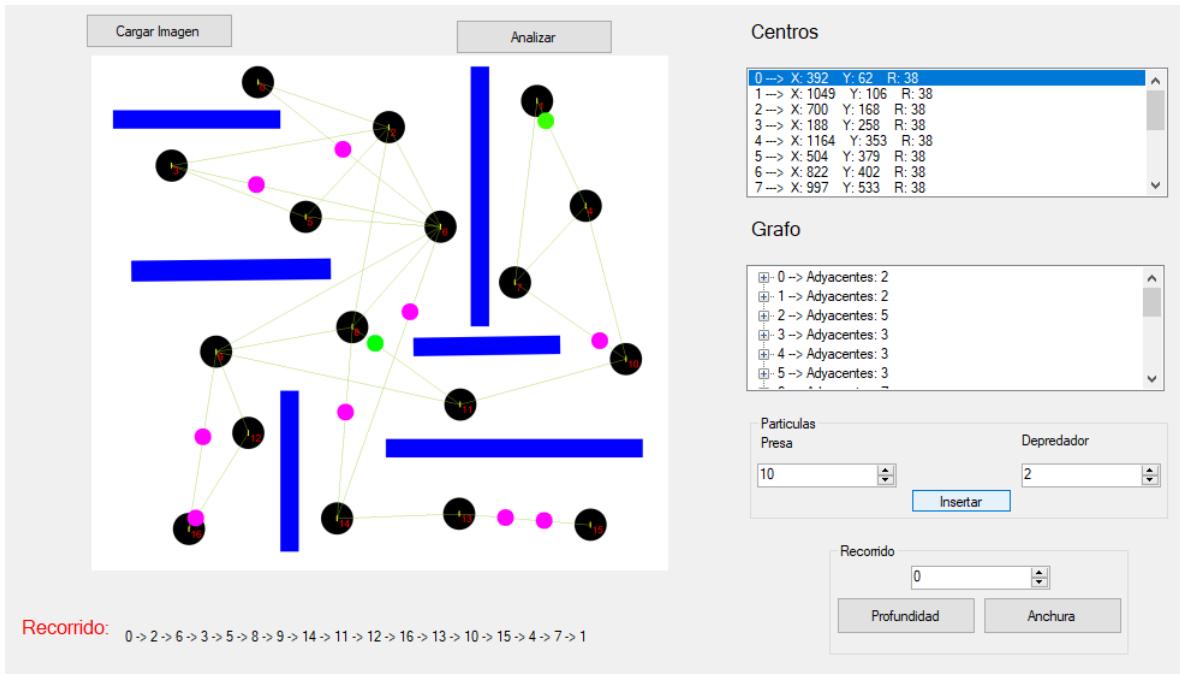


Ilustración 18

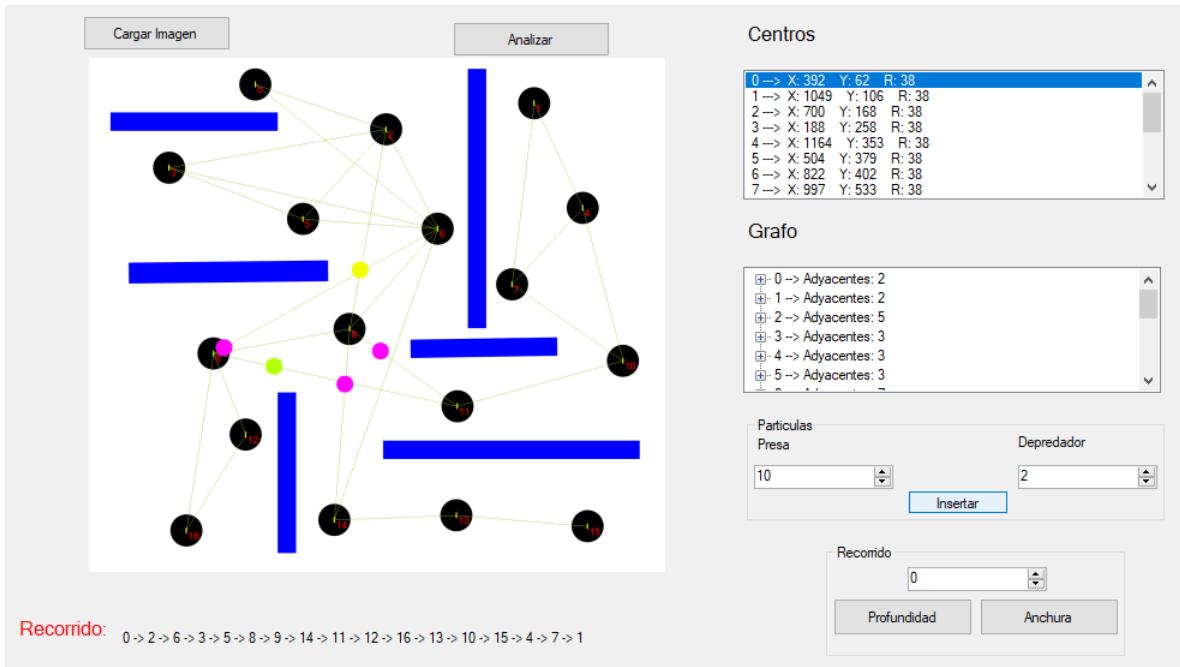


Ilustración 19

Etapa 4:

```
public void Kruskal(List<Arista> KRUSKAL){
    Arista Actual = new Arista();
    int indOrg, indDes; // e_1, e_2

    List<Arista> Candidatos = new List<Arista>(); // C
    CreateListOfCandidates (Candidatos); // Inicializar lista de candidatos
    SortAscToDec(Candidatos); //Sort Candidates

    List<List<int>> Arbol = new List<List<int>>(); // CC
    InitializeTree(Arbol); // inicializa n componentes conexos, cada

    while(Candidatos.Count > 0){ // Mientras tenga aristas

        Actual = Candidatos[0]; // La arista de menor peso
        Candidatos.RemoveRange(0,1);
        indOrg = PosicionOfVertex(Actual.GetOrigen().GetOrigen().GetID(), Arbol);
        indDes = PosicionOfVertex(Actual.GetDestino().GetOrigen().GetID(), Arbol);
        if(indOrg != indDes){
            Arbol[indOrg].AddRange(Arbol[indDes]);
            Arbol.RemoveRange(indDes,1);
            KRUSKAL.Add(Actual);
        }
    }
} // agraegar a diagrama
```

Ilustración 6

```
void CreateListOfCandidates(List<Arista> Candidatos){
    foreach(Vertice i in ListaVertices){
        foreach(Arista j in i.GetListaAristas()){
            if(ThereIsAnEdge(Candidatos,j)){
                Candidatos.Add(j);
            }
        }
    }
}
```

Ilustración 7

```
void SortAscToDec(List<Arista> Candidatos){
    Arista Aux = new Arista();
    for(int i=0; i<Candidatos.Count; i++){
        for(int j=i; j<Candidatos.Count; j++){
            if(Candidatos[i].GetDistancia() > Candidatos[j].GetDistancia() ){
                Aux = Candidatos[j];
                Candidatos[j] = Candidatos[i];
                Candidatos[i] = Aux;
            }
        }
    }
}
```

Ilustración 8

```

void InitializeTree(List<List<int>> CC){
    List<int> Aux;
    for(int i=0; i<listaVertices.Count; i++){
        Aux = new List<int>();
        Aux.Add(i);
        CC.Add(Aux);
    }
}

```

Ilustración 9

```

public void Prim(List<Arista> PRIM,int Origen){
    int nVertices = 1, newOrigen;
    List<Arista> Candidatos = new List<Arista>(); // C

    CreateListOfCandidates (Candidatos);           // Inicializar lista de candidatos
    SortAscToDec(Candidatos);                     //Sort Candidates

    List<string> Visitados = new List<string>();
    List<string> Arbol = new List<string>();       // S = Solucion
    Visitados.Add(Origen.ToString());
    Arbol.Add(Origen.ToString());

    while(nVertices < listaVertices.Count){          // Mientras tenga aristas
        Arista Actual = new Arista();
        Actual = Outgoing(Candidatos,Arbol);
        if(Actual == null){
            newOrigen = NewOrigin(Visitados);
            Arbol.Add(newOrigen.ToString());
            Visitados.Add(newOrigen.ToString());
            nVertices++;
        }else{
            PRIM.Add(Actual);
            if(Find(Arbol,Actual.GetOrigen().GetOrigen().GetID())){
                newOrigen = Actual.GetDestino().GetOrigen().GetID();
                Arbol.Add(newOrigen.ToString());
                Visitados.Add(newOrigen.ToString());
            }else{
                newOrigen = Actual.GetOrigen().GetOrigen().GetID();
                Arbol.Add(newOrigen.ToString());
                Visitados.Add(newOrigen.ToString());
            }
            nVertices++;
        }
    }
}

```

Ilustración 10

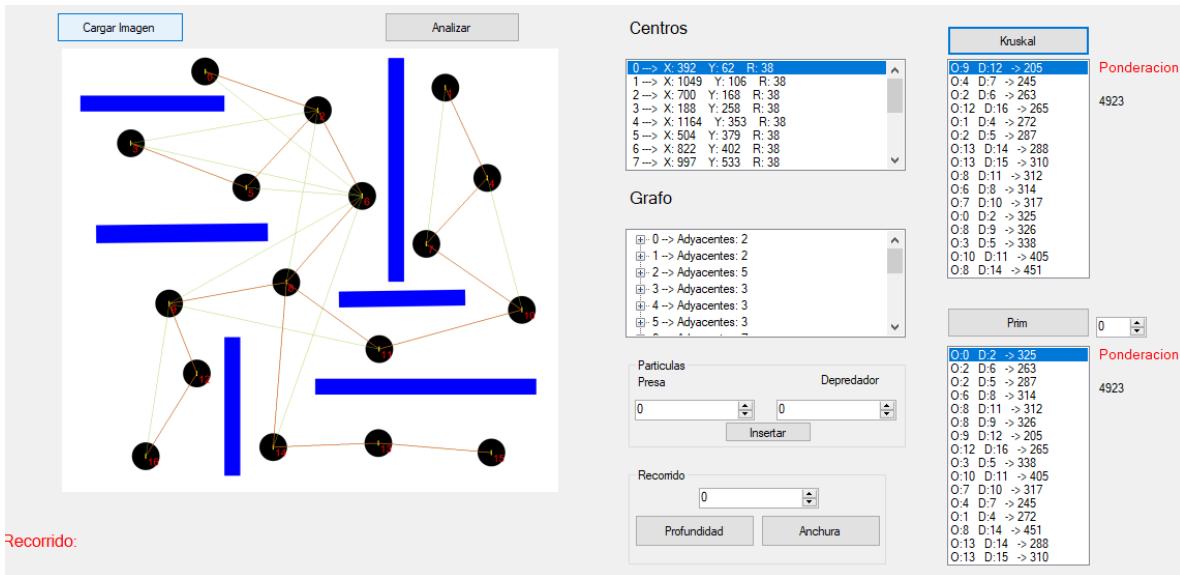


Ilustración 11

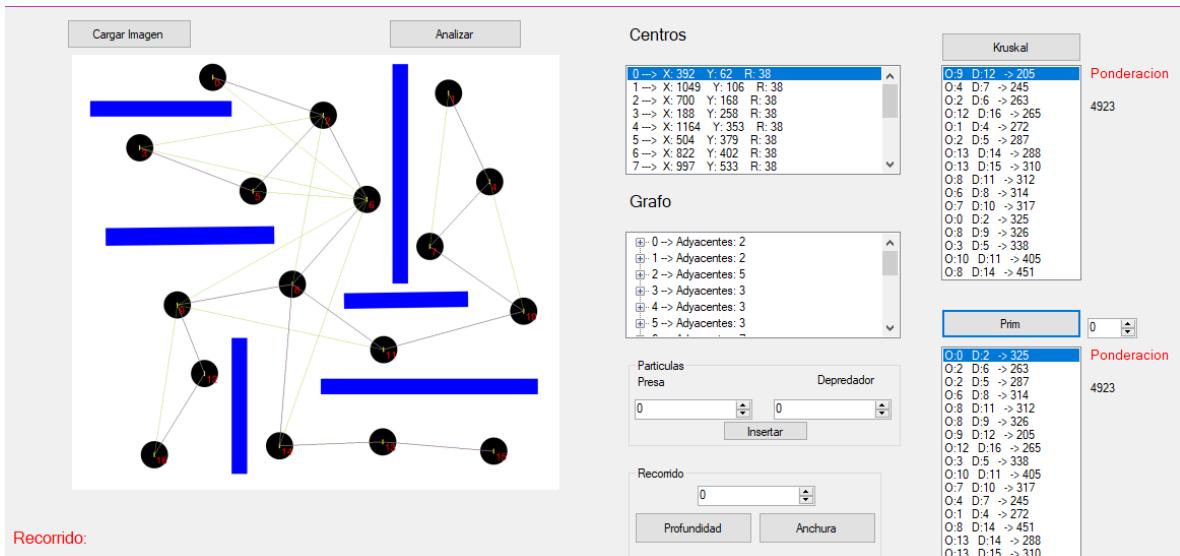


Ilustración 12

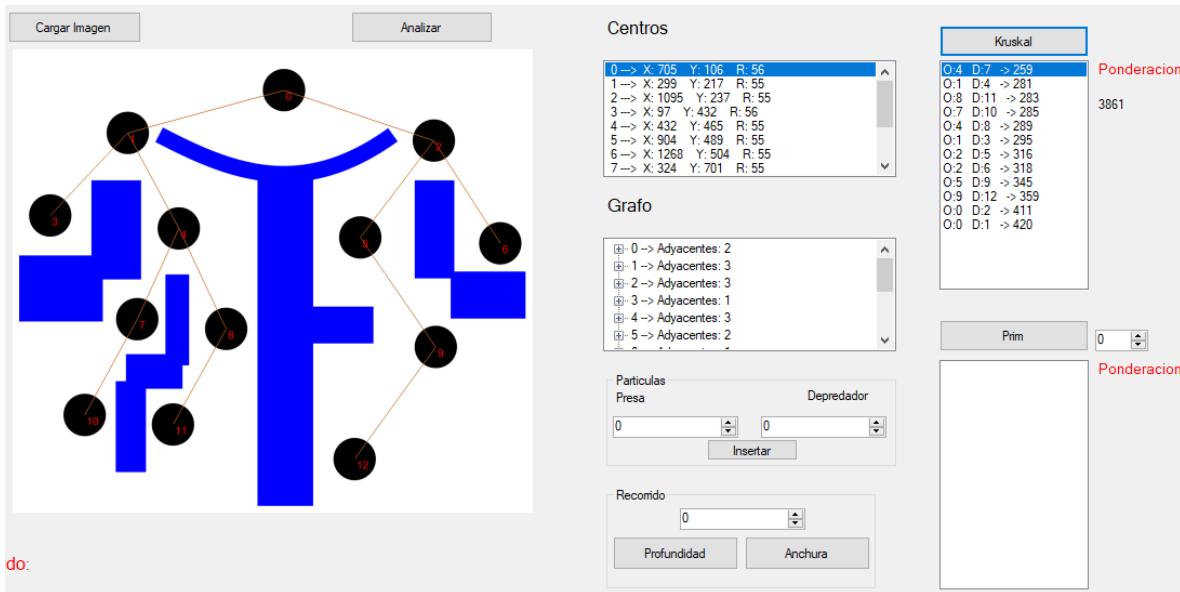


Ilustración 13

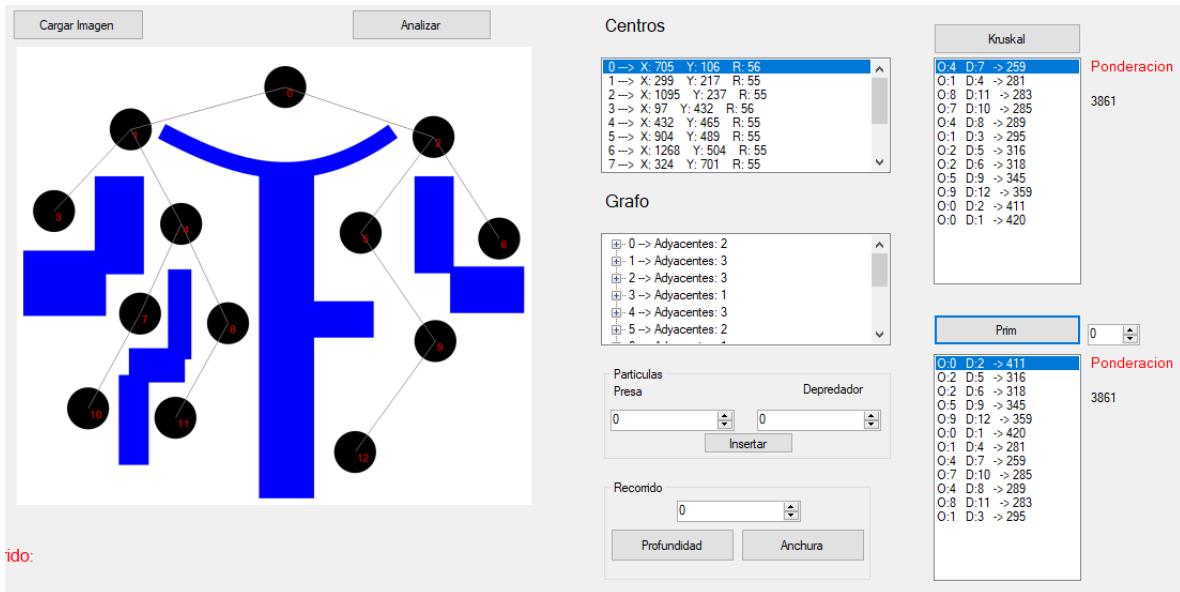


Ilustración 14

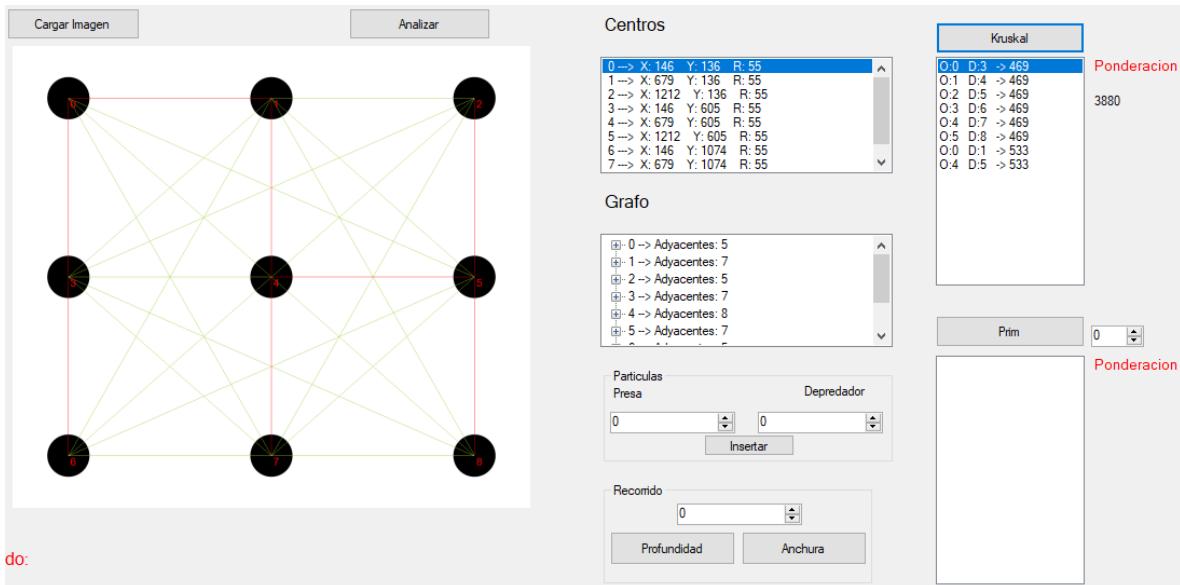


Ilustración 15

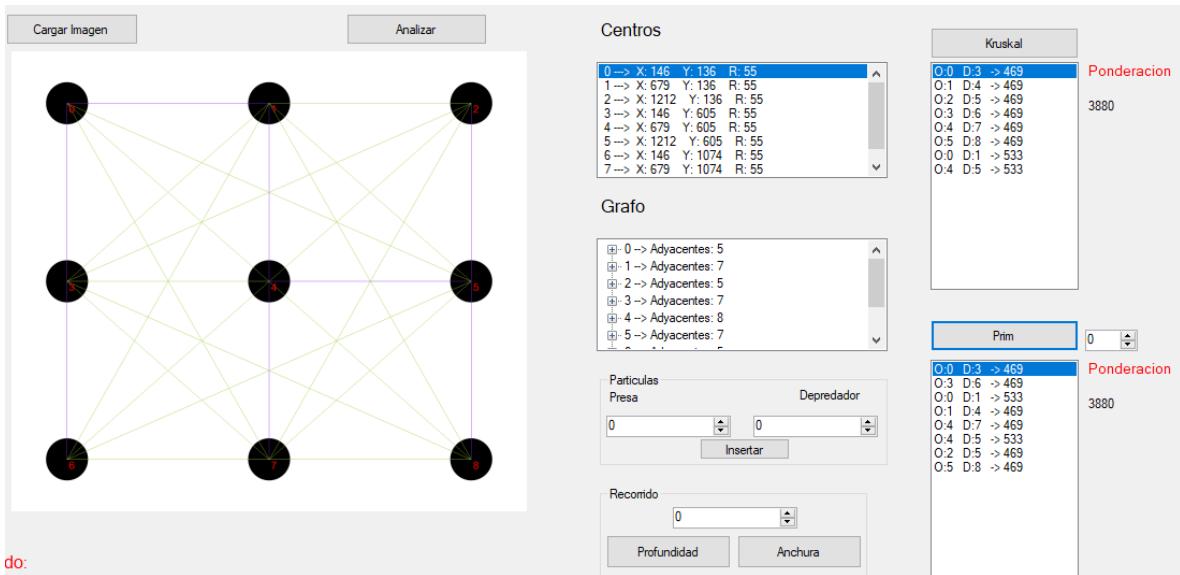


Ilustración 16

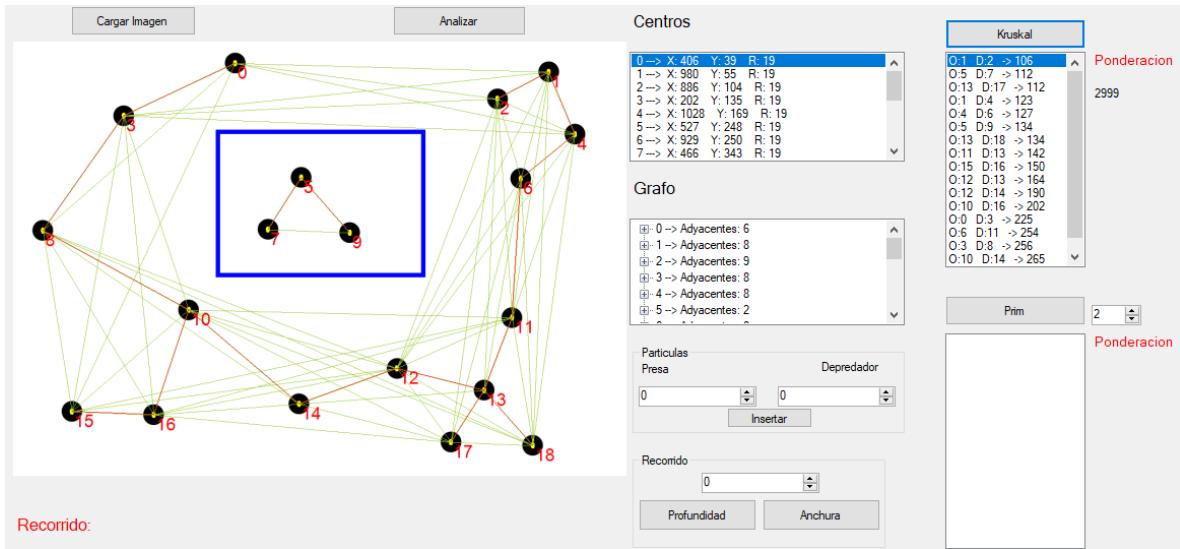


Ilustración 17

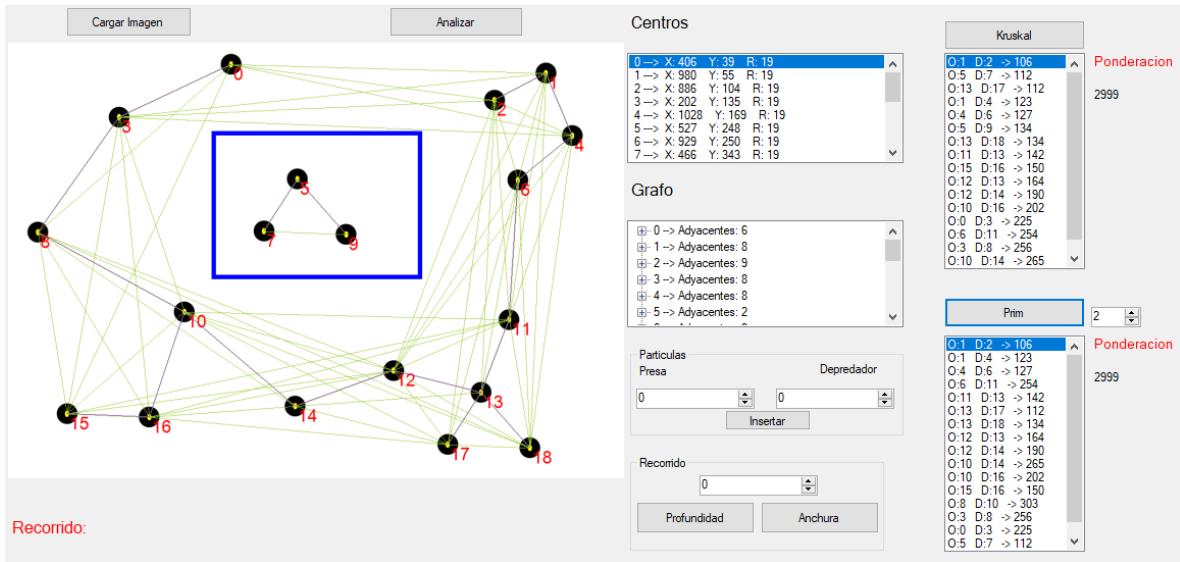


Ilustración 18

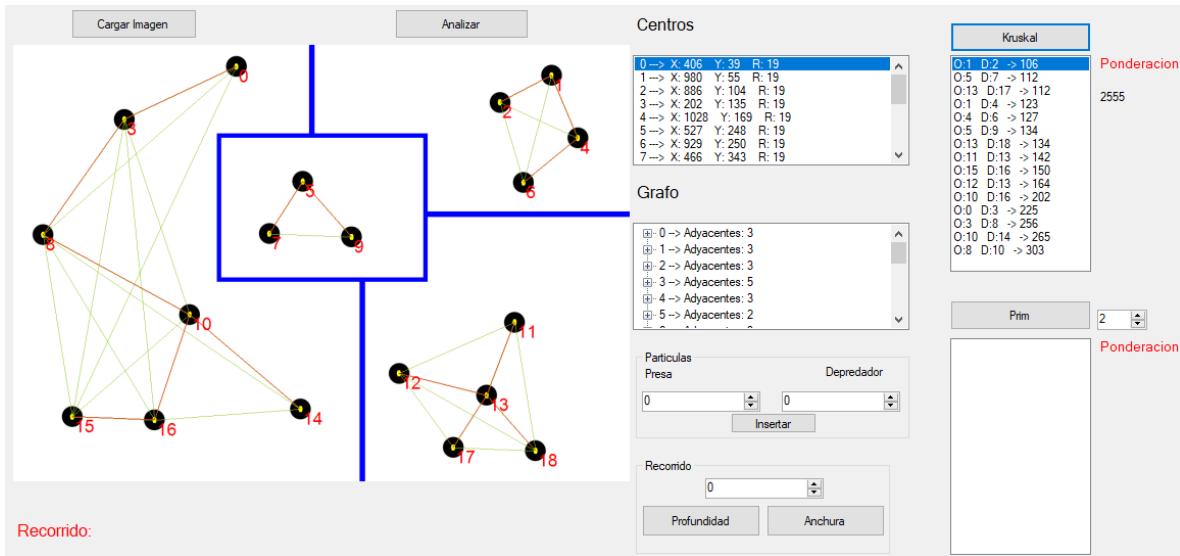


Ilustración 19

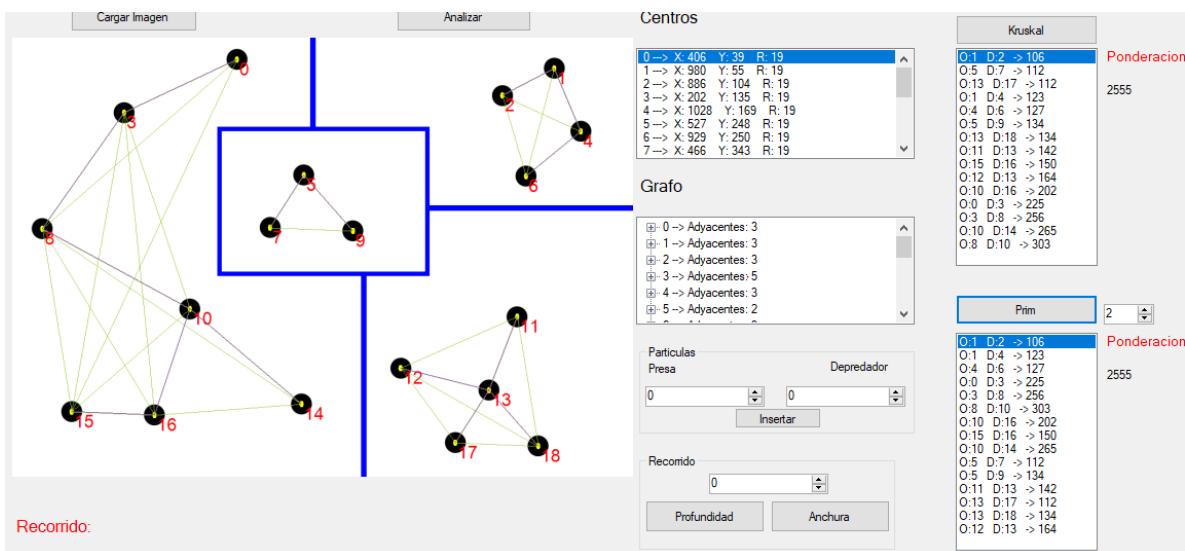


Ilustración 20

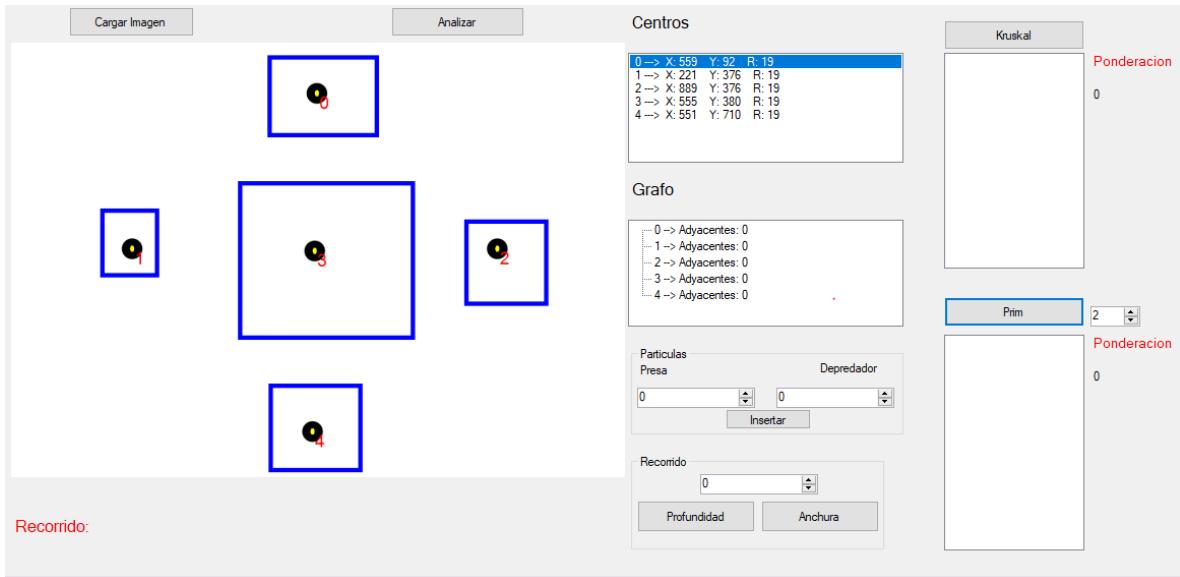


Ilustración 21

Etapa 5:

```
public void SolutionDijkstra(List<int> Recorrido, int Origen, int Destino){
    Dijkstra Actual = new Dijkstra();
    List<Dijkstra> Candidatos = new List<Dijkstra>();

    inicializaDijkstra(Candidatos,Origen);
    while(!ItsSolutionDijkstra(Candidatos,Destino)){
        Actual = SeleccionDefinitivo(Candidatos);
        Actualiza(Candidatos,Actual);
    }
    getCaminoDijkstra(Candidatos,Recorrido,Destino);
}
```

Ilustración 22

```
void inicializaDijkstra(List<Dijkstra> Candidatos,int Origen){
    Dijkstra Candidato;
    foreach(Vertice V in ListaVertices){
        Candidato = new Dijkstra(V.GetOrigen().GetID(),-1,double.PositiveInfinity);
        Candidatos.Add(Candidato);
    }
    Candidatos[Origen].SetPeso(0);
}
```

Ilustración 23

```

Boolean ItsSolutionDijkstra(List<Dijkstra> Candidatos, int Destino){
    if(Candidatos[Destino].GetDefinitivo()){
        return true;
    }
    return false;
}

```

Ilustración 24

```

Dijkstra SeleccionDefinitivo(List<Dijkstra> Candidatos){
    Dijkstra Seleccion = new Dijkstra(-1,-1,double.PositiveInfinity);
    foreach(Dijkstra D in Candidatos){
        if(!D.GetDefinitivo()){
            if(D.GetPeso() < Seleccion.GetPeso()){
                Seleccion = D;
            }
        }
    }
    Candidatos[Seleccion.GetOrigen()].ChangeDefinitivo();
    return Seleccion;
}

```

Ilustración 25

```

void Actualiza(List<Dijkstra> Candidatos,Dijkstra Actual){
    Dijkstra Aux = new Dijkstra();
    foreach(Arista A in ListaVertices[Actual.GetOrigen()].GetListaAristas()){
        Aux = Candidatos[A.GetDestino().GetOrigen().GetID()];
        if(Aux.GetPeso() > (Actual.GetPeso() + A.GetDistancia())){
            Aux.SetPeso(Actual.GetPeso() + A.GetDistancia());
            Aux.SetProcedente(Actual.GetOrigen());
        }
    }
}

```

Ilustración 26

```

void getCaminoDijkstra(List<Dijkstra> Candidatos,List<int> Recorrido,int Destino){
    Recorrido.Add(Candidatos[Destino].GetOrigen()); // Generar
    for(int i = Destino; ;){
        if(Candidatos[i].GetProcedente() != -1){
            i = Candidatos[i].GetProcedente();
        }else{
            break;
        }
        Recorrido.Add(Candidatos[i].GetOrigen());
    }
    Recorrido.Reverse();
}

```

Ilustración 27

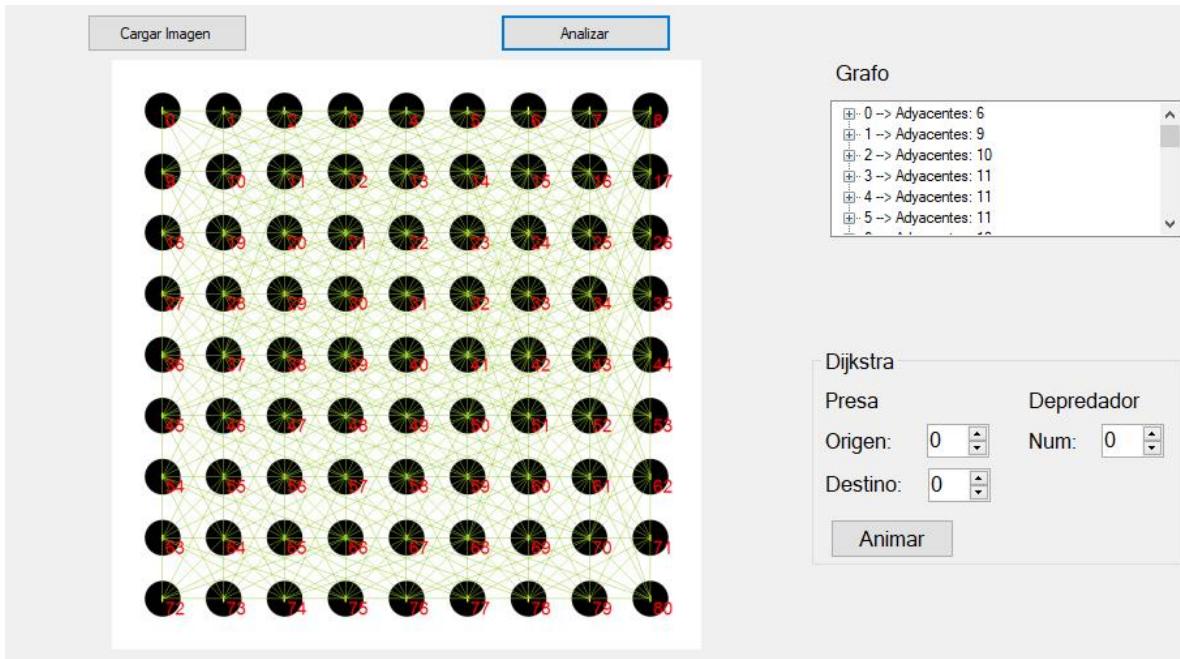


Ilustración 28

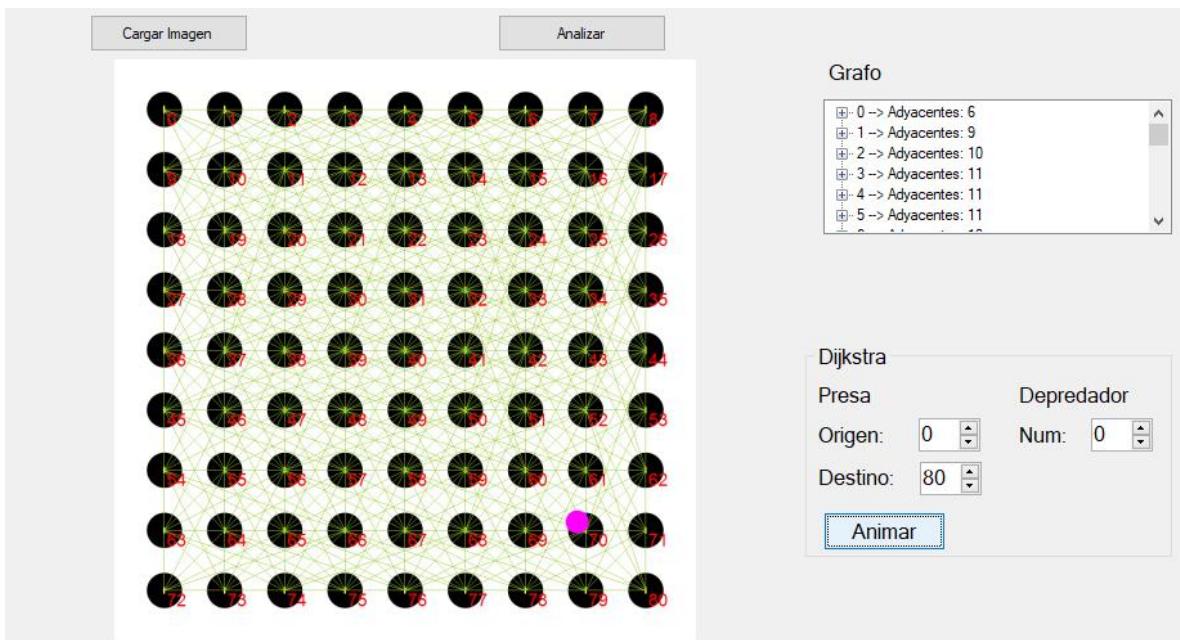


Ilustración 29

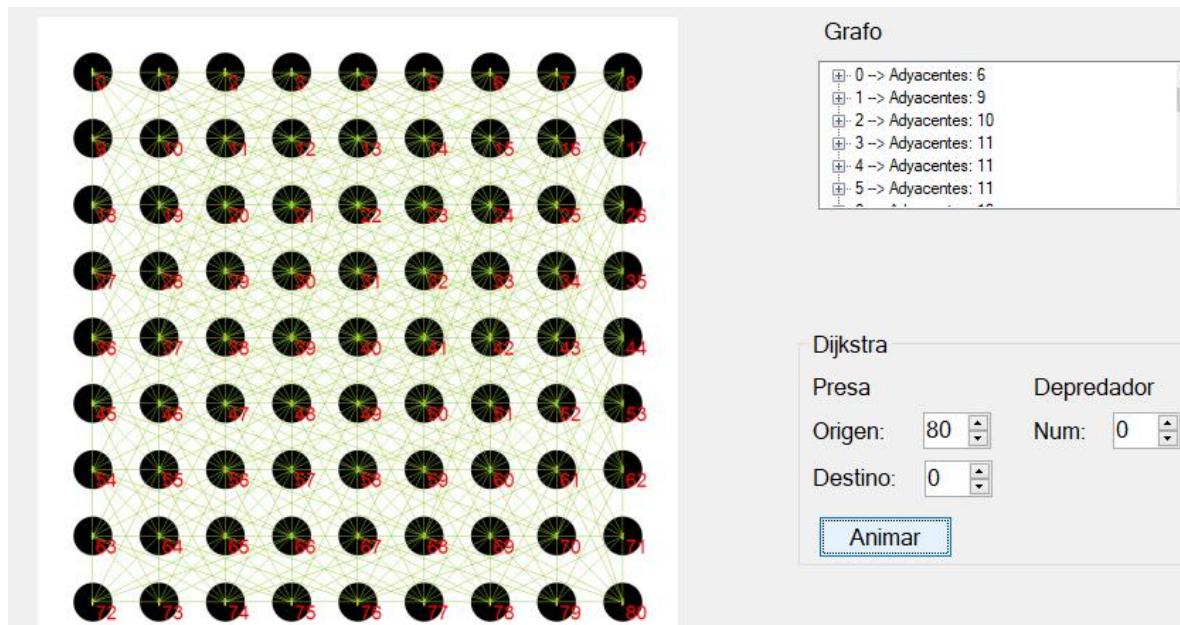


Ilustración 30

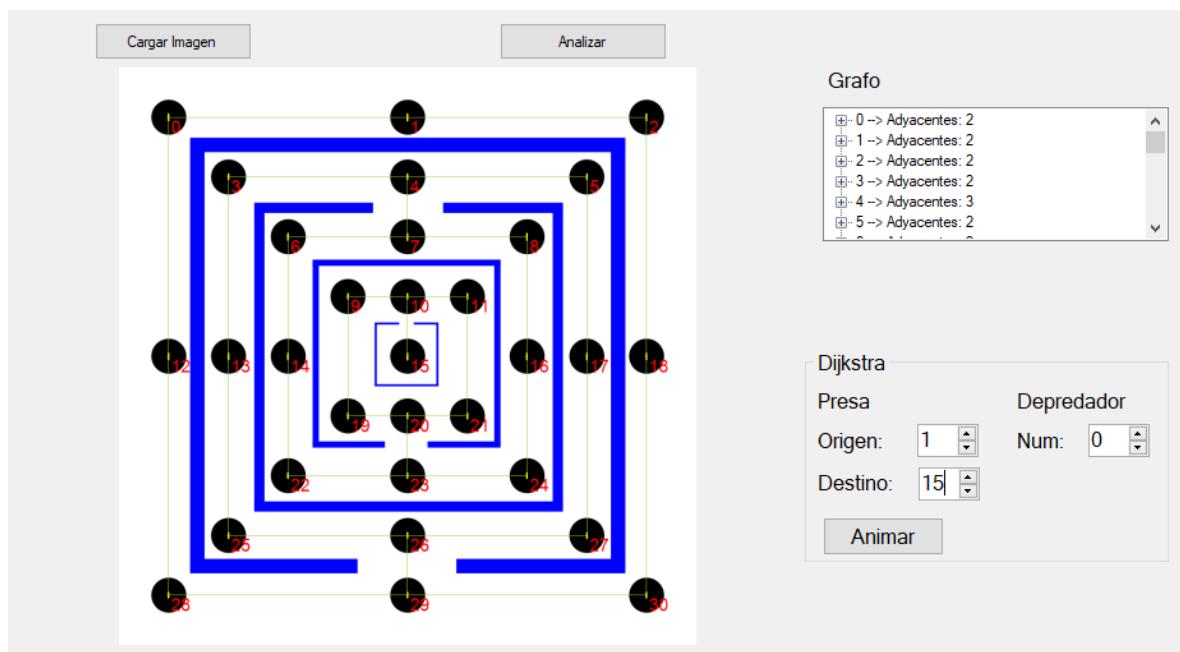


Ilustración 31

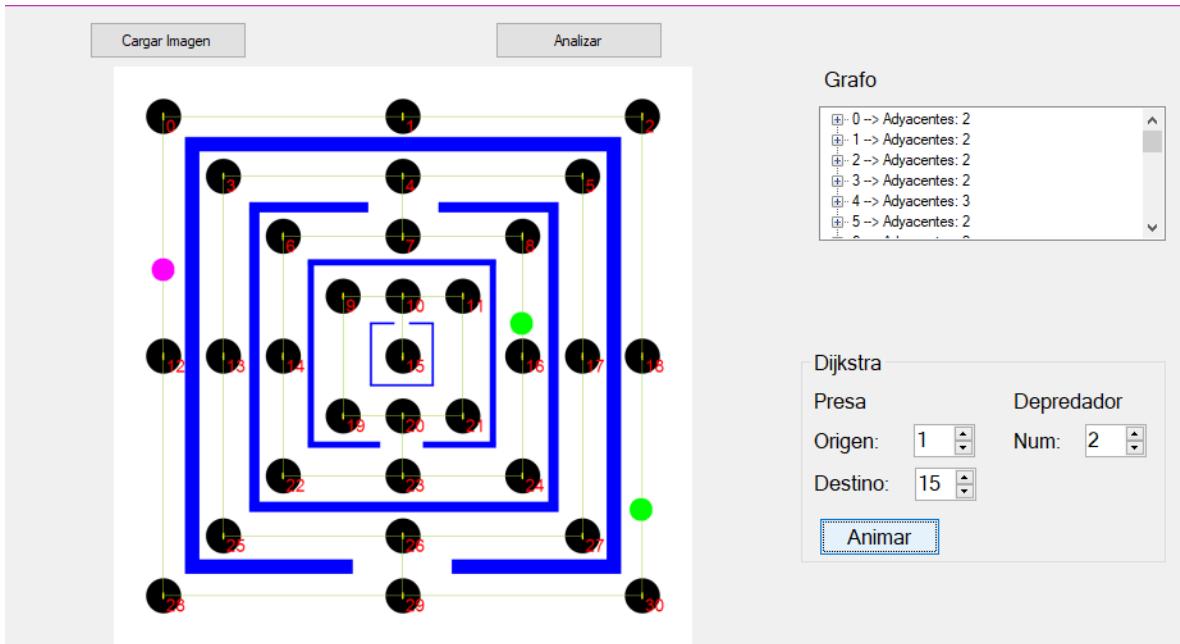


Ilustración 32

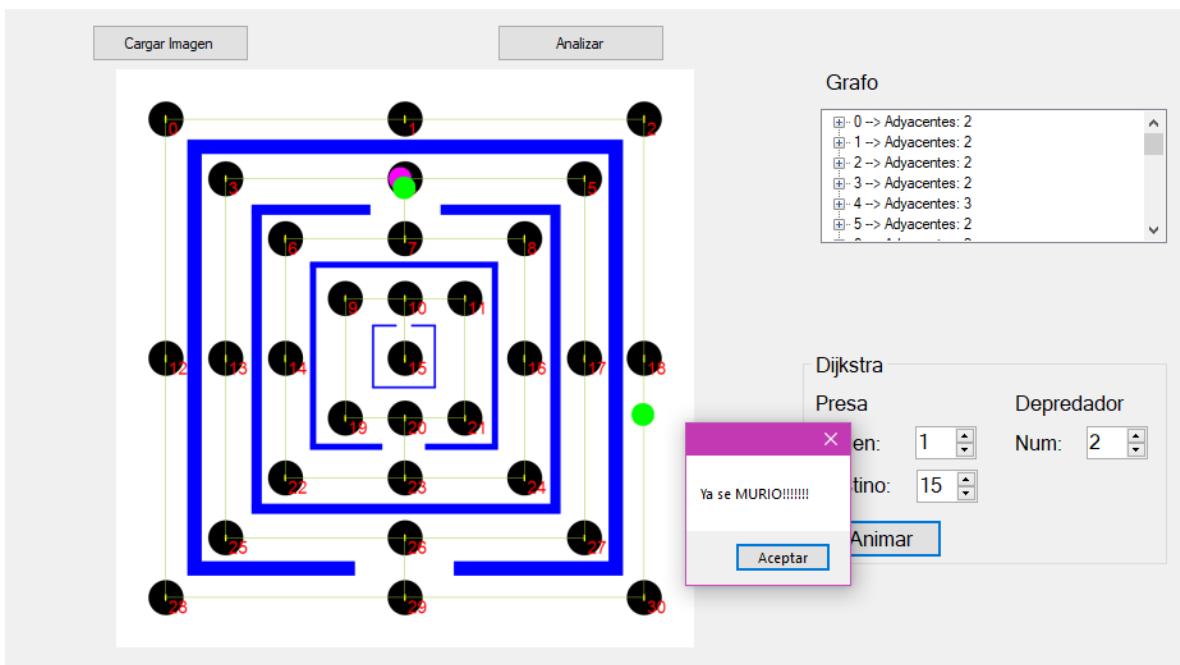


Ilustración 33

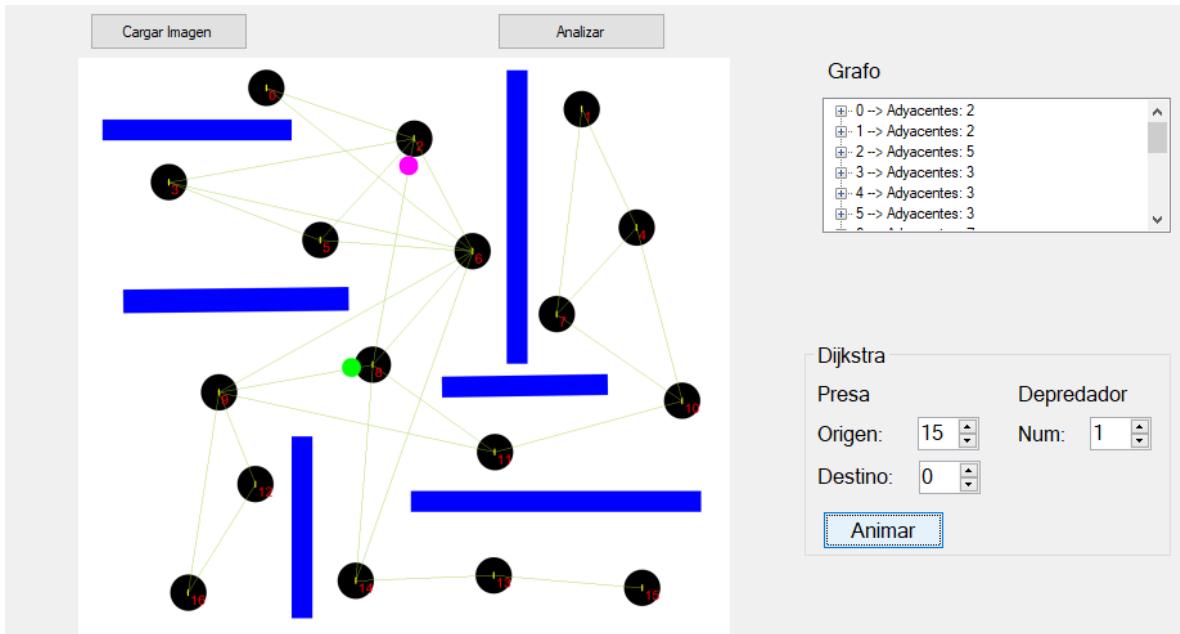


Ilustración 34

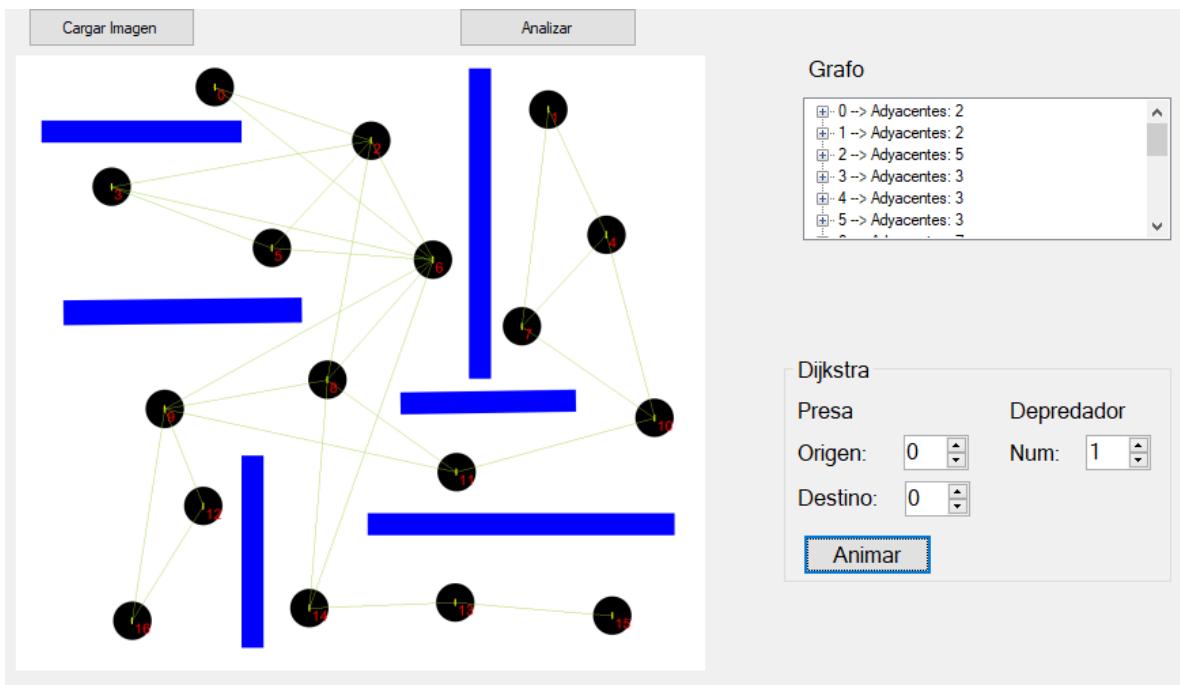


Ilustración 35

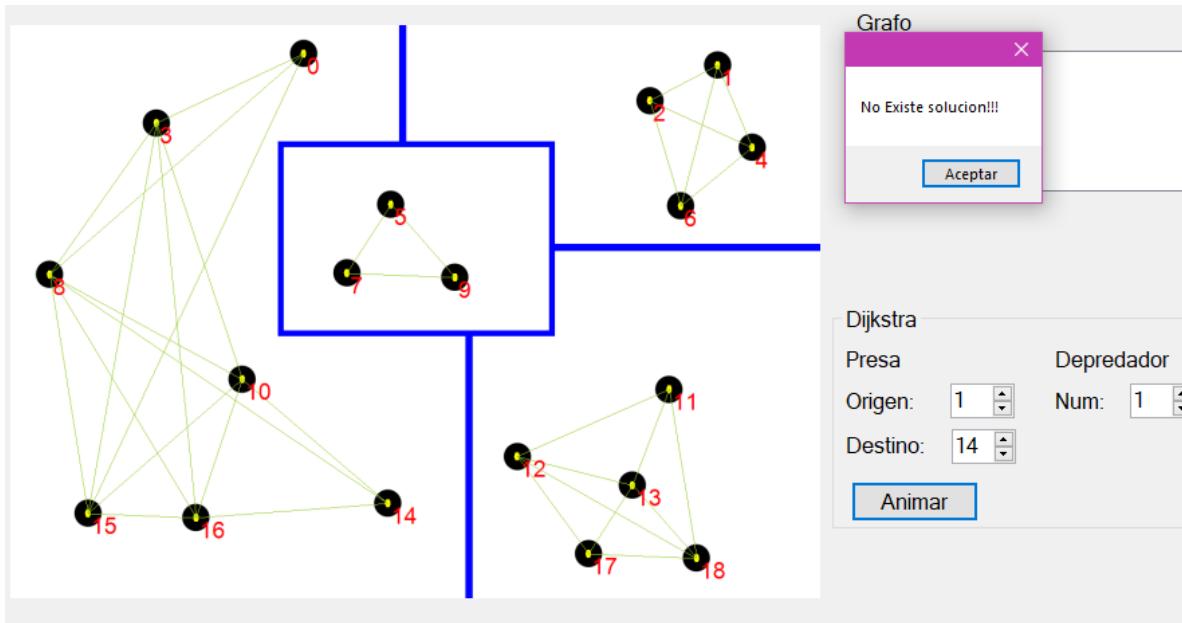


Ilustración 36