Seminario de Solución de Problemas de Algoritmia

GUSTAVO PADILLA VALDEZ

Sección D11

Actividad 04

PLANTEAMIENTO DEL PROBLEMA

Diseñe un sistema computacional en C# con paradigma POO que genere un grafo a partir de la imagen analizada, cada círculo de la imagen representa a un vértice y cada vértice contiene adyacencias (aristas) que lo conecta 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 debe tener la capacidad de añadir una partícula al grafo en un vértice específico, esta partícula será colocada por el usuario y también elegirá el vértice destino.

La partícula debe conocer los caminos más cortos, los caminos más cortos los encuentra a partir del algoritmo de Dijkstra aplicado al grafo generado de la imagen, desde el vértice en el que se encuentra, a todos los demás vértices. Si la partícula ya está en el grafo (en reposo, en algún vértice), se puede seleccionar cualquier vértice destino, y se animará un recorrido del camino mínimo desde el vértice en el que se encuentra, hasta el vértice seleccionado, si existe un camino.

Los recorridos podrán verse de forma animada, la partícula se desplaza desde un vértice a otro a través de sus aristas, es decir, una partícula x puede ir del vértice a al vértice b si existe una arista que conecta a los dos vértices. El desplazamiento es visualmente progresivo (se desplaza sobre la arista).

Estos recorridos de las partículas podrán ser generados infinitamente en los vértices, de tal manera que no afecte visualmente, ni internamente al programa.

También cabe destacar, que por cuestiones visuales, de no sobrecargar la vista del usuario a tal grado que no pueda visualizar lo que pasa, las aristas NO serán pintadas, y solo existirán como datos para los algoritmos.

OBJETIVOS

- Manejo del paradigma de programación POO.
- Creación e implementación exitosa de clases en C#
- Manejo de una programación orientada a eventos
- Poner a prueba las capacidades de codificación.
- Manejo de imágenes.
- Detección de colores en una imagen.
- Dibujado de líneas entre 2 puntos en una imagen.
- Detección de obstáculos en una imagen.
- Manejo del TDA Grafo.
- Implementación y comprensión de grafos.
- Comprensión del algoritmo de Dijkstra.
- Conversión de un grafo dinámico a estático para obtención de caminos.
- Crear una interacción del usuario con una imagen.
- Animación de un punto en un picturebox.

MARCO TEORICO

Lenguaje de programación: Tener el conocimiento básico de programación (principalmente de C#) para la mayor comprensión del problema.

Creación y manejo de clases en C#.

Manejo de interfaces graficas: Entender cómo es que funciona básicamente un programa con interfaz gráfica y como se interactúa con el usuario, así como funciones básicas.

Tipos de datos estructurados: Conocer el cómo funcionan los tipos de datos estructurados como los arreglos, el hecho de saber cómo se declaran, modifican, eliminan, etc. **Programación Orientada a Objetos:** Conocer en que consiste esta manera de programar, y como funciona y difiere de su parte anterior, la programación modular.

Listas: Saber que es una lista y como debe comportarse, que es lo que puede hacer y su función.

Trigonometría.

Grafos: Comprensión del TDA y sus operaciones básicas.

Vértices y Aristas.

Manejo de memoria de variables en C#: Saber de manera básica como es que las variables pueden tomar el lugar de otra, siendo parecidos a los punteros.

Grafos conexos y no conexos.

Grafos ponderados.

Algoritmo Dijkstra: El funcionamiento general del algoritmo.

Ecuación para la recta pendiente.

Interacción con picturebox en C#.

DESARROLLO

Como ya es costumbre, reutilice el código desde la actividad pasada, incluso dejando los métodos del grafo que ya no utilizare por ahora, como el algoritmo de Kruskal y Prim, simplemente quitando los elementos no necesarios y que incluso pueden a llegar a estorbar y complicar las cosas, dejando lo básico. De esta manera ya tengo toda la detección de círculos, junto con la generación de un grafo, solo que esta vez, no lo representare gráficamente para mejor visualización del objetivo de la práctica.

De esta manera empecé por el camino de crear el algoritmo de Dijkstra y luego ya preocuparme por la animación gráfica. Lo primero que necesitaba era implementar una manera de crear un grafo estático que lo represente, para poder aplicar el algoritmo de caminos cortos, pues como se sabe, estos se suelen almacenar en arreglos por simplicidad y efectividad.

Primero, para Dijkstra necesitare un nuevo tipo de vértice, el cual llegara a representar el camino, de tal manera que este vértice contiene una lista de vértices, un peso total y si es el camino definitivo, siendo la clase VerticeD (1).

Con esto ya empiezo la codificación de Dijkstra, este método será un método público de la clase grafo, pues se aplica directamente a él, solamente recibiendo el id del vértice inicial. Primero empiezo por llenar una lista de candidatos de verticesD en base a las aristas del vértice inicial, y su vértice siguiente como el inicial y de esta manera obtener el verticeD más cercano al inicial para empezar mi iteración de encontrar los caminos (2).

Para encontrar los caminos necesitare de un método que será solución, y verificara toda la lista de verticesD y comprobara que sus caminos no sean definitivos y mandara un false, en caso de que sean todos definitivos, mandara un true (3).

De esta manera, mientras que la solución sea falsa, iremos recorriendo la lista de las aristas del vértice para obtener el camino para cada par, verificando y guardando el peso de cada camino, obteniendo el peso total de este, y comparando los pesos para poder obtener los caminos más cortos, una vez obtenido se cambia el valor del VerticeD def a verdadero, el cual indicara que el camino es el definitivo y el más corto, para esto cabe destacar que el algoritmo intentara obtener el camino para cualquier vértice, por lo que se implementa un handler, en caso de llegar a los vértices a los cuales ya no tiene acceso mediante una arista, terminara y regresara los caminos. De esta manera obteniendo los caminos más cortos con Dijkstra (4).

Teniendo el Dijkstra funcional y encontrando el camino, ahora viene la representación visual, primeramente implementare que la animación se haga en base a la imagen cargada, por lo cual tendré que tener una variable de control para el tamaño cuando el usuario haga click, para esto cambiare el tamaño de la imagen en base a constantes, y así obtener una variable del tamaño, que podré usar a la hora de que el usuario haga click en la imagen, y así coincida en valores, el click y la posición x, y (5).

De esta manera, ya es visualmente correcto el click del usuario con relación a la imagen, ahora como siguiente paso, solo le hare caso al click, si este se hizo dentro de un vértice, esto no es mayor problema pues el circulo internamente ya tiene un método para calcular si un pixel es parte de él, así que solo recorro la lista entera de círculos para verificar si se dio click en alguno de ellos (6).

Ahora cuando se de click es cuando empezara una animación, para una mejor representación meramente gráfica, tendré 3 bitmaps, uno que contendrá la imagen original donde solo se colorearon los círculos, centros y ID´s, un buffer que me servirá para hacer más visual el elegir vértices, pues se resaltara su centro, y cuando acabe o falle, regresara al estado original. Por ultimo tendré un bitmap vacío que será solo para la animación.

Con una variable entera controlare en que click está, si es el primer click, este creara la partícula en el bitmap buffer, obtendrá el círculo origen, y cambiara el entero para el siguiente click (7).

Ahora como siguiente paso, si el click que se dio es el segundo, crea la partícula, y manda llamar el método para ejecutar Dijkstra con el vértice destino y origen, se evalúa si es el mismo vértice y regresa un null en caso de serlo, si no lo es obtiene el camino de las distancias como lista de VerticesD, empieza desde el vértice destino y para hasta llegar al inicial, esto estará dentro de un handler, en caso de que no exista un camino, el handler evitara que el camino se llene infinitamente y mandara un mensaje de que no existe tal camino (8).

Teniendo el arreglo de distancias, si se obtuvo null, se reinicia la imagen a la original y no se ejecuta ninguna animación, en caso de que si se reciba, ahora procederá a ejecutar la animación.

Esta animación trabajara sobre otro bitmap vacío copia de la imagen, donde se ira dibujando punto por punto la partícula recorriendo la arista que conecta los vértices, a través de cálculos. Esta ira borrando su rastro en su propio bitmap, mientras que el background de la imagen no se ve afectada, dando el efecto de la animación de la partícula (9).

Así, el usuario podrá ver el recorrido de la partícula cuantas veces quiera, del origen al destino que sea, y cargando incluso diferentes imágenes sin cerrar el programa, dado a su versatilidad.

PRUEBAS Y RESULTADOS

Pantalla Principal:

Imagen cargada y analizada lista para interactuar:

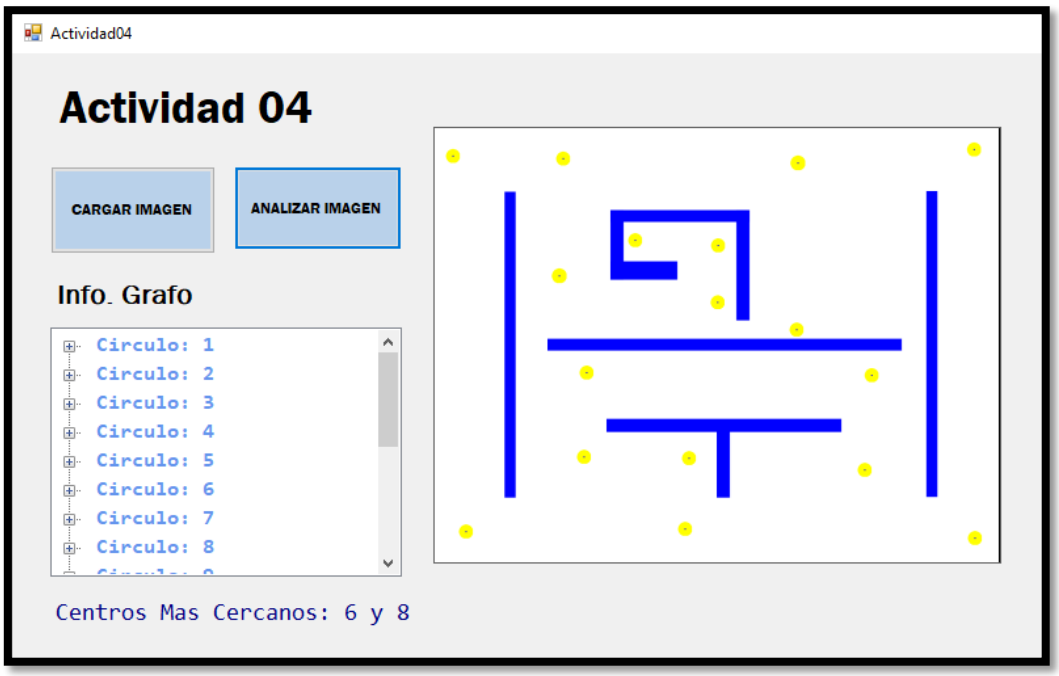
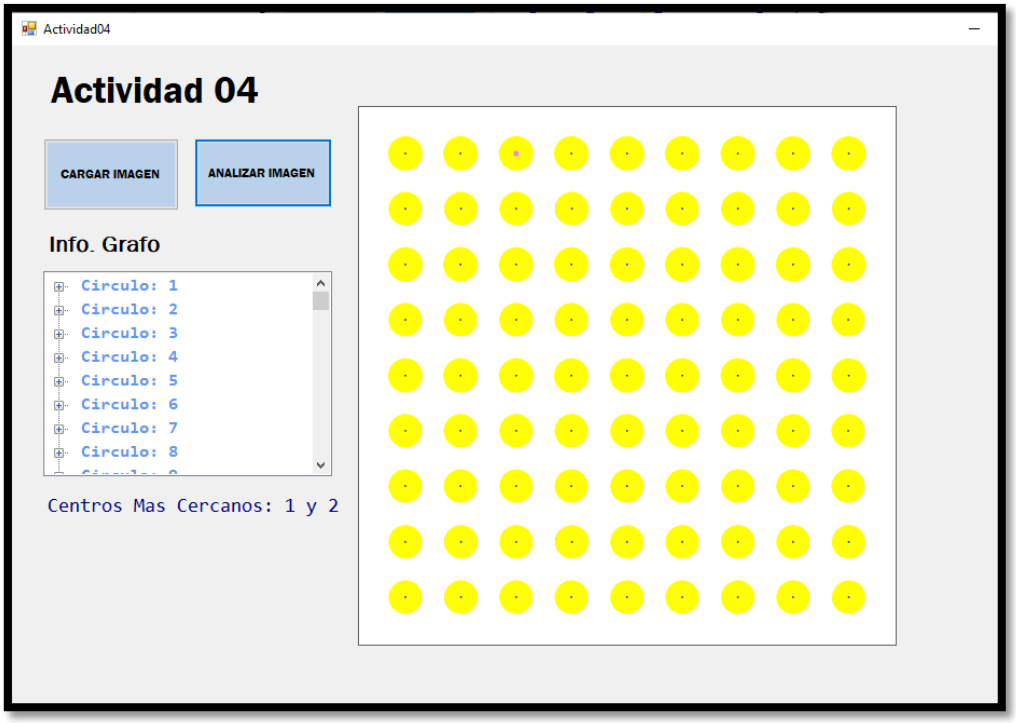
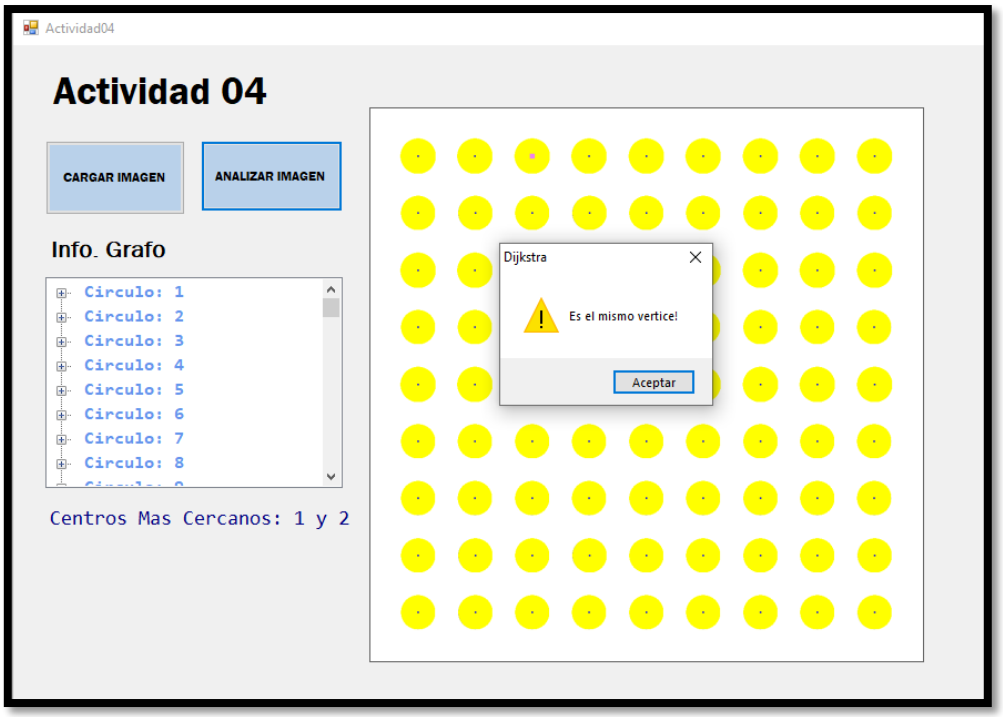


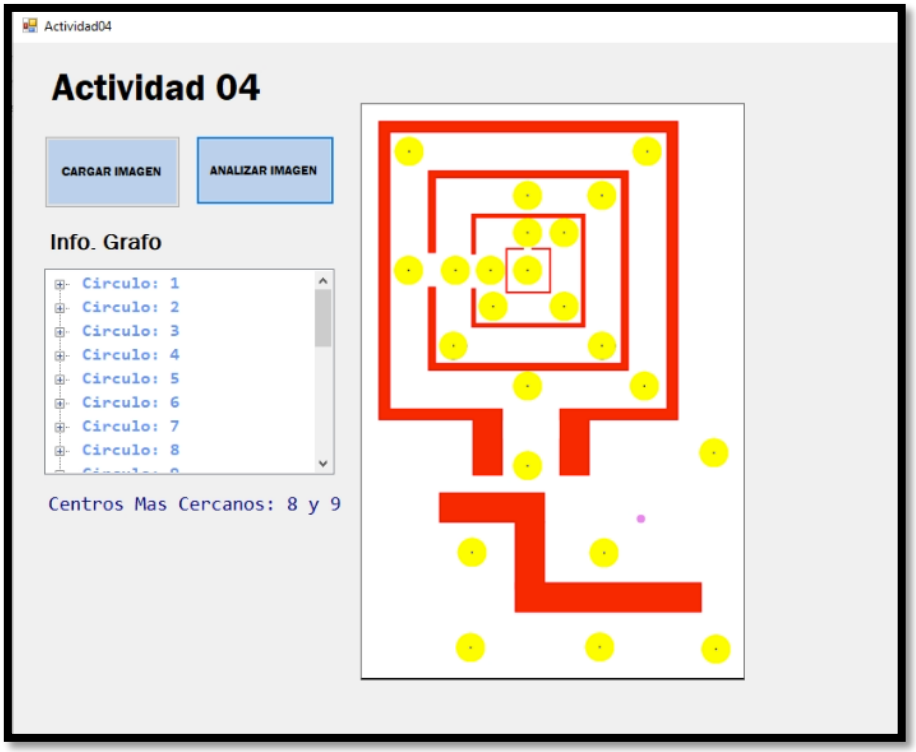
Imagen con un vértice origen seleccionado:



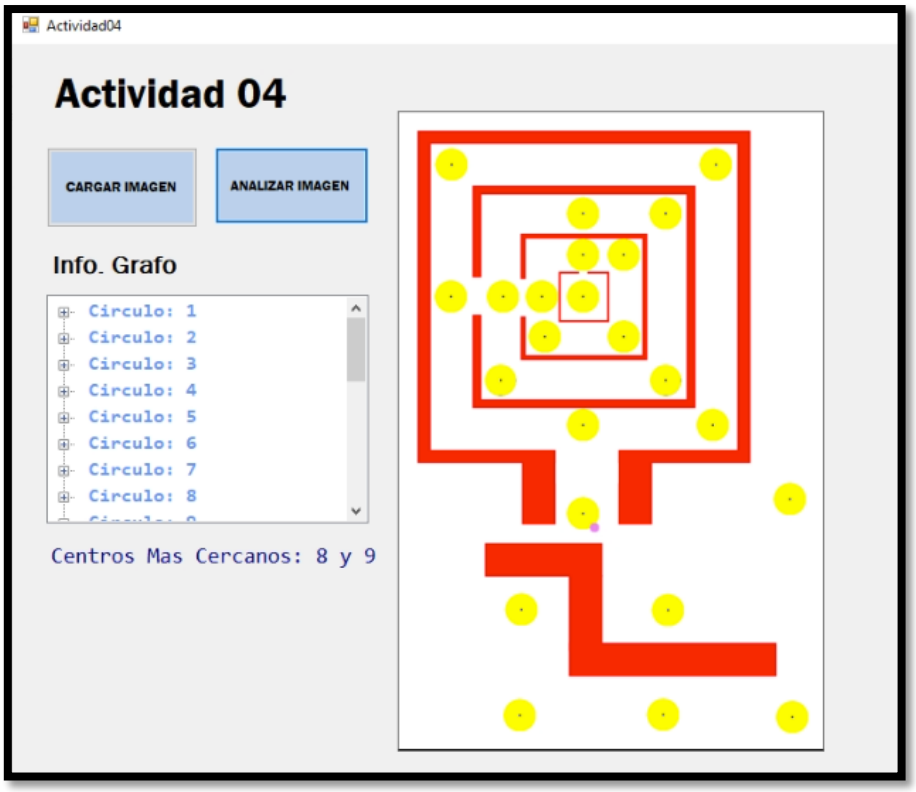
Programa cuando el vértice origen y destino seleccionados son el mismo:



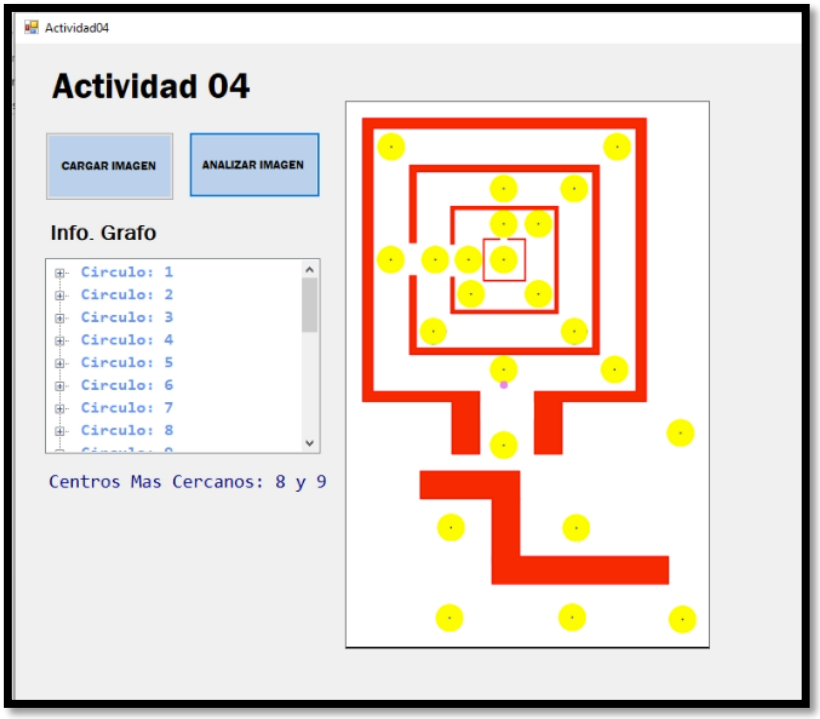
Partícula en movimiento hacia un vértice (Misma Corrida):



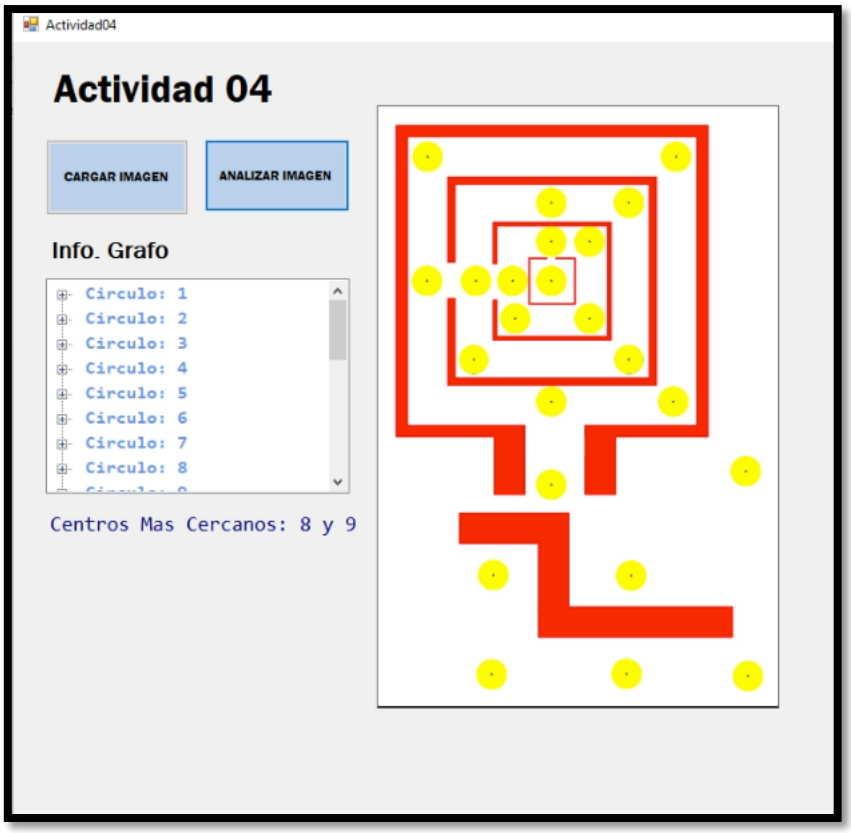
Partícula en movimiento hacia otro vértice (Misma Corrida):



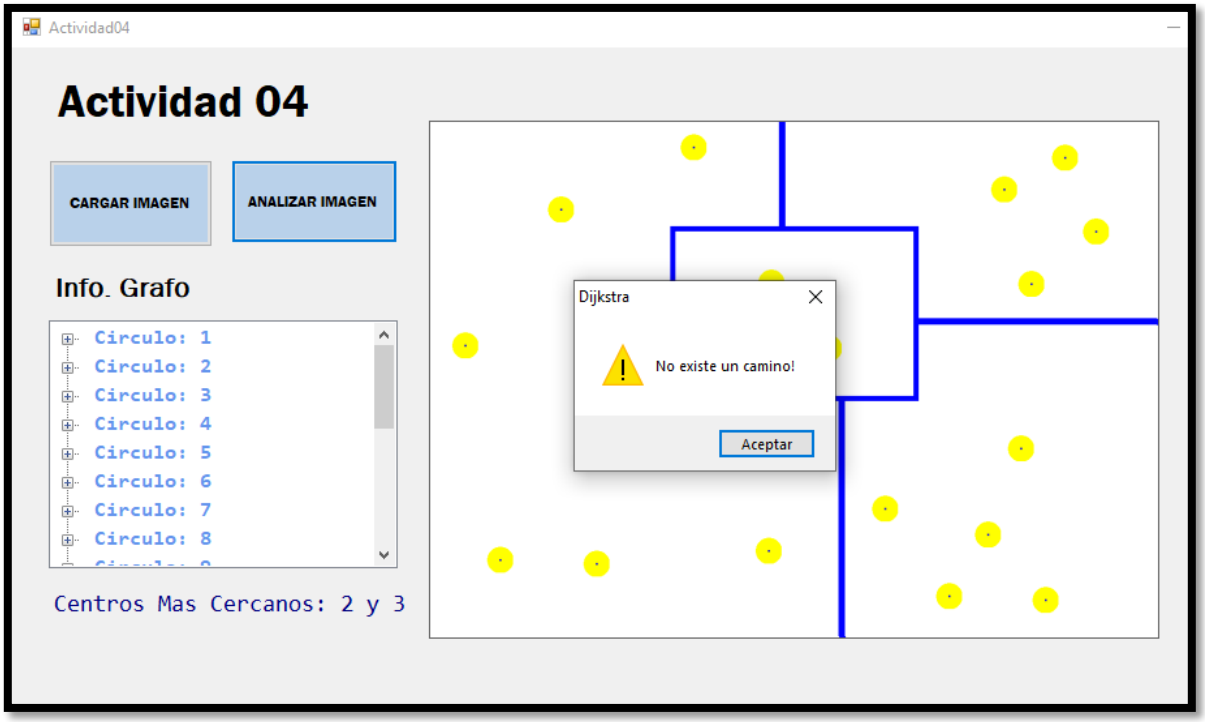
Partícula en movimiento hacia último vértice de su recorrido:



Recorrido terminado, listo para otro recorrido nuevo:



Programa cuando eliges 2 vértices sin camino entre ellos:



CONCLUSIONES

Esta práctica en mi opinión no fue complicada, el algoritmo de Dijkstra se me hizo menos complejo de entender que los anteriores para crear ARM de Kruskal y Prim, y no me costó mucho la implementación. En este caso tarde más en la GUI y la animación, para que funcionara bien la interacción, y el traslado de la partícula, dado que había que hacer un poco más de investigación sobre las propiedades de los widgets, además del entendimiento de cómo sería una animación a lo "fuerza bruta" como se implementó.

APENDICES

(1)

```
public class VerticeD
  {
    public Vertice vertice = null;
    public VerticeD verticeP = null;
    public float peso = Single.PositiveInfinity;
    public bool def = false;
  }
```

(2)

```
        List<VerticeD> candidatos = new List<VerticeD>();
        Vertice vI = new Vertice();
        Vertice aux=fst;
        while(aux != null)
        {
          if(aux.getID() == id)
            vI=aux;
          VerticeD v = new VerticeD();
          v.vertice = aux;
          v.verticeP = v;
          candidatos.Add(v);
          aux=aux.getSig();
        }

        int tmpI = candidatos.FindIndex(x => x.vertice == vI);
        int dbg = tmpI;
        VerticeD tmp = candidatos[tmpI];
        tmp.peso = 0;
        tmp.def = true;
        candidatos[tmpI] = tmp;
        VerticeD actual = candidatos[tmpI];
```

(3)

```
private bool solucion(List<VerticeD> can)
    {
      foreach (var c in can)
      {
        if (!c.def)
          return false;
      }

      return true;
    }
```

(4)

```
while (!solucion(candidatos))
        {
           Arista auxA=actual.vertice.getAdy();
           while(auxA != null)
           {
              tmpI =  candidatos.FindIndex(x => x.vertice == auxA.getAdy());
              tmp = candidatos[tmpI];
              if (!tmp.def)
              {
                 float peso = actual.peso + (float)auxA.getDe();
                 if (tmp.peso > peso)
                 {
                    tmp.peso = peso;
                    tmp.verticeP = actual;
                    candidatos[tmpI] = tmp;
                 }
              }
              auxA=auxA.getSig();
           }

           VerticeD min = new VerticeD();
           min.peso = Single.PositiveInfinity;
           bool f = false;
           foreach (var can in candidatos)
           {
              if (!can.def)
              {
                 if (can.peso < min.peso)
                 {
                    min = can;
                 }
                 f = true;
              }
           }
           try{

              if (f)
              {
                 tmpI =  candidatos.FindIndex(x => x.vertice == min.vertice);
                 tmp = candidatos[tmpI];
                 tmp.def = true;
                 actual = tmp;
              }
```

```
            }catch{
               return candidatos;
            }
         }
         return candidatos;
      }
```

(5)

```
void CargarClick(object sender, EventArgs e)
      {
         if(abrir.ShowDialog() == DialogResult.OK)
         {

            label_NOIMG.Hide();
            imagen.Image = Image.FromFile(abrir.FileName);
            if(imagen.Image.Height > imagen.Image.Width)
               tamano = imagen.Image.Height / 375;
            else
               tamano = imagen.Image.Width / 333;
            imagen.Width = imagen.Image.Width/tamano;
            imagen.Height = imagen.Image.Height/tamano;
            restart();
         }

      }
```

(6)

```
public bool esParte(int xA,int yA)
    {
      int A,B,C,fc;
      A=(x-xA);
      A=A*A;
      B=(y-yA);
      B=B*B;
      C=radio;
      C=C*C;
      C=C+10;
      fc=A+B-C;
      if(fc<=0)
      {
        return true;
      }
      return false;
    }
```

(7)

```
        else
            {
              clic=1;
              origenDij=c;
              crearParticula(c,9);
              imagen.Image=buffer;

            }
```

(8)

```
List<VerticeD> Camino(int origen,int destino)
    {
      if(origen==destino)
      {
        if (MessageBox.Show("Es el mismo vertice!", "Dijkstra", MessageBoxButtons.OK,
                    MessageBoxIcon.Exclamation) == DialogResult.OK)
        {
          return null;

        }
        return null;
      }
      List<VerticeD> distancias= new List<VerticeD>();
      distancias=grf.Dijkstra(origen);
      Vertice cI = grf.circuloToVert(origen);
      VerticeD cF = distancias.Find(x=>x.vertice.getID() == destino);
      List<VerticeD> camino = new List<VerticeD>();

      try{
        while(cF.vertice != cI)
        {
          camino.Add(cF);
          cF = cF.verticeP;
        }
        camino.Reverse();
      }catch{
        if (MessageBox.Show("No existe un camino!", "Dijkstra", MessageBoxButtons.OK,
                    MessageBoxIcon.Exclamation) == DialogResult.OK)
        {
          return null;

        }
        return null;

      }
      return camino;

    }
```

(9)

```
void animacion(float x_0,float y_0,float x_f,float y_f)
    {
      anim = new Bitmap(orig.Width,orig.Height);
      imagen.Image = anim;
      if(x_0 == x_f)
        x_f+=1;
      float r=10;
      Graphics g = Graphics.FromImage(anim);
      Brush br = new SolidBrush(Color.Violet);
      Brush wh = new SolidBrush(Color.White);
      float x_k=x_0,y_k=y_0,m,b,x_ant=x_0,y_ant=y_0;
      int inc=1;m = (y_f - y_0) / (x_f - x_0); b = y_f - x_f * m;
      if(m>-1 && m<1)
      {
        if(x_f < x_0)
          inc = -1;
        for(x_k = x_0+1; x_k != x_f; x_k+= inc)
        {
          y_k = m*x_k+b;
          g.FillEllipse(wh,(int)(Math.Round(x_k-inc-r)),(int)(Math.Round(y_ant-r)),2*r,2*r);
          g.FillEllipse(br,(int)(Math.Round(x_k-r)),(int)(Math.Round(y_k-r)),2*r,2*r);
          imagen.Refresh();
          y_ant=y_k;
          g.Clear(Color.Transparent);

        }
      }
      else {
        if(y_f < y_0)
          inc = -1;
        for(y_k = y_0; y_k != y_f; y_k +=inc)
        {
          x_k = (y_k - b) / m;
          g.FillEllipse(wh,(int)(Math.Round(x_ant-r)),(int)(Math.Round(y_k-r-inc)),2*r,2*r);
          g.FillEllipse(br,(int)(Math.Round(x_k-r)),(int)(Math.Round(y_k-r)),2*r,2*r);
          imagen.Refresh();
          x_ant=x_k;
          g.Clear(Color.Transparent);
        }
      }
    }
```