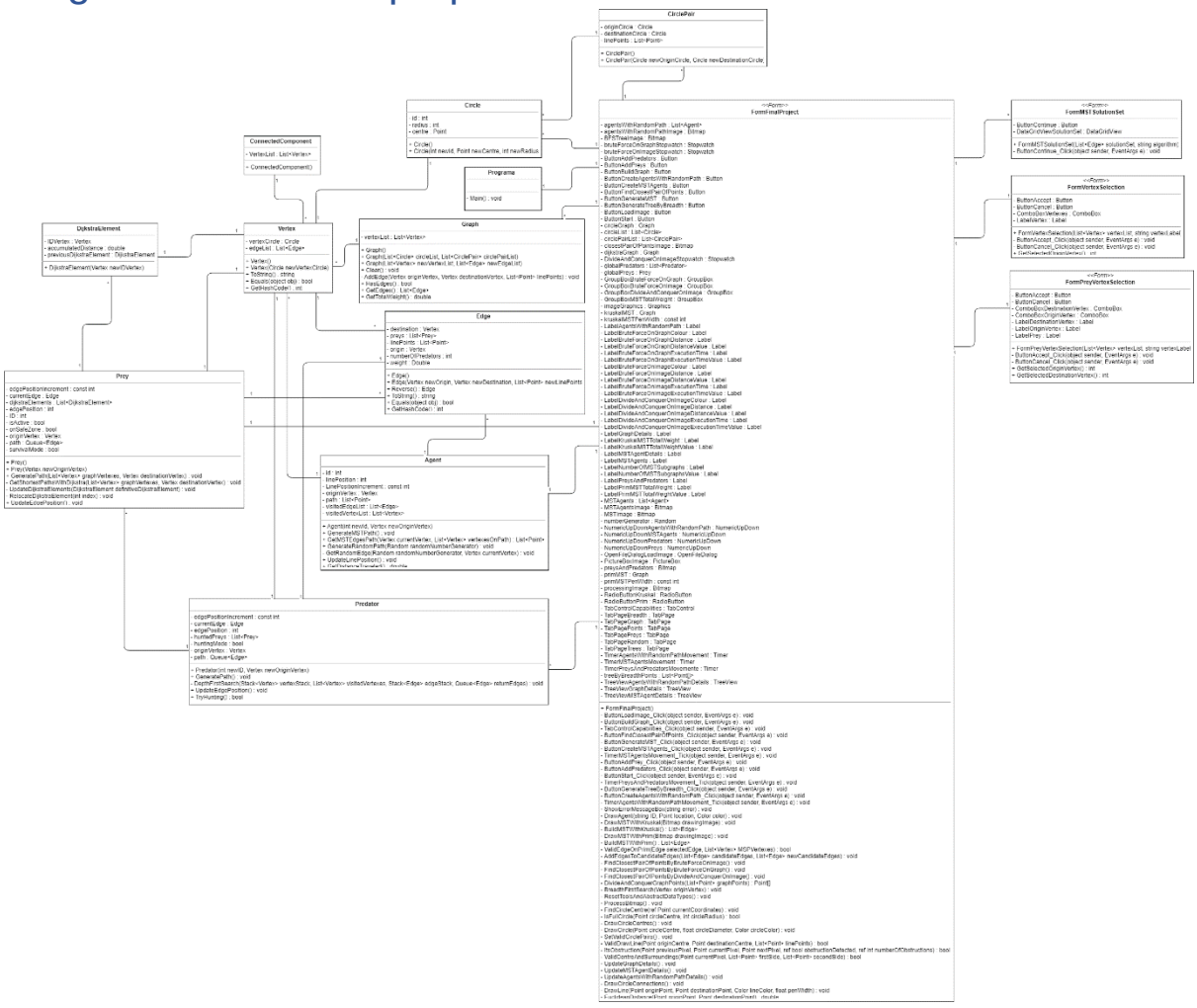
PROYECTO INTEGRADOR

Seminario de Solución de Problemas de Algoritmia

31/05/2019

Shamir Issai Caro Ortiz
216788146

Diagrama de clases propuesto



Objetivo

Desarrollar un sistema computacional que recopile todos los requerimientos de todas las etapas previamente desarrolladas.

El sistema debe de ser capaz de generar un grafo mediante una imagen, esta es elegida por el usuario. Cada circulo identificado en la imagen representa un vértice, y cada línea recta desde el centroide de un circulo a otro, una arista. La arista solamente puede ser trazada si entre el centroide origen y el destino no existe obstrucción alguna. La información del grafo debe de ser mostrada en pantalla; esto es, mostrar los vértices junto con sus aristas.

El sistema debe de contar con la posibilidad de calcular el par de puntos más cercanos en la imagen; esto es, que la distancia entre ellos sea la menor de todos los centroides identificados. El cálculo de este par debe de ser realizado mediante la estrategia de “fuerza bruta” y “divide y vencerás”, al final del cálculo debe de mostrarse en pantalla el par de puntos mas cercanos, la distancia entre ellos y, el tiempo de ejecución de cada estrategia, esto para comparar su eficiencia.

El sistema debe de poder encontrar arboles de recubrimiento mínimo sobre el grafo generado, estos pueden ser encontrados utilizando los algoritmos de Prim y de

Kruskal, el grafo puede o no ser conexo. Al final del cálculo, se debe de mostrar en pantalla el peso total del árbol encontrado junto con las aristas, en orden de elección, utilizadas para llevar a cabo la construcción del árbol.

El sistema debe de contar con la capacidad de poder crear agentes de distintos tipos. Un tipo de agente debe de recorrer el grafo generado de manera aleatoria; sin embargo, dentro de su recorrido no debe de utilizar la misma arista más de una vez, en caso de que exista más de un agente de este tipo, se debe mostrar en pantalla el agente que más vértices recorrido, el criterio de desempate reside en la mayor distancia recorrida.

Otro tipo de agente debe de recorrer los árboles generados mediante Prim y Kruskal asegurándose que pasará por todos los vértices del grafo utilizando únicamente las aristas del árbol.

Los últimos tipos de agente se dividen en dos: presas y depredadores. Los primeros, pueden llegar a cualquier otro vértice a partir de su vértice origen, esto siempre y cuando exista una ruta entre estos. El camino que toma el agente debe de ser el de menor peso acumulado. Por si parte, los segundos recorren el grafo por profundad y tienen el objetivo de consumir presas, cuando esto sucede, la presa desaparece.

Por último, el sistema debe de ser capaz de verificar si es posible crear un árbol en el que todas sus hojas cuenten con el mismo nivel, este árbol debe de ser creado recorriendo en amplitud el grafo generado a partir de la imagen seleccionada.

Marco teórico

Mapa de bits

Un mapa de bits es un formato de imagen que puede ser utilizado para crear y almacenar gráficos de computadora. Un archivo de mapa de bits muestra pequeños puntos que juntos forman una imagen. Una imagen de mapa de bits es una cuadricula formada por filas y columnas en donde a cada celda (píxel) se le da un valor que la llena o la deja en blanco, de esta manera se crea una imagen por medio de información. Al convertir una imagen a un mapa de bits, los píxeles pueden ser manipulados usando sus métodos, basta con conocer sus coordenadas para hacerlo.

Ecuación de una recta

Al conocer las coordenadas de dos puntos en el plano cartesiano, es posible trazar una recta que pase por ellos mediante la ecuación:

$$y = mx + b$$

En esta ecuación, x y y son coordenadas de un punto, m es la pendiente y b es la coordenada y de la intersección en y.

Para calcular la pendiente m se utiliza la siguiente ecuación:

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

En donde x_1 y y_1 son las coordenadas de un punto y, x_2 y y_2 son las del otro.

Para calcular la coordenada b se utiliza la siguiente ecuación:

$$b = y - mx$$

En donde x y y son coordenadas de un punto.

En situaciones donde la pendiente es mayor a 1 o menor que -1. Es necesario iterar en y, ya que si se itera sobre x la recta será inconsistente. Para esta situación se utiliza la ecuación:

$$x = \frac{y - b}{m}$$

Grafo

Un grafo es un conjunto de vértices unidos por enlaces llamados aristas. Una arista puede presentar o no una ponderación, esta puede indicar ya sea el costo o beneficio de una transición. Un grafo modela las relaciones que presenta un conjunto de elementos y, permite estudiar la manera en la que interactúan los unos con los otros.

Se puede definir un grafo como un par ordenado:

$$G = (V, E)$$

En donde V representa el conjunto de vértices y E el de aristas.

Generalmente, un grafo es gráficamente representado como un conjunto de círculos enlazados por medio de líneas. A partir de esta representación, es posible formar una lista de adyacencia, esta es una representación no gráfica de un grafo. Cualquier problema puede modelarse fácilmente mediante el uso de un grafo, ya que su representación es simple de comprender.

Un grafo puede ser dirigido o no dirigido, el primero se refiere a la unidireccionalidad de las aristas, mientras que el segundo habla de bidireccionalidad.

Algoritmos de fuerza bruta

Un algoritmo de fuerza bruta se caracteriza por enumerar todas las posibles soluciones al problema propuesto, con el fin de verificar que candidato satisface la solución al mismo.

Par de puntos más cercanos mediante divide y vencerás

El algoritmo recibe como entrada un arreglo de n puntos $P[]$ y retorna la menor distancia entre dos puntos del arreglo. El arreglo debe de ser ordenado con respecto a la coordenada x.

1. Encontrar el punto medio del arreglo ordenado, este puede ser $P\left[\frac{n}{2}\right]$.
2. Dividir el arreglo en dos mitades. El primer arreglo contiene los puntos desde $P[0]$ hasta $P\left[\frac{n}{2}\right]$. El segundo arreglo contiene los puntos desde $P\left[\frac{n}{2} + 1\right]$ hasta $P[n-1]$.
3. Encontrar recursivamente la mínima distancia en ambos arreglos. Las distancias serán d_i y d_d. Encontrar el mínimo entre d_i y d_d, el cual será d.

4. Ahora, se deben de considerar los pares de puntos en los cuales un punto se encuentra en la mitad izquierda y el otro en la mitad derecha. Considerar la línea vertical que pasa a través de $P\left[\frac{n}{2}\right]$ y encontrar todos los puntos cuya coordenada x se encuentre más cerca que d a la línea vertical central. Construir un arreglo $strip[]$ con dichos puntos.
5. Ordenar el arreglo $strip[]$ con respecto a la coordenada y.
6. Encontrar la mínima distancia en $strip[]$. Solo es necesario verificar los puntos cuya coordenada y se encuentre a una distancia menor a d.

Retornar el mínimo entre d y la distancia calculada en el paso anterior.

Clase Random

Representa un generador de números aleatorio, el cual es un dispositivo que produce una secuencia de números que cumplen con ciertos requerimientos estadísticos para aleatoriedad.

Los números aleatorios se eligen con igual probabilidad de un conjunto finito de números. Los números elegidos no son aleatorios del todo porque se utiliza un algoritmo matemático para seleccionarlos; sin embargo, son lo suficientemente aleatorios para fines prácticos.

Algoritmos voraces

Los algoritmos voraces suelen ser bastantes simples. Estos son empleados generalmente para la resolución de problemas de optimización. Usualmente, los elementos que conforman un algoritmo voraz son los siguientes:

- Un conjunto de candidatos.
- Un conjunto de decisiones ya tomadas.
- Una función que determina si un conjunto de candidatos representa una solución al problema.
- Una función que determina si a un conjunto se le pueden añadir nuevos candidatos. De esta manera, se verifica si con este nuevo conjunto es posible encontrar una solución al problema.
- Una función de selección que elige el mejor candidato.
- Una función objetivo que provee el valor de una solución.

Algoritmo de Prim

El algoritmo de Prim, dado un grafo conexo, no dirigido y ponderado, encuentra un árbol de recubrimiento mínimo. En otras palabras, es capaz de encontrar un subconjunto de las artistas de un grafo para formar un árbol que incluya todos los vértices del grafo, en donde el peso total de las aristas del árbol es el mínimo posible.

El algoritmo de Prim funciona de la siguiente manera:

1. Se elige un vértice del grafo, este será el vértice de partida.
2. Se selecciona la arista de menor peso que tenga como origen el vértice seleccionado anteriormente y se selecciona el vértice destino de la arista.

3. Se repite el paso dos, esto siempre y cuando la arista seleccionada tenga como destino un vértice que no ha sido previamente seleccionado. Es decir, que la arista seleccionada no genere un ciclo.
4. El árbol de recubrimiento mínimo es encontrado cuando el numero de aristas sea igual al número de vértices menos uno.

Algoritmo de Kruskal

El algoritmo de Kruskal, dado un grafo conexo, no dirigido y ponderado, encuentra un árbol de recubrimiento mínimo. En otras palabras, es capaz de encontrar un subconjunto de las artistas de un grafo para formar un árbol que incluya todos los vértices del grafo, en donde el peso total de las aristas del árbol es el mínimo posible.

El algoritmo de Kruskal funciona de la siguiente manera:

1. Se selecciona, de entre todas las aristas del grafo, la de menor ponderación siempre y cuando no se genere un ciclo.

Se repite el paso uno hasta que el número de aristas sea igual al número de vértices menos uno.

Algoritmo de Dijkstra

El algoritmo de Dijkstra es un algoritmo voraz de la teoría de grafos utilizado para encontrar un árbol de recubrimiento mínimo en un grafo conexo y ponderado. Se encarga de buscar un subconjunto de aristas que, mediante la generación de un árbol, se incluyan todos los vértices asegurándose que el valor total de todas las aristas que forman el árbol es el mínimo.

Pseudocódigo

```
ElemenotDijkstra{
    distanciaAcumulada = infinito;
    definitivo = falso
    proveniente = nulo
}

actualizaVector(VectorDijkstra, ind){
    para cada arista a de G[ind]:
        si a.pond + vectorDijkstra[ind] < vectorDijkstra[destino de a]
            vectorDijkstra[destino de a].distancia = a.pond + vectorDijkstra[ind];
            vectorDijkstra[destino de a].proveniente = G[ind]
}

pesos[N] PesosMinimos(v_o){
    VectorDijstra[N] = inicializa();
    vectorDijkstra[v_o].distancia = 0;
    mientras(!solución(vectorDijkstra)){
        ind = menor(VectorDijstra);
        vectorDijkstra[ind].definitivo = verdadero;
        actualizaVector(VectorDijstra, ind);
    }
}
```

Búsqueda en profundidad

Una búsqueda en profundidad es un algoritmo que permite recorrer todos los vértices de un grafo o árbol de manera ordenada, pero no uniforme. Consiste en ir expandiendo todos y cada uno de los vértices que va localizando, de forma recurrente, en un camino concreto.

Pseudocódigo

```
DFS(pila, visitados){
    v_act = pila.top
    para cada arista "a" de v_act
        si el destino de la arista no pertenece a visitados:
            pila.push(destino de a)
            visitados.inserta(destino de a)
            DFS(pila, visitados)
    pila.pop()
}
```

Búsqueda en anchura

La búsqueda en anchura sobre un grafo supone que el recorrido se lleve a cabo por niveles.

Pseudocódigo

```
cola = v_inicial
visitados = v_inicial
listaAristas BFS(cola,visitados){
    listaAristas;
    si cola.count != 0:
        v_actual = cola.desencolar()
        para cada arista a de v_actual
        si (!visitados.existe(a.destino)):
            visitados.insertar(a.destino)
            cola.encolar(a.destino)
            listaAristas.inserta(a);
        listaAristas += BFS(cola, visitados)
    return listaAristas
}
```

Desarrollo

Lo primero que debe de tener el sistema es la capacidad de poder cargar una imagen cualquiera, esto se realiza mediante el uso de un “OpenFileDialog”, este es utilizado para buscar en el sistema la imagen deseada para posteriormente analizarla. La imagen es convertida a un mapa de bits, esto se hace para poder analizarla píxel por píxel. Este nuevo mapa de bits es mostrado en un “PictureBox”.

Al ya contar con el mapa de bits de la imagen seleccionada, este es analizado con la pulsación del botón “Construir Grafo” (véase apéndice G) en busca de círculos. Esta búsqueda es realizada por el método “FindCentres” (véase apéndice G), la cual busca la ocurrencia de un píxel negro para después calcular el centroide del circulo encontrado.

En base al píxel negro encontrado es posible calcular el punto medio del circulo horizontal y verticalmente; es decir, calcular la posición del centroide. Cuando este es

encontrado, se dibuja utilizando el método “DrawCircle” (véase apéndice G) para de esta manera identificarlo gráficamente y; además, un nuevo objeto de tipo “Circle” (véase apéndice B) es creado e insertado en la lista global de círculos.

Algo que siempre se debe de tener en consideración es la eficiencia del sistema, es por esto por lo que, al analizar píxel por píxel, en caso de encontrar un píxel negro, lo primero que se hace es verificar si el círculo ya fue procesado para de esta manera, evitar procesamiento extra.

Al ya contar con los círculos de la imagen, el siguiente paso consiste en encontrar las conexiones válidas entre estos; es decir, las aristas del grafo; ya que cada círculo representa un vértice. Las aristas pueden ser trazadas si entre ambos vértices, origen y desino, no existe obstrucción alguna. Este proceso es realizado utilizando el método “SetValidCirclePairs”, el cual utiliza un algoritmo de fuerza bruta para calcular todas las posibles conexiones entre todos los vértices del grafo (véase apéndice G). Por su parte, el método anterior utiliza otro de nombre “ValidDrawLine” (véase apéndice G), este método utiliza la educación de la recta para analizar píxel por píxel la ruta entre dos vértices distintos, en cada píxel se manda llamar al método “ItsObstruction”, el cual analiza los ocho píxeles vecinos del píxel actual en busca de cualquier obstrucción. En caso de que se encuentre una arista válida entre dos vértices, se crea un objeto de tipo “CirclePair”, el cual contendrá los dos círculos involucrados. Cuando el método “SetValidCirclePairs” finaliza su ejecución, se cuenta con una lista de todos los pares de círculos que pueden conectarse entre sí.

Al ya contar con una lista de círculos y una lista con las conexiones válidas entre estos, se crea un objeto de tipo “Graph” (véase apéndice F). Este es creado utilizando uno de sus constructores que recibe como parámetro la lista de círculo y la de conexiones válidas, aquí, cada circulo se convierte en un vértice y cada conexión, una arista.

El constructor inserta cada círculo de la lista de círculos en la lista de vértices del grafo a crear, este bucle se realiza un total de V veces, en donde V representa la cantidad de vértices. Después, se crea otro bucle en donde se lleva a cabo la creación de las aristas con ayuda de la lista de conexiones válidas. Este bucle itera un total de E veces, en donde E es el número total de aristas. El algoritmo lo primero que hace es buscar en la lista de vértices el círculo origen del objeto tipo “CirclePair” y después repite el mismo proceso para el círculo destino. Considerando el peor caso, la primera búsqueda verificará V elementos, mientras que la segunda búsqueda lo hare $V - 1$ veces, esto se debe a que no existen los vértices con aristas a sí mismos. En este caso, la complejidad algorítmica del constructor depende de ambas, la cantidad de vértices y de aristas. Esta puede ser estimada mediante la fórmula $V + 2EV - E$.

Un requerimiento del programa establece que los datos del grafo deben de ser visibles al usuario. Esta representación se realiza de dos maneras distintas: gráficamente y en un “TreeView”, en este ultimo se listan todos los vértices junto con todas sus aristas más su ponderación.

Con un grafo ya construido ya es posible realizar el calculo de los puntos más cercanos mediante la pulsación del botón “Encontrar Puntos Más Cercanos” en la pestaña de “Puntos” (véase apéndice G).

El primer algoritmo en ejecutarse es el de fuerza bruta, este se encarga de calcular la distancia entre todas las posibles combinaciones de pares de puntos, esto significa que la complejidad de este es de $O(N^2)$, ya que en el peor caso posible se verifican N veces $N-1$, en donde N representa el total de puntos. Esta multiplicación da un total de $\frac{N^2-N}{2}$, se divide entre dos ya que la distancia entre A y B es lo mismo que entre B y A. A final de cuentas, se toma únicamente el término dominante, el cual es N^2.

Al finalizar el algoritmo de fuerza bruta se ejecuta el algoritmo de divide y vencerás, este divide la lista de puntos en dos partes y recursivamente obtiene el par de puntos más cercanos de cada lado, el cálculo de estos puntos tiene una complejidad constante; ya que en el peor caso el algoritmo de fuerza bruta se realizará un total de seis veces. Después, se deben de obtener los puntos que se encuentren dentro del rango determinado por la menor distancia entre los dos lados, este paso tiene una complejidad de $O(N)$; ya que en el peor caso todos los puntos del grafo se encuentran en este rango. Estos puntos ahora pasan a ser ordenados con respecto a su coordenada y, el método de ordenamiento utilizado por C# tiene una complejidad de $O(N \log N)$. Al tener los puntos ordenados, el paso final consiste en calcular la distancia entre ellos; de esta manera, se obtiene o confirma el par de puntos más cercanos. Este paso utiliza unos ciclos que en un principio pueden parecer implementar la estrategia de fuerza bruta; sin embargo, la complejidad de estos es de $O(N)$, ya que, en el peor caso, todos los puntos únicamente verificaran un máximo de seis puntos para encontrar la mínima distancia.

De todas las complejidades algorítmicas expuestas, la complejidad que se toma es la de $O(N \log N)$, ya que es la dominante de todas las demás presentes; sin embargo, no hay que olvidar que el método hace dos llamadas recursivas y esto significa que las divisiones hechas a la lista original de puntos se harán un total de $\log N$ veces, lo cual significa que la complejidad algorítmica del método de divide y vencerás es de $O(N(\log N)^2)$.

En ambos, en el algoritmo de fuerza bruta y de divide y vencerás, se calcula el tiempo total de ejecución y la distancia entre el par de puntos más cercanos encontrado, estos datos son mostrados en pantalla cuando los algoritmos concluyen su ejecución.

El sistema debe de tener la capacidad de agregar múltiples tipos de agentes: con recorrido aleatorio, que recorran un árbol de recubrimiento mínimo y, presas y depredadoras. Primero, realice la implementación de los agentes con recorrido aleatorio.

Los agentes con recorrido aleatorio se generan al especificar la cantidad de agentes deseada y pulsar el botón “Crear Agentes” en la pestaña de “Aleatorio”. Aquí se ejecuta un ciclo que crea el número de agentes especificado en el “NumericUpDown”, dentro de este se crea un objeto de tipo “Agent” (véase apéndice A) con un vértice aleatorio como origen y se genera su recorrido aleatorio, este es almacenado en una

lista de puntos. Al final, el agente creado es agregado a lista de agentes con recorrido aleatorio utilizando el método “GenerateRandomPath” (véase apéndice A), este tiene una complejidad algorítmica de $O(N^2)$ ya que, en el peor caso, todos los vértices contendrán aristas a todos los vértices y, el método debe de procesarlas. Al crear todos los agentes necesarios, se activa un “Timer”, este será el encargado de realizar la animación de estos (véase apéndice G).

Dentro del evento “Tick” del “Timer” lo primero que se hace es una limpia del “Bitmap” utilizado para la animación. Después, cada agente en la lista de agentes es dibujado sobre el mapa de bits y se actualiza su índice que utiliza para recorrer su lista de puntos, la cual representa el recorrido que el agente realizará. La simulación termina cuando todos los agentes llegan al último índice de su lista de puntos.

Al final de la simulación, se utiliza el método “UpdateAgentWithRandomPathDetails” para cargar los datos de todos los agentes en un “TreeView” (véase apéndice G). Cada agente muestra su ID, la cantidad de vértices por los que paso y la distancia total que recorrió. Dentro de este método se identifica cual fue el agente que más vértices recorrió. Se compara el tamaño de la lista de vértices visitados del agente actual con la del agente en el índice del agente con la mayor cantidad de vértices visitados, en caso de que la lista sea mayor, se reemplaza el índice del vértice con la mayor cantidad de vértices visitados por el del agente actual. En caso de que la lista de vértices visitados sea igual, entonces el agente que haya recorrido la mayor distancia sale ganador. Este método tiene una complejidad algorítmica de $O(N)$ ya que, siempre se realiza esta comparación con todos los agentes en la lista global de agentes.

Después de realizar la implementación de los agentes con recorrido aleatorio, implementé los agentes que recorren un árbol de recubrimiento mínimo. Para poder generar agentes de este tipo es necesario primero crear un árbol de recubrimiento mínimo, esto se hace pulsando el botón “Generar ARM” y especificando un vértice origen; ya que al pulsar este botón se genera un árbol utilizando los algoritmos de Prim y de Kruskal.

El primer algoritmo que se ejecuta es el de Kruskal, este es dibujado con ayuda del método “DrawMSTWithKruskal” (véase apéndice G) la cual recupera las aristas pertenecientes al árbol de recubrimiento mínimo mediante el uso del método “BuildMSTWithKruskal” (véase apéndice G).

En este último, se inicializa el conjunto candidatos con todas las aristas del grafo, el conjunto prometedor vacío, y los componentes conexos con todos los vértices del grafo. La inicialización de los componentes conexos cuenta con una complejidad de $O(N)$. Ya que el número inicial de componentes conexos es igual al número de vértices, o sea, N.

Antes que nada, se realiza un ordenamiento del conjunto candidatos, el cual cuenta con una complejidad de $O(N^2)$. Esto se debe a que el método utilizado para ordenar los elementos es “QuickSort”, el cual en su peor caso presenta dicha complejidad. El ordenamiento se hace para que de esta manera quede ordenado por el peso de las aristas.

Ahora, mientras número de elementos en el conjunto prometedor no sea igual al número de vértices menos uno, se selecciona una arista, el cual siempre será el primer elemento del conjunto candidatos, ya que como se encuentra ordenado, el primer elemento siempre representara la siguiente arista con el menor peso. Después, se busca en la lista de componentes conexos los componentes en donde se encuentran el origen y el destino de la arista seleccionada, si estas dos son diferentes entre sí, la arista seleccionada se agrega al conjunto prometedor y los componentes conexos del origen y del destino se hacen uno mismo. Al final, se elimina del conjunto candidatos la arista seleccionada y se verifica si el conjunto candidatos aún tiene elementos, en caso de que, si tenga, se realiza una nueva iteración. Este ciclo se repite $O(N^2)$. Esto pasa cuando se utilizan todas las aristas del conjunto candidatos. Dentro del ciclo, la mayoría de las operaciones realizadas presentan una complejidad constante; sin embargo, se realizan dos búsquedas las cuales cuentan con una complejidad de $O(N)$. Por lo tanto, la complejidad del ciclo puede expresarse como $O(N^3)$. En base a esto, se puede concluir que el cálculo del árbol de recubrimiento mínimo por medio del algoritmo de Kruskal cuenta con una complejidad de $O(N^3)$.

Al final del cálculo del árbol de recubrimiento mínimo, se genera un nuevo grafo de nombre “kruskalMST” con únicamente las aristas del conjunto solución y, dependiendo del número de componentes conexos al final, se especifica el número de subgrafos encontrados. Para terminar, se retorna el conjunto solución, aquí es donde la función “DrawMSTWithKruskal” lo utiliza para dibujar sobre un mapa de bits las aristas seleccionadas, después se actualiza el peso total del árbol de recubrimiento mínimo con Kruskal, y se muestran en pantalla, a través de un “FormMSTSolutionSet” (véase apéndice H), las aristas del conjunto solución.

A continuación, se presenta el algoritmo de Kruskal utilizado:

```
ARM Kruskal(){
    candidatos = aristas de G
    prometedor = vacío
    CC = vértices de G
    ordenar candidatos
    mientras(tamaño de prometedor != número de vértices de G - 1){
        arista{v_origen, v_destino} = candidatos[0];
        CC_origen = buscaCCde(v_origen);
        CC_destino = buscaCCde(v_destino);
        if(CC_origen != CC_destino)
            prometedor = prometedor U arista
            fusionaCC(CC_origen,CC_destino)
         candidatos = candidatos – arista
         if(tamaño de candidatos = 0)
              salir
    }
    devolver prometedor
}
```

Al finalizar el algoritmo de Kruskal, se ejecuta el algoritmo de Prim, partiendo del vértice especificado previamente. Este es dibujado con ayuda de la función “DrawMSTWithPrim” (véase apéndice G) la cual recupera las aristas pertenecientes

al árbol de recubrimiento mínimo mediante la función “BuildMSTWithPrim” (véase apéndice G).

Lo primero que se hace es la inicialización del conjunto candidatos con las aristas del vértice origen, la inicialización de una variable que llevará el conteo del número de subgrafos, y la inicialización de los vértices del árbol de recubrimiento mínimo con el vértice origen. La inicialización del conjunto candidatos cuenta con una complejidad de $O(N)$. Esto se debe a qué en el peor caso posible, el vértice origen contará con aristas hacia todos los demás, lo cual es igual a $N-1$ aristas.

Antes de pasar al ciclo, se ordena el conjunto candidatos dependiendo del peso de las aristas, una vez hecho esto, comienza el bucle. El ordenamiento de los candidatos cuenta con una complejidad de $O(N)$. Esto se debe a que el método utilizado para ordenar los elementos es “QuickSort”, el cual en su peor caso presenta dicha complejidad.

Dentro del ciclo, primero se verifica si el conjunto candidatos se encuentra vacío, esta situación puede generarse por dos distintas causas, o ya se terminaron todas las aristas del grafo, o el grafo es no conexo y aún quedan vértices que procesar. En la primera opción, la función termina; sin embargo, en la segunda opción, se toma un vértice de los restantes, se agrega a lista de vértices del árbol de recubrimiento mínimo, se agregan sus aristas de manera ordenada al conjunto candidatos y se incrementa una unidad la variable con el número de subgrafos. En el caso contrario en el que el conjunto candidatos aun no es cero, entonces se selecciona la primera arista de él. Después se verifica si el origen o destino de esta arista seleccionada no se encuentra en la lista de vértices del árbol de recubrimiento mínimo, en caso de que así sea, esta arista es agregada al conjunto prometedor y después se agrega el vértice origen o destino, esto depende de cual aún no se encuentra en la lista de vértices del árbol de recubrimiento mínimo. Al final, se elimina la arista seleccionada del conjunto candidatos y se realiza una nueva iteración del bucle. Este ciclo termina cuando el número de aristas del conjunto prometedor es igual al número de vértices menos uno, o se terminen las aristas del grafo. El ciclo se repite $O(N^2)$. Esto pasa cuando se utilizan todas las aristas del conjunto candidatos. Dentro del ciclo, la mayoría de las operaciones realizadas presentan una complejidad constante; sin embargo, se realiza dos búsquedas en la lista de vértices del árbol de recubrimiento mínimo y una búsqueda binaria en el conjunto candidatos, con complejidades $O(N)$ y $O(N \log N)$. Por lo tanto, la complejidad del ciclo puede expresarse como $O(N^3)$. En base a esto, se puede concluir que el cálculo del árbol de recubrimiento mínimo por medio del algoritmo de Prim cuenta con una complejidad de $O(N^3)$.

Al final del algoritmo, se genera un nuevo grafo de nombre “primMST” con únicamente las aristas del conjunto solución y se especifica el número de subgrafos encontrados. Lo último por realizar es retornar el conjunto candidato que ahora es el conjunto solución. Este conjunto será utilizado por la función “DrawMSTWithPrim” para dibujar sus aristas sobre un mapa de bits, después se actualiza el peso total del árbol de recubrimiento mínimo con Prim, y se muestran en pantalla, a través de un “FormMSTSolutionSet” (véase apéndice H), las aristas del conjunto solución.

A continuación, se presenta el algoritmo de Prim utilizado:

```
ARM Prim(Vertice v_inicial){
    candidatos = aristas de v_inicial
    prometedor = vacio
    v_prim = vértice inicial
    num_subgrafos = 1
    ordenar candidatos
    mientras(tamaño de prometedor != número de vértices de G - 1){
        if(tamaño de candidatos = 0)
            v_restantes = vértices de G – v_prim
            if(tamaño de v_restantes = 0)
                salir
            num_subgrafos++
            candidatos = candidatos U aristas de v_restantes[0]
            v_prim = v_prim U v_restantes[0]
        arista{v_origen, v_destino} = candidatos[0];
        if(v_prim no contiene v_origen y v_destino){
            prometedor = prometedor U arista
            if (v_origen pertenece a v_prim)
                v_prim = v_prim U v_destino
                candidatos = candidatos U aristas de v_destino
            else
                v_prim = v_prim U v_origen
                candidatos = candidatos U aristas de v_origen
        }
        candidatos = candidatos - arista
    }
    devolver prometedor
}
```

Una vez que los arboles son creados, ya es posible agregar agentes para que los recorran. Estos pueden ser agregados especificando la cantidad, el algoritmo y, si los vértices origen serán aleatorios o no, para después presionar el botón “Crear Agentes” en la pestaña de “Árboles”.

El recorrido del agente es generado por la función “GenerateMSTPath” (véase apéndice A), esta función por su parte manda llamar a otra función recursiva, esta de nombre “GetMSTEdgesPath” (véase apéndice A). Esta última recibe como parámetros un vértice actual y una lista de vértices visitados, “GenerateMSTPath” le envía como parámetros el vértice origen elegido para el agente, y una lista que contiene este vértice. “GetMSTEdgesPath” recorre la lista de aristas del vértice proporcionado, para cada una verifica si su destino no se encuentra en la lista de vértices visitados, en caso de que así sea se agrega el vértice destino a la lista de visitados, el arista a la lista de aristas visitados (esta es utilizada para detallar el recorrido del agente), los puntos de la arista se concatenan a los puntos existentes del recorrido y se realiza una llamada recursiva a la función ahora con el vértice destino de la arista. En caso de que el destino se encuentre en la lista de vértices visitados, entonces se asume que se trata de una arista para retornar a un vértice anterior. Al final de esto se verifica si se encontró una arista para retornar o no, si sí se agrega esta arista a la lista de aristas visitadas y se concatenan sus puntos a la lista de los puntos del recorrido. Al final, para ambos casos se retornan los puntos acumulados en la iteración.

Dependiendo del algoritmo que el usuario haya especificado para los agentes se determinarán las aristas que utilizarán. De igual manera, en caso de que se haya especificado que los vértices orígenes serán aleatorios, el programa le asignara un vértice origen de su elección a todos los agentes generados; de lo contrario, el usuario tiene que elegir, uno por uno, los vértices origen de todos los agentes.

Una vez que todos los agentes cuentan con sus respectivos recorridos, se activa un "Timer". Dentro del evento "Tick", lo primero que se hace es una limpia del "Bitmap" utilizado para la animación. Después, cada agente es dibujado sobre el mapa de bits y se actualiza su índice que utiliza para recorrer su lista de puntos, la cual representa el recorrido que el agente realizará. Este comenzará en un vértice y terminará en el mismo. La simulación termina cuando todos los agentes llegan al último índice de su lista de puntos.

Al finalizar la programación de los agentes que recorren un árbol de recubrimiento mínimo, era tiempo de implementar los agentes presas y depredadores. Para poder iniciar una simulación con este tipo de agentes, lo primero que se debe de hacer es agregar presas al grafo, esto se hace especificando la cantidad de presas deseadas y presionando el botón "Agregar Presas" en la pestaña de "Presas". Al presionar este botón se despliega una ventana en donde el usuario especifica el origen y destino de la presa actual, esto se repite en cada uno de los agentes agregados. Aquí, se genera el camino entre estos dos vértices.

La generación del camino se lleva a cabo mediante el uso de un método perteneciente a la clase "Prey" de nombre "GeneratePath"; por su parte, este método utiliza otro de nombre "GetShortestPathsWithDijkstra" para encontrar el camino de menor peso del origen al destino utilizando el algoritmo de Dijkstra (véase apéndice G).

Primero se realiza la inicialización de las variables, entre las cuales se encuentra una lista de objetos de tipo "DijkstraElement" (véase apéndice J), la inicialización de esta lista cuenta con una complejidad algorítmica de $O(N)$, esto se debe a que el número de objetos "DijkstraElement" insertados en la lista es igual al número de vértices en el grafo.

Después, al "DijkstraElement" que contiene el vértice origen de la presa, se le asigna una distancia acumulada de cero y se especifica el "DijkstraElement" actual como el proveniente; ya que para llegar al "DijkstraElement" no es necesario hacer nada porque ya nos encontramos ahí. Como los valores del "DijkstraElement" cambiaron, es necesario hacer una reinserción del objeto en la lista para determinar su nueva posición, esta relocalización se hace mediante el método "RelocateDijkstraElement", el cual realiza una búsqueda binaria e inserta el elemento en su nuevo índice (véase apéndice J). Este proceso cuenta con una complejidad algorítmica de $O(\log(N))$, ya que el peor caso de la búsqueda binaria presenta esta complejidad.

En la última parte del método se realiza un ciclo para encontrar el camino del vértice origen de la presa al vértice destino. Este ciclo puede terminar por tres distintas razones:

1. Todos los objetos "DijkstraElement" de la lista ya son definitivos.
2. El "DijkstraElement" que contiene el vértice destino ha pasado a ser definitivo.

3. La distancia acumulada del “DijkstraElement” hecho definitivo en la iteración actual tiene un valor infinito.

En cada iteración del ciclo, un elemento de la lista de objetos “DijkstraElement” se hace definitivo y este es procesado mediante el método “UpdateDijkstraElements” (véase apéndice J).

“UpdateDijkstraElements” tiene un ciclo que recorre todas las aristas del vértice contenido en el “DijkstraElement” definitivo; por lo tanto, este ciclo puede repetirse hasta un total de $N-1$ veces, porque en el peor caso, el vértice cuenta con aristas a todos los demás. Lo primero que se hace en cada iteración es encontrar el “DijkstraElement” que contiene al vértice destino de la arista actual, esto se hace mediante una búsqueda lineal la cual cuenta con una complejidad de $O(N)$. En caso de que la distancia acumulada del “DijkstraElement” definitivo más el peso de la arista actual sea menor a la distancia acumulada del “DijkstraElement” que contiene al vértice destino de la arista, entonces, en este último “DijkstraElement”, la distancia acumulada pasa a ser sustituida por la suma realizada previamente, el vértice proveniente pasa a ser el “DijkstraElement” definitivo y, por último, se relocaliza mediante el método “RelocateDijkstraElement”, el cual, como se mencionó previamente, cuenta con una complejidad algorítmica de $O(\log(N))$. En base a esto, podemos determinar que la complejidad del método “UpdateDijkstraElements” es de $O(N^2)$.

Cuando el método “UpdateDijkstraElements” finaliza, también lo hace la iteración del ciclo del método “GetShortestPathsWithDijkstra”. Por su parte, este ciclo tiene una complejidad de $O(N^3)$. Esto se debe a que, en el peor caso, el ciclo terminará cuando todos los objetos “DijkstraElement” de la lista sean definitivos, lo cual requiere de un total de N iteraciones y, que en cada iteración se manda llamar al método “UpdateDijkstraElements” el cual tiene una complejidad algorítmica de $O(N^2)$. Con estos datos, se puede concluir que el método “GetShortestPathsWithDijkstra” tiene una complejidad de $O(N^3)$, en donde se usan las complejidades de: la inicialización de la lista de objetos “DijkstraElement” ($O(N)$), la relocalización del “DijkstraElement” con el vértice origen ($O(\log(N))$) y, el ciclo ($O(N^3)$). Cuando este ciclo termina, el camino de la presa ha sido generado.

Después de agregar presas, ahora se deben de agregar depredadores. De igual manera que con las presas, se especifica la cantidad de depredadores deseados y se presiona el botón “Agregar Depredadores” en la pestaña de “Presas”. Este botón es el encargado de generar los depredadores deseados en vértices aleatorios.

Al final, para comenzar la simulación basta con presionar el botón de “Iniciar”. Aquí se generan todos los caminos de los depredadores. Para esto se utiliza un método perteneciente a la clase “Predator” de nombre “GeneratePath”; por su parte, este método inicializa una lista vértices visitados y, una pila de vértices, en donde en ambas inserta el vértice origen del depredador. Después, manda llamar a otro método de nombre “DepthFirstSearch” que utiliza la lista y la pila para recorrer el grafo por profundidad, este último método es recursivo (véase apéndice G).

El método “DepthFirstSearch” inicializa un vértice actual con el último elemento apilado en la pila de vértices, después, para cada arista del vértice actual se verifica si el destino de esta se encuentra en la lista de vértices visitados o no. En caso de que no se encuentre en la lista, se inserta la arista en el camino del depredador y, el vértice destino de la arista actual es apilado en la pila de vértices e insertado en la lista de vértices visitados para después mandar llamar recursivamente al método “DepthFirstSearch”, el cual ahora repetirá el mismo proceso con el vértice destino de la arista. En caso de que el vértice si se encuentre en la lista, entonces se desapila un elemento de la pila de vértices y termina el método.

Este método se asegura de recorrer todos los vértices del grafo; sin embargo, también cuenta con los mecanismos necesarios para trazar un camino lógico; es decir, que no se produzcan cambios repentinos de un vértice a otro, que el camino generado le permita al depredador moverse en el grafo de manera visualmente progresiva sin ningún brinco de un vértice a otro. Esto lo hace cuando se determina que el vértice destino de la arista actual si se encuentra en la lista de vértices visitados, en este caso, se verifica si el vértice rechazado es el vértice del que se proviene, en caso de que así sea, se almacena la arista en una pila. En esta pila se pueden apilar las aristas necesarias para trazar un camino de regreso. Estas aristas son desapiladas cuando al terminar de procesar el vértice actual se vuelve a hacer una llamada al método “DepthFirstSearch”, de esta manera se comprueba que aun no se han visitado todos los vértices del grafo y; por consecuencia, hay que trazar un camino de regreso.

El método “DepthFirstSearch” es llamado un total de N veces, una por cada vértice perteneciente al subgrafo con el que se trabaja. Este método, como se mencionó previamente, recorre todas las aristas del vértice actual, lo cual establece que el ciclo que realiza este recorrido itera un total de $N-1$ veces, ya que en el peor caso existe una arista a todos los demás vértices del grafo. Dentro de este ciclo, la mayoría de las operaciones que se realizan presentan una complejidad constante; sin embargo, este no es el caso cuando se insertan al camino del depredador las aristas apiladas para trazar un camino de vuelta. Estas aristas son insertadas en un ciclo máximo iterará un total de $N-2$ veces, esto se debe a que, en el peor caso, se tomará una arista del vértice origen y en base a esta se podrá trazar recursivamente un camino a todos los demás vértices del subgrafo, excepto a uno, el cual únicamente puede ser accedido mediante otra arista del vértice origen. Con estos datos, podemos determinar que el ciclo del método “DepthFirstSearch” cuenta con una complejidad de $O(N^2)$ y; por lo tanto, el cálculo del recorrido de los depredadores tiene una complejidad algorítmica de $O(N^3)$.

Una vez que las presas y depredadores cuentan con sus respectivos recorridos, se activa un “Timer”. Dentro del evento “Tick”, lo primero que se hace es una limpia del “Bitmap” utilizado para la animación. Después, las presas y los depredadores avanzan sobre su arista actual, esta es obtenida de su camino, el cual es una cola de aristas. Cuando se termina de recorrer los puntos de la arista actual, se desencola la nueva arista actual del camino y, el proceso se repite. Esto aplica para ambos, las presas y los depredadores. El objetivo de una presa consiste en llegar al vértice destino; sin embargo, al recorrer las aristas de su camino debe de asegurarse que estas se encuentren libres de depredadores. Una presa permanece en el principio de la arista

actual hasta que esta sea desocupada por los depredadores, esta verificación es posible gracias a un contador con el que cada objeto de tipo “Edge” cuenta (véase apéndice E), cada vez que un depredador entra a una arista este le suma una unidad al contador y, cuando cambian de arista le restan una unidad; de esta manera, cuando el contador se encuentra en cero, la presa pasa a recorrer la arista. Sin embargo, puede darse la situación en la que la presa se encuentra recorriendo una arista y un depredador entra a ella, en estos casos los depredadores pueden consumir a la presa y terminar la simulación.

Hay de dos casos, un depredador puede recorrer la arista en la misma dirección que una presa o en dirección inversa; en el primer caso la presa hace caso omiso del depredador y continua a su destino; en el segundo caso la presa invierte su dirección y se dirige al origen de la arista, para de esta manera, evitar ser consumida. El único caso en el que una presa puede ser consumida es cuando queda atrapada entre dos depredadores recorriendo la arista en direcciones inversas. Cuando una presa llega a su destino, si el usuario así lo desea se puede asignar un nuevo vértice destino a la presa; de lo contrario, la presa es eliminada de la lista de presas y la simulación continua con las presas restantes.

La simulación puede terminar por dos situaciones distintas, la primera de ellas consiste en que todas las presas sean consumidas por depredadores, y, la segunda en que el usuario decida no asignarles un nuevo destino a todas las presas.

Con todos los tipos de agentes implementados, el ultimo requerimiento especifica que debe de ser posible intentar la creación de un árbol en donde todas sus hojas se encuentren al mismo nivel; sin embargo, la creación de este debe de ser realizada recorriendo el grafo en amplitud. La generación del árbol se intenta con todos los vértices y concluye cuando se encuentre uno.

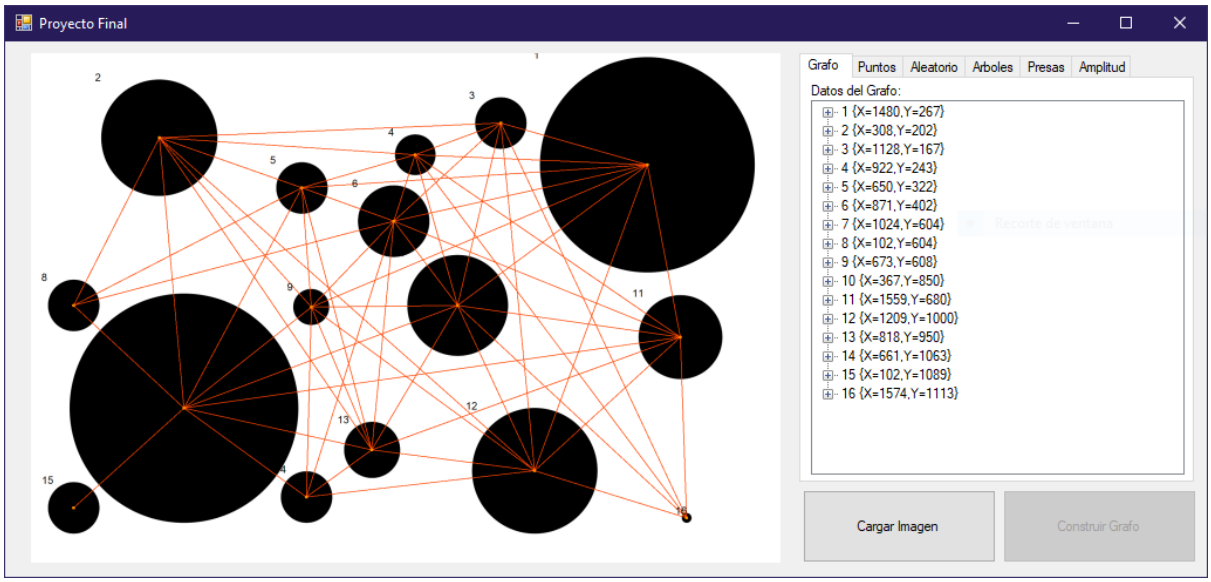
Para la generación del árbol basta con presionar el botón “Generar Árbol en Amplitud” en la pestaña de “Amplitud” (véase apéndice G), aquí se ejecuta el método “BreadthFirstSearch”. Este método recibe como parámetro un vértice origen y retorna verdadero o falso, en caso de que exista un árbol o no. Se inicializa una cola de vértices y una lista de vértices visitados, ambas con el vértice origen.

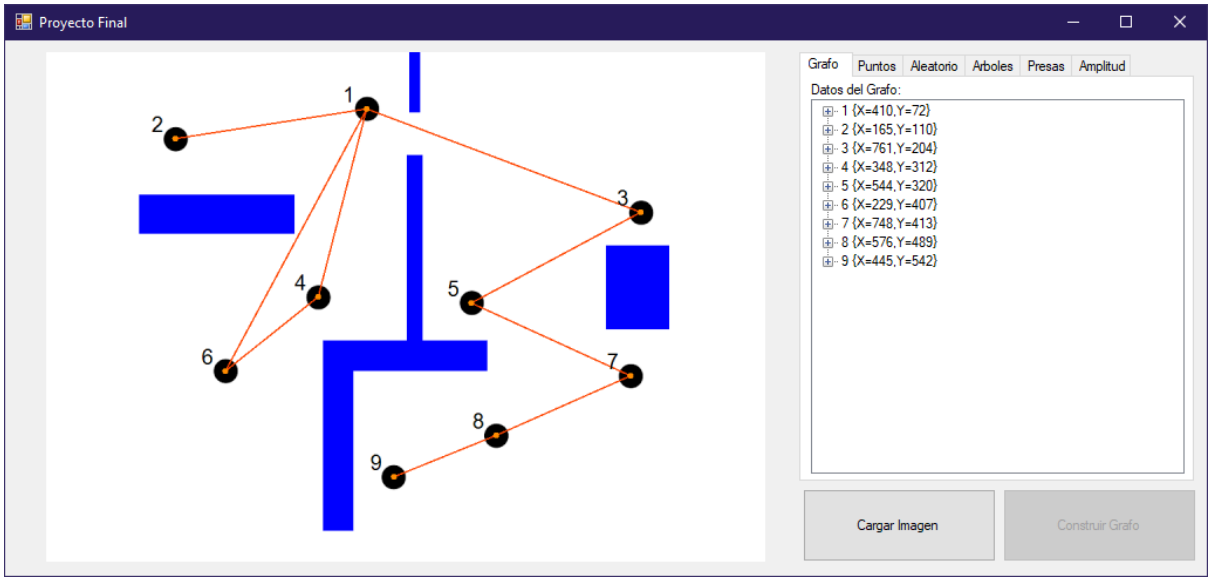
Primero, se ejecuta un ciclo, dentro de este, se desencola un vértice y este se convierte en el vértice actual, después, para cada arista del vértice origen, se verifica si su destino aún no ha sido visitado, en caso de que no, el vértice destino pasa a ser encolado y agregad a la lista de vértices visitados; además, ambos puntos, el origen y el destino de la arista son agregados a una lista que posteriormente será utilizada pues dibujar el árbol, esto en caso de que si exista. De lo contrario, en caso de que el vértice destino ya haya sido visitado, se ignora la arista. Aquí se lleva a cabo el cálculo de los nuevos nodos generados del árbol, esto se hace incrementando una unidad el valor de una variable cada vez que se agregue uno. Este cálculo se hace para llevar un control del nivel del árbol y comprobar si todas las hojas de este tienen el mismo. El ciclo exterior, el que itera sobre la cola, se repite un total de N veces, ya que en el peor caso se procesarán todos los vértices del grafo. Y el ciclo interior, el que itera sobre las aristas del vértice actual, se repite un total de $N - 1$ veces; ya que, en el peor caso, el vértice presenta aristas a todos los demás.

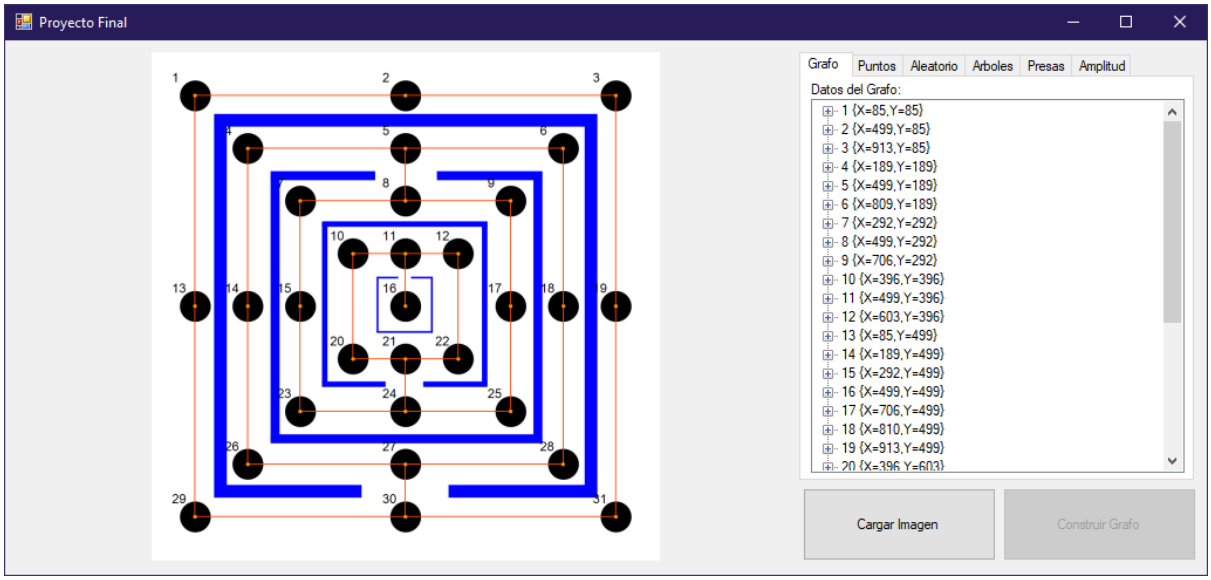
Después de procesar todas las aristas del vértice origen, en caso de que este vértice no sea el primer nodo analizado del nivel actual, entonces se verifica si el vértice anterior agregó nodos o no, está adición debe de coincidir con la del vértice actual; es decir, si en el vértice pasado se agregó un nuevo nivel, en el actual debió de haber sucedido lo mismo. En caso de que no coincida, se retorna falso y se intenta nuevamente la generación del árbol utilizando un vértice origen diferente. Al final del ciclo, si ya se verificaron todos los nodos del nivel actual, se pasan a crear los nodos del siguiente nivel para posteriormente ser verificados, esto se realiza con un ciclo que itera un máximo de $N - 1$ veces, considerando que, en el peor caso, se agregan todos los vértices del grafo al árbol. Una vez que la cola de vértices se encuentra vacía, el método termina y se retorna verdadero; en otras palabras, se encontró el árbol. En base a todo lo previamente establecido, el método "BreadthFirstSearch" cuenta con una complejidad algorítmica de $O(N^2)$, ya que este es el término dominante.

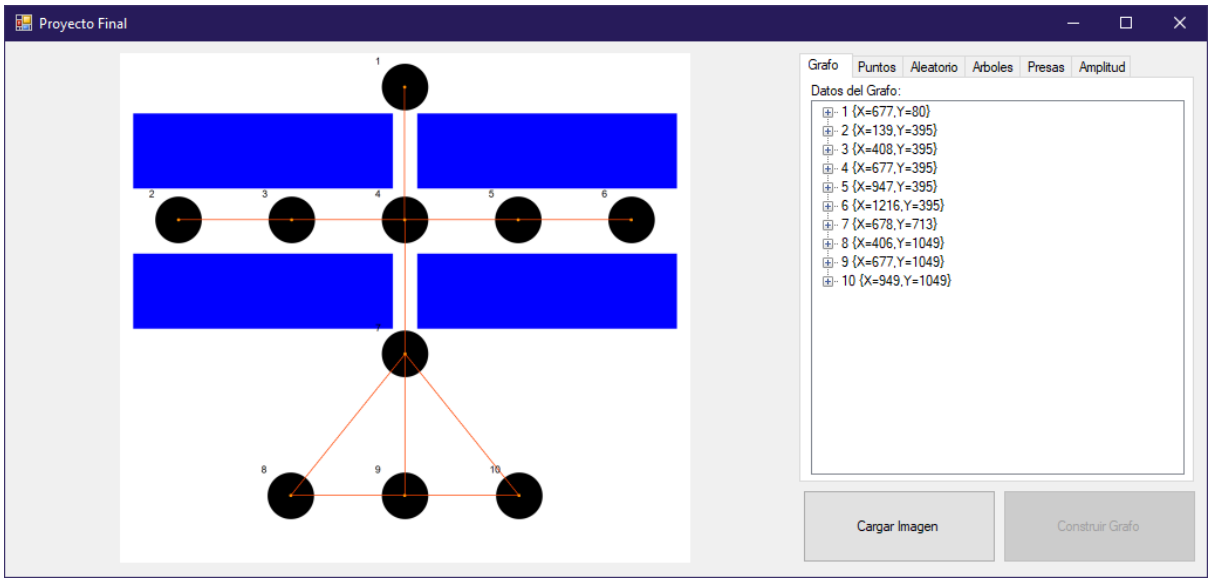
Pruebas y resultados

Pestaña de "Grafo"

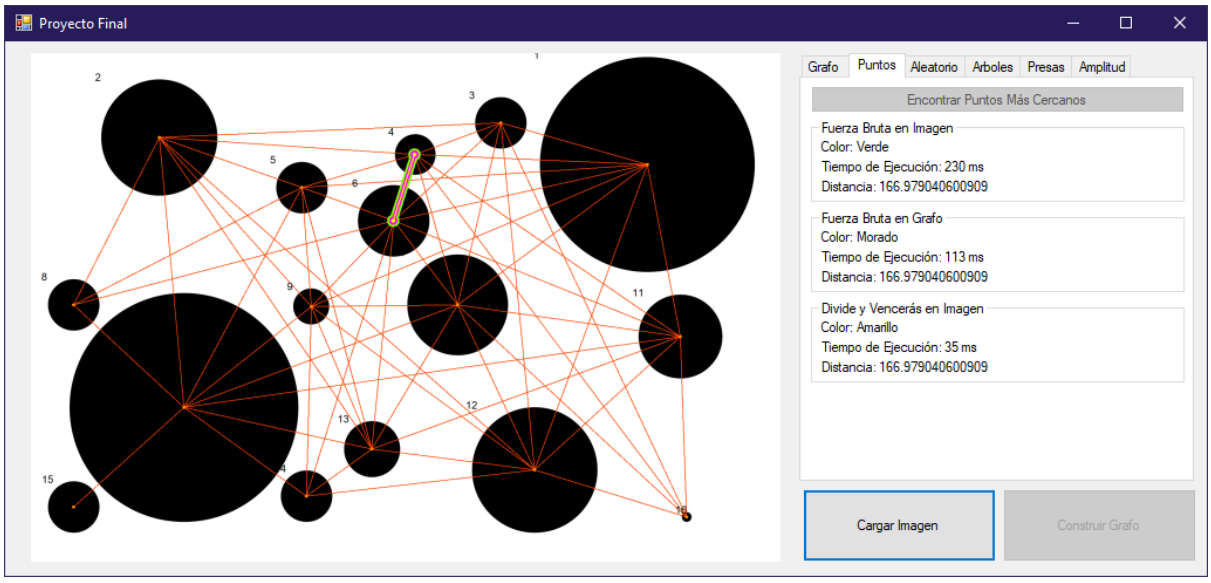


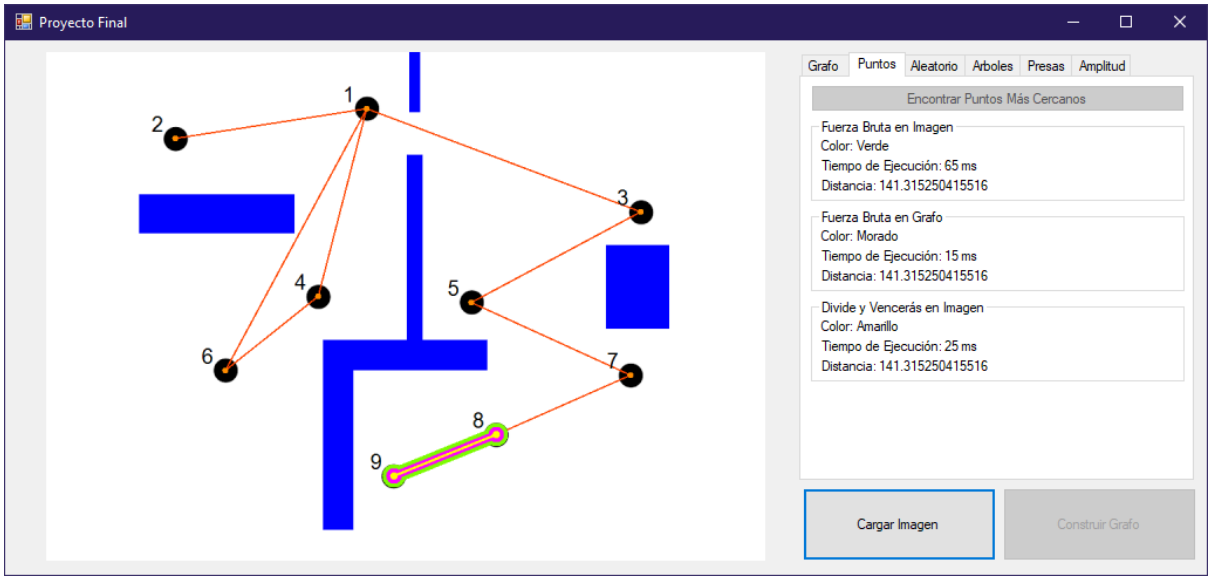


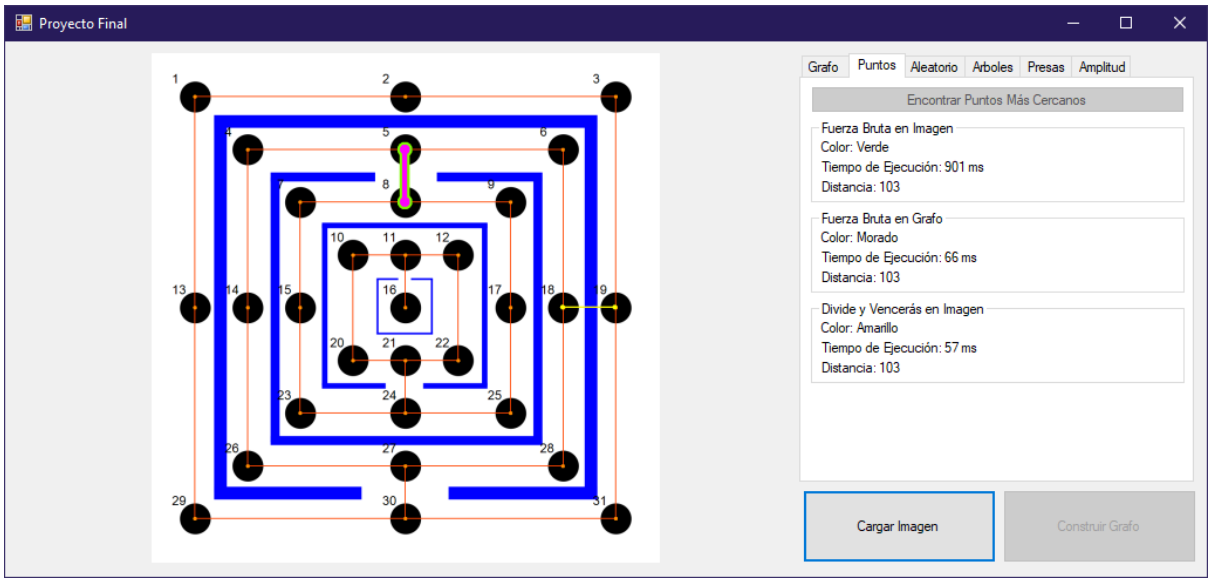


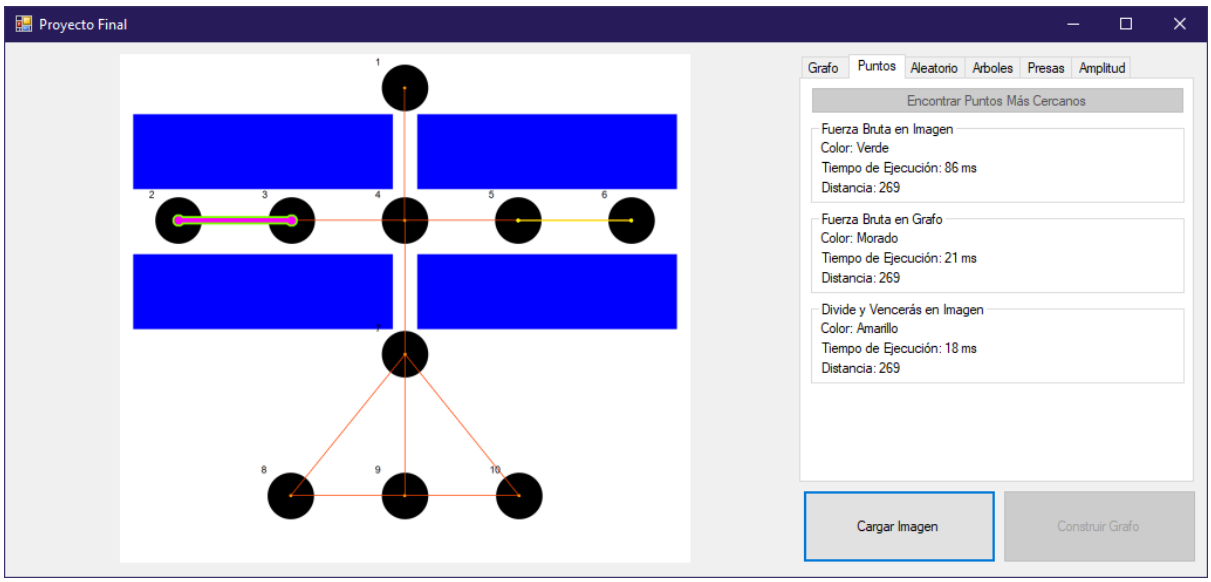


Pestaña de “Puntos”

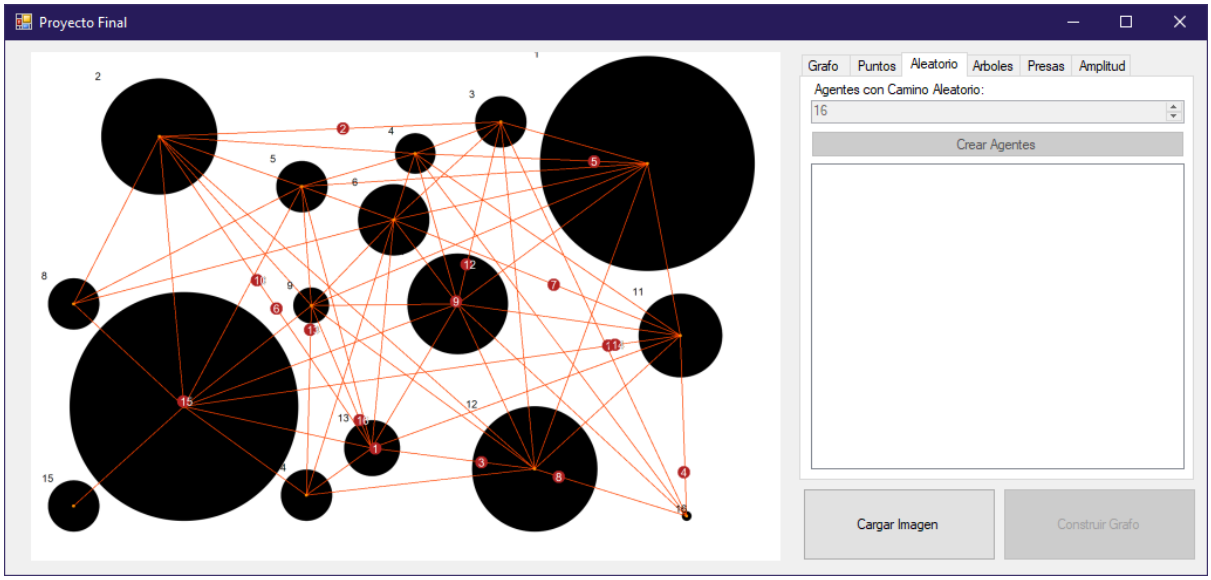


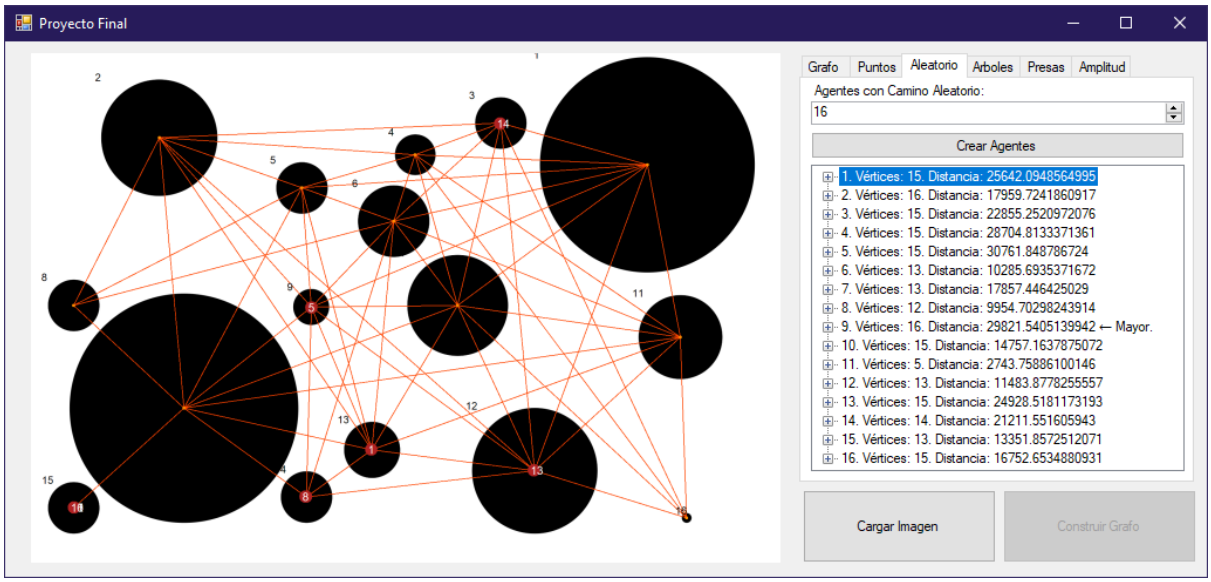

Proyecto Final
Grafo
Puntos
Aleatorio
Arboles
Presas
Amplitud
Encontrar Puntos Más Cercanos
Fuerza Bruta en Imagen
Color: Verde
Tiempo de Ejecución: 65 ms
Distancia: 141.315250415516
Fuerza Bruta en Grafo
Color: Morado
Tiempo de Ejecución: 15 ms
Distancia: 141.315250415516
Divide y Vencerás en Imagen
Color: Amarillo
Tiempo de Ejecución: 25 ms
Distancia: 141.315250415516
Cargar Imagen
Construir Grafo

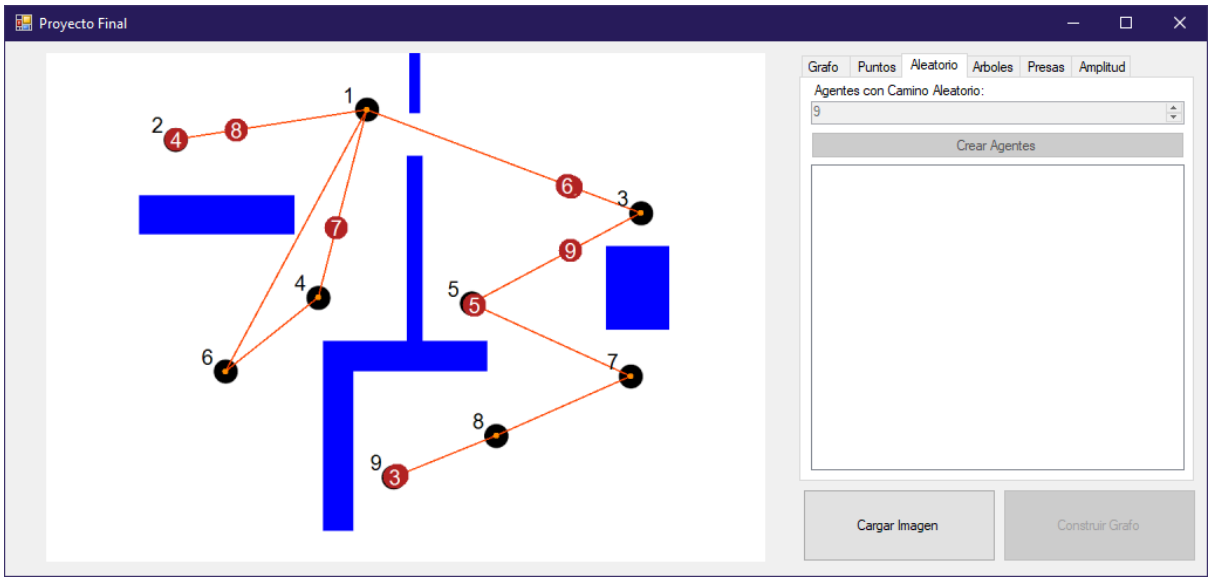

Proyecto Final
Grafo
Puntos
Aleatorio
Arboles
Presas
Amplitud
Encontrar Puntos Más Cercanos
Fuerza Bruta en Imagen
Color: Verde
Tiempo de Ejecución: 901 ms
Distancia: 103
Fuerza Bruta en Grafo
Color: Morado
Tiempo de Ejecución: 66 ms
Distancia: 103
Divide y Vencerás en Imagen
Color: Amarillo
Tiempo de Ejecución: 57 ms
Distancia: 103
Cargar Imagen
Construir Grafo

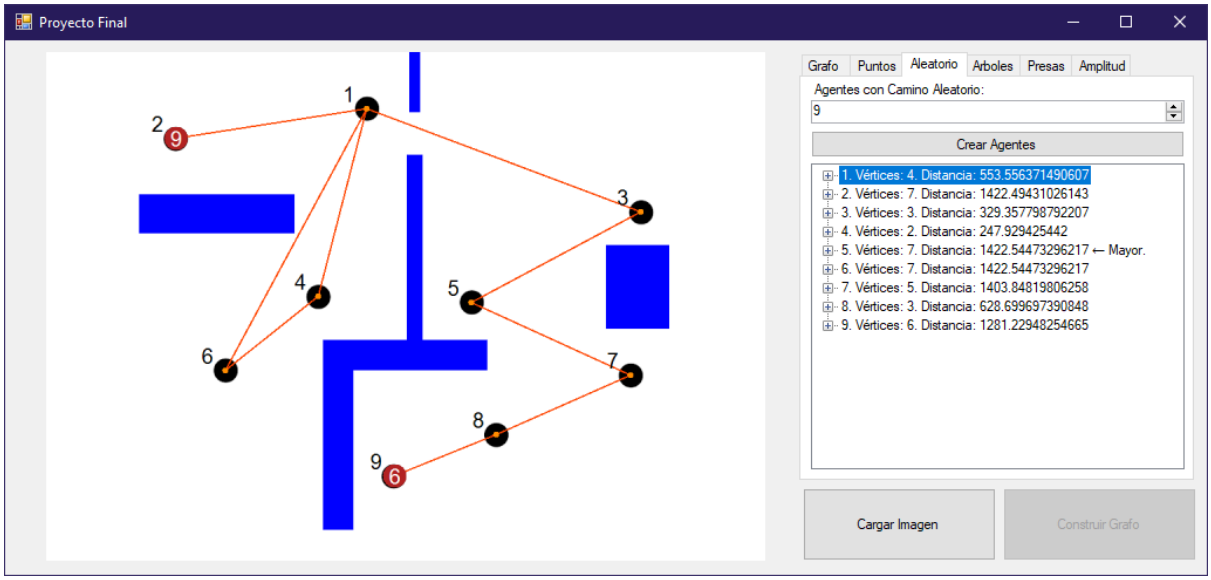

Proyecto Final
Grafo
Puntos
Aleatorio
Arboles
Presas
Amplitud
Encontrar Puntos Más Cercanos
Fuerza Bruta en Imagen
Color: Verde
Tiempo de Ejecución: 86 ms
Distancia: 269
Fuerza Bruta en Grafo
Color: Morado
Tiempo de Ejecución: 21 ms
Distancia: 269
Divide y Vencerás en Imagen
Color: Amarillo
Tiempo de Ejecución: 18 ms
Distancia: 269
Cargar Imagen
Construir Grafo

Pestaña de “Aleatorio”

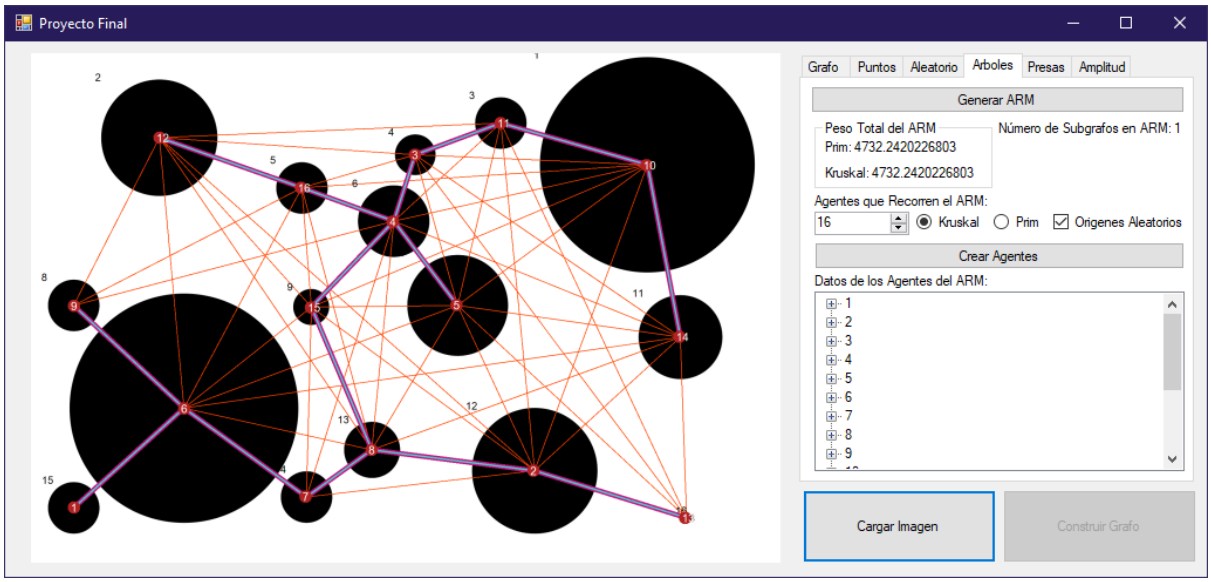


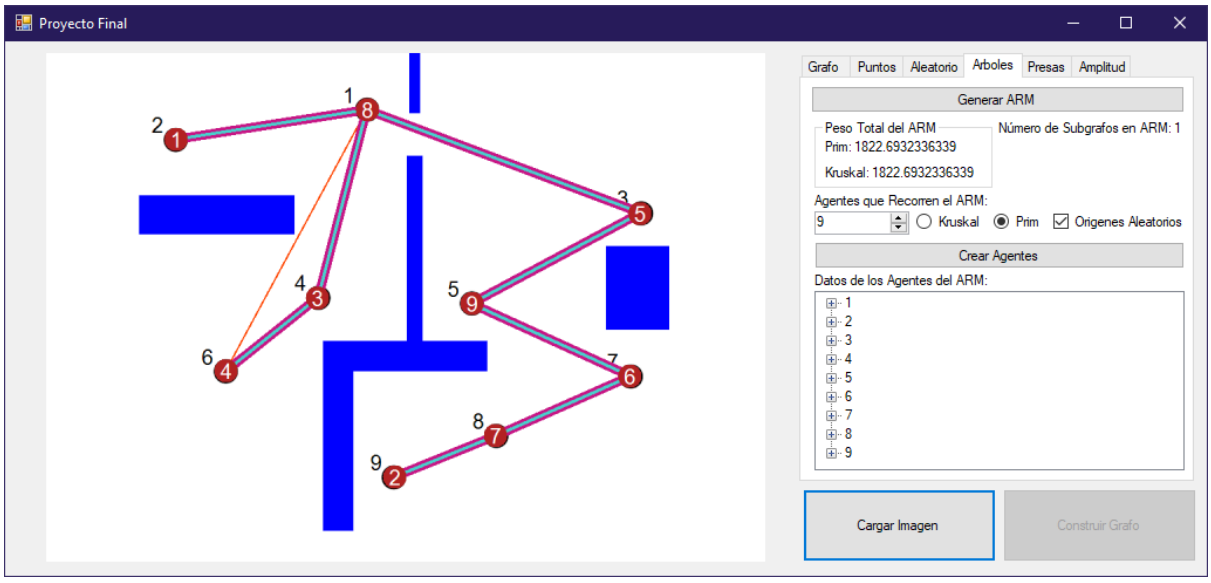


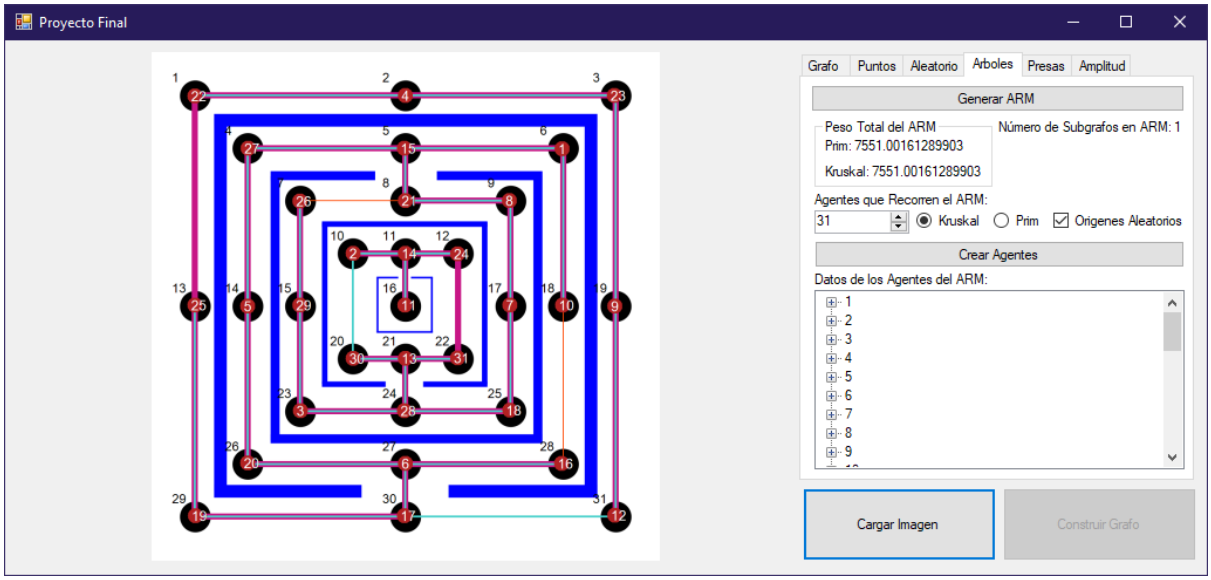


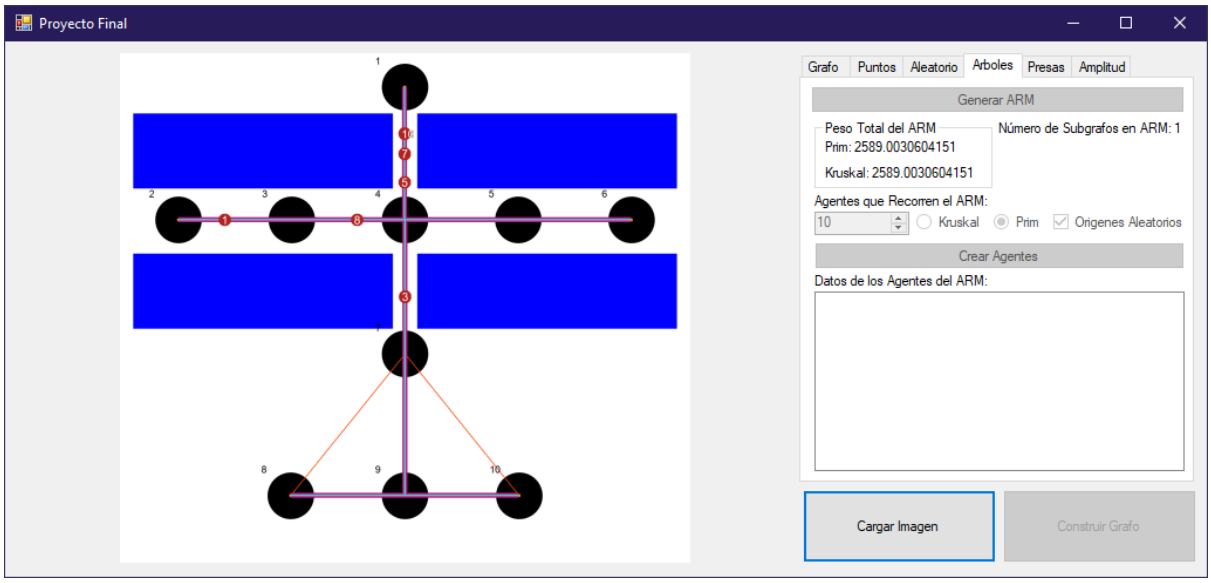


Pestaña de “Árboles”

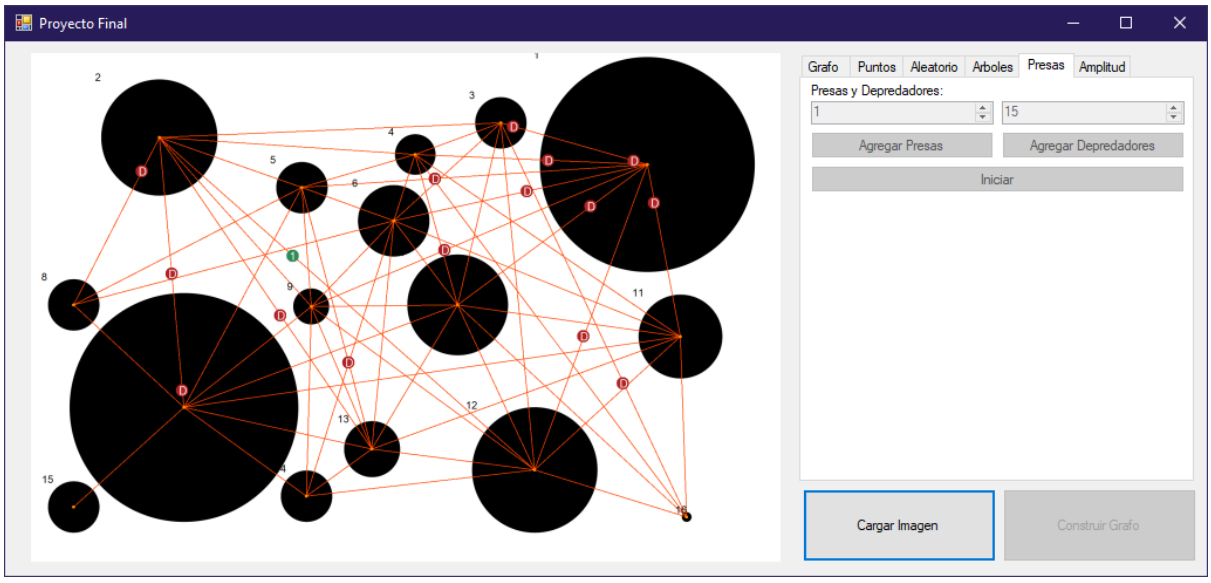


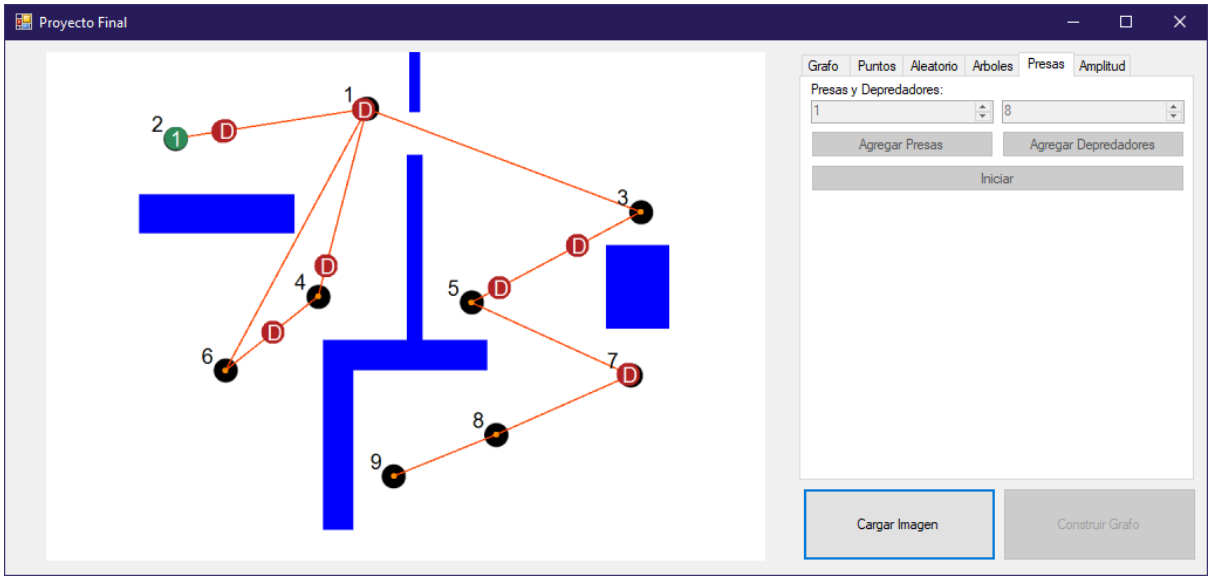


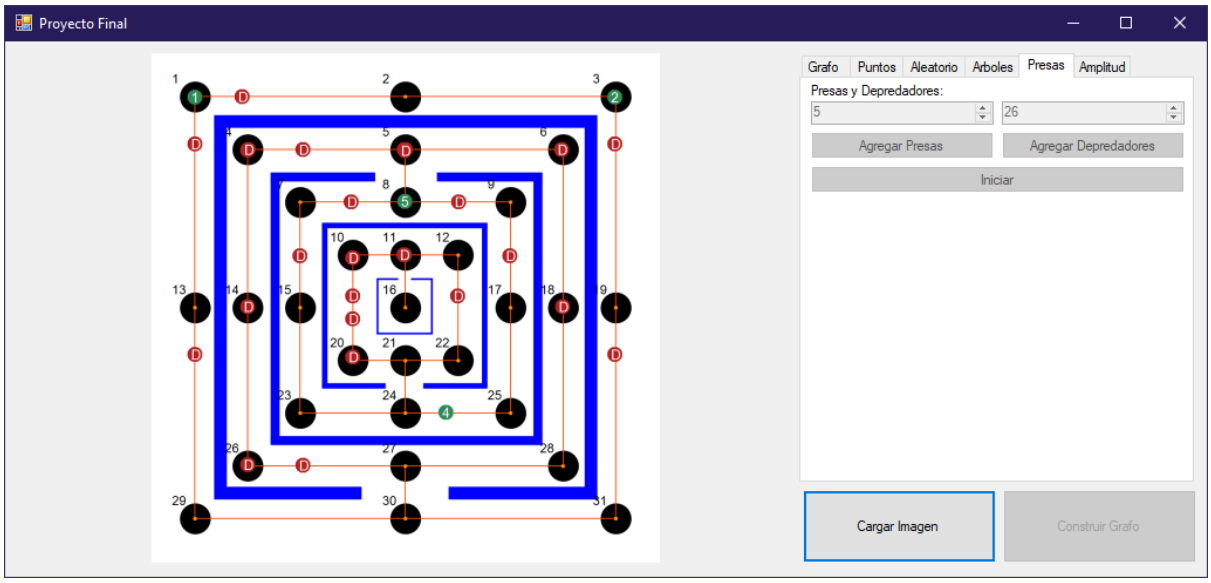


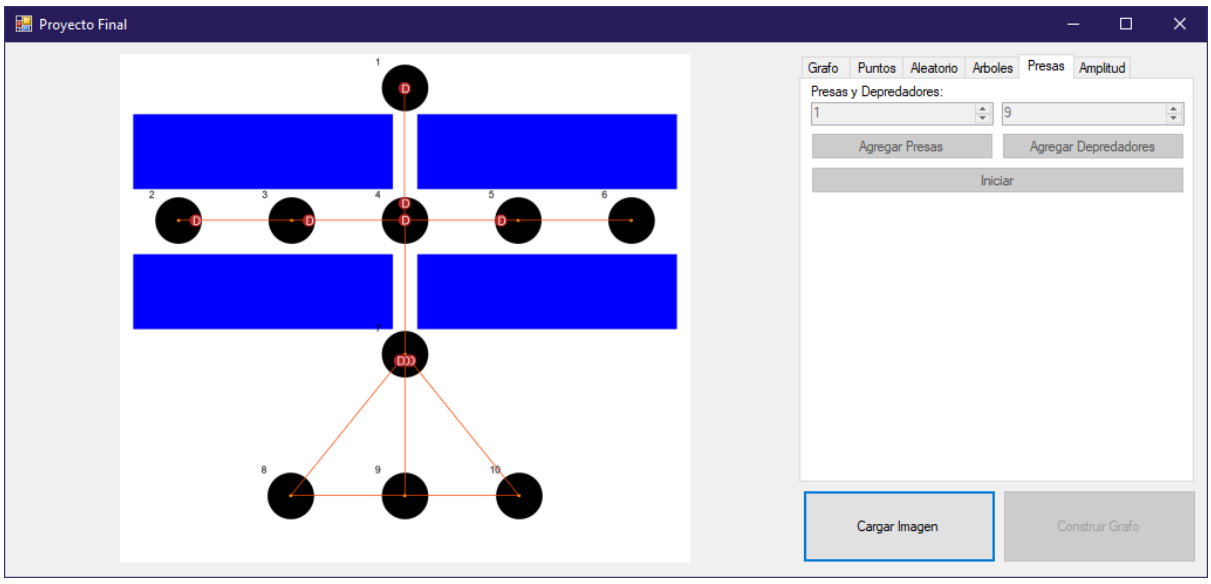


Pestaña de “Presas”

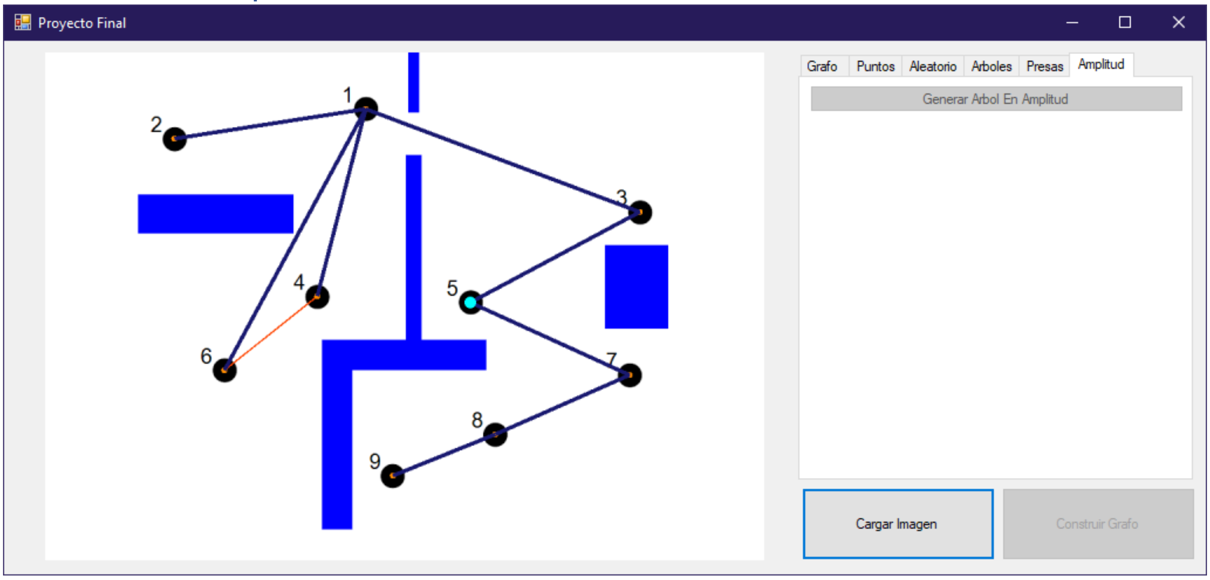


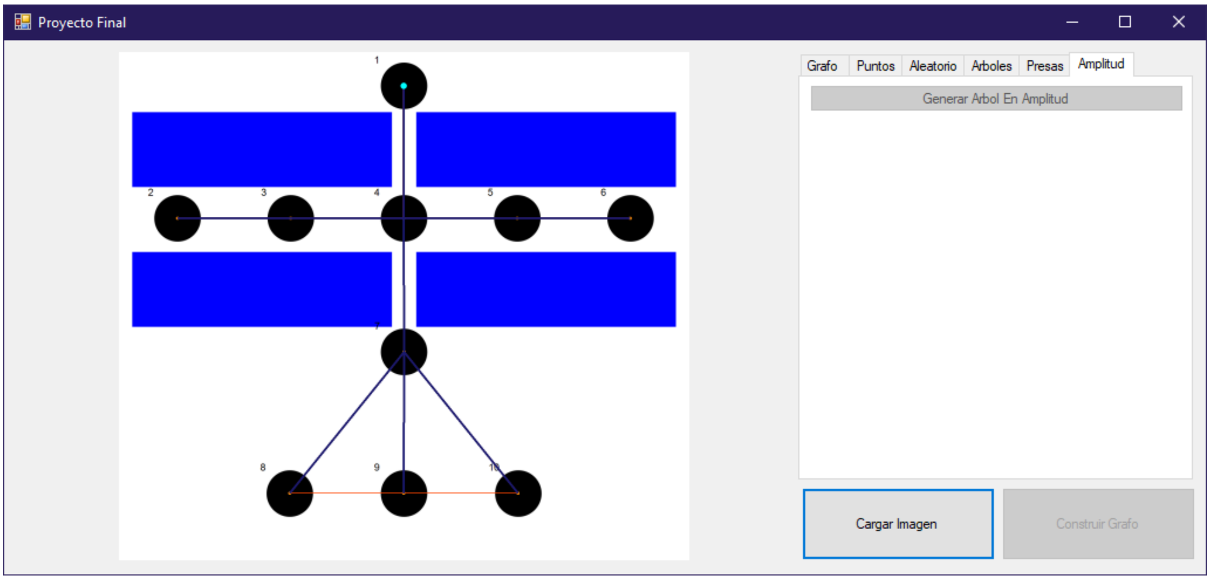

Proyecto Final
Grafo
Puntos
Aleatorio
Arboles
Presas
Amplitud
Presas y Depredadores:
1
8
Agregar Presas
Agregar Depredadores
Iniciar
Cargar Imagen
Construir Grafo

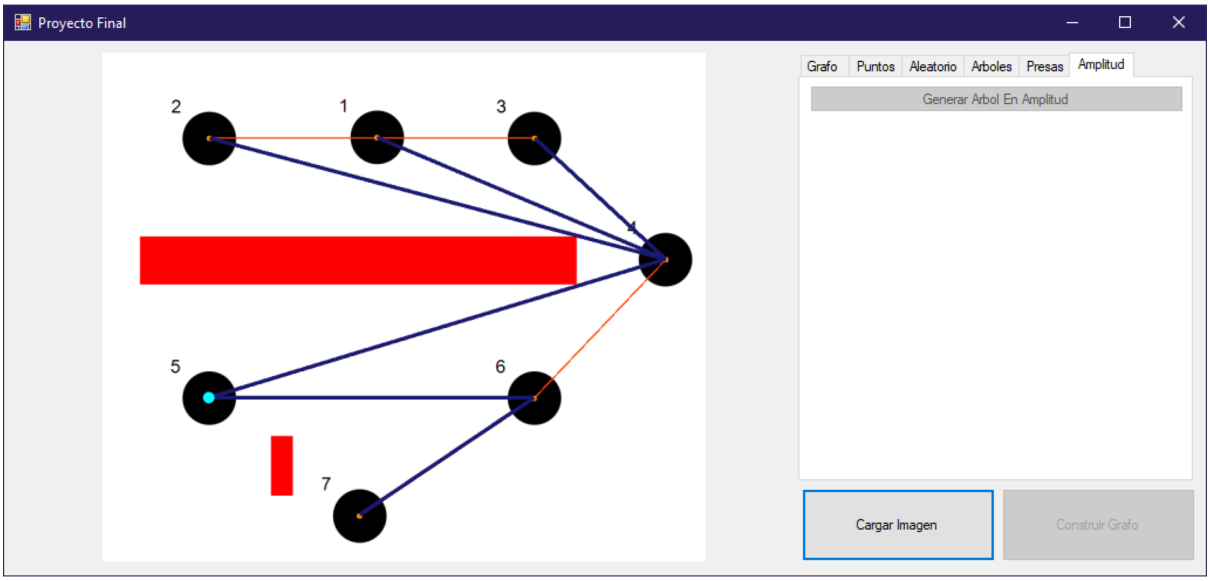

Proyecto Final
Grafo
Puntos
Aleatorio
Arboles
Presas
Amplitud
Presas y Depredadores:
5
26
Agregar Presas
Agregar Depredadores
Iniciar
Cargar Imagen
Construir Grafo

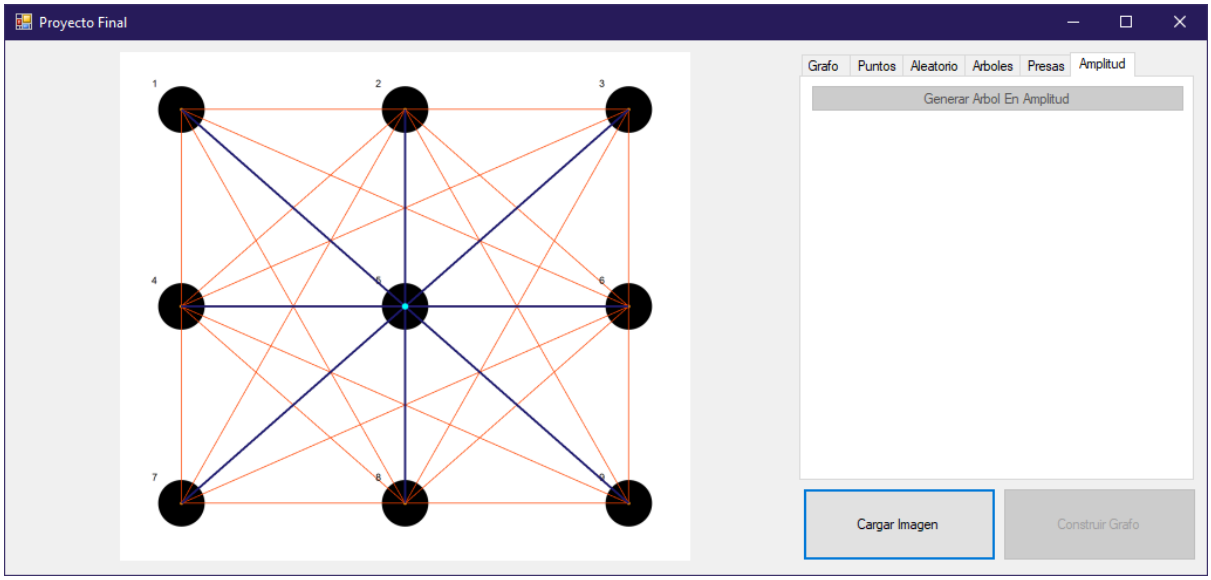

Proyecto Final
Grafo
Puntos
Aleatorio
Arboles
Presas
Amplitud
Presas y Depredadores:
1
9
Agregar Presas
Agregar Depredadores
Iniciar
Cargar Imagen
Construir Grafo

Pestaña de "Amplitud"









Conclusiones

Esta versión del proyecto final consistía en recopilar todas las etapas desarrolladas a lo largo del semestre en un solo sistema computacional. Esto significa que ninguna de las funcionalidades del sistema debe de interferir con las acciones de las demás, todo debe de funcionar a la perfección.

Originalmente, el sistema no se encontraba dividido en pestañas, todos los botones se encontraban presentes en todo momento. Puede que para las etapas individuales esto no haya sido ningún inconveniente; ya que normalmente la revisión únicamente consistía en verificar que las funcionalidades exclusivas de la etapa funcionaban; sin embargo, en el proyecto final todo debe de poder funcionar con todo y, al tener todos los botones presentes me veía obligado a bloquear algunos en situaciones específicas para que el programa pudiera continuar su ejecución sin ningún error. Esto no es nada amigable al usuario, por lo cual decidí reinventar mi interfaz gráfica para que fuera más intuitiva y responsiva.

Dependiendo de la pestaña activa, el mapa de bits cambia. Esto no es nada difícil, una propiedad de la herramienta “TabControl” facilita la gestión de estas. Por ejemplo, si el usuario se encuentra en la pestaña de “Grafo”, entonces se muestra el grafo sin modificación alguna; o si se encuentra en la pestaña de “Árboles”, entonces se muestra el grafo con los árboles de recubrimiento mínimo encontrados. Esto significa que cada pestaña cuenta con su propio mapa de bits, lo que facilita ampliamente la gestión de las funcionalidades del sistema. Ya no es necesario; por ejemplo, limpiar el mapa de bits de los árboles para mostrar a las presas y depredadores; todo puede ser visualizado en cualquier momento.

El hacer esto me obligaba a llevar una gestión perfecta entre los mapas de bits del programa, ya que podría darse la situación en donde se dibujan agentes en el mapa de bits incorrecto y; por consecuencia, se muestran en la pestaña incorrecta. Es por esta misma razón que los “Graphics” del sistema se asignan al presionar los botones, esto obliga al sistema a realizar el dibujo de formas y letras sobre el mapa de bits especifico de la pestaña en la que se encuentra el usuario.

Con respecto a las funcionalidades del sistema, tuve que modificar varias de las ya existentes.

En el caso de los puntos mas cercanos, en lugar de mostrar una caja de texto con los tiempos de ejecución decidí mejor actualizar unas “Labels” en el “Form” principal.

En el caso de los agentes con recorrido aleatorio, me vi obligado a volver a incluir esta funcionalidad en el sistema; ya que previamente la había sustituido por los agentes que recorren los arboles de recubrimiento mínimo. Esto no fue difícil, ya que afortunadamente contaba con un respaldo de la etapa en donde se implementó esa funcionalidad, lo único que tuve que hacer fue adaptarla a mi interfaz actual.

En el caso de los árboles, tuve que modificar el sistema para que este fuera capaz de agregar mas de un agente que recorriera los arboles de recubrimiento minimo. Originalmente creí que con hacer una lista de agentes en lugar de tratar con uno solo bastaría; sin embargo, la situación cambio cuando supe que era mejor especificar si toda la tanda de agentes recorrerá el árbol por Prim o por Kruskal; en lugar de especificar uno por uno el algoritmo a utilizar. Esto me llevo a desechar un “Form” que utilizaba e incluir sus características en el “Form” principal, esto es, permitirle al usuario elegir desde un principio que algoritmo se utilizará. Adicionalmente, decidí agregarle una función en la que se le asigna a todos los agentes generados un vértice origen aleatorio, esto principalmente para evitar tener que ingresar uno por uno el origen en caso de que sean muchos agentes.

En el caso de las presas y depredadores, similar al apartado de los árboles, lo único que tuve que hacer fue que el sistema tuviera la capacidad de agregar múltiples presas. Esto en un principio fue fácil, únicamente me encargue de tener una lista de presas en lugar de tratar con un solo objeto; sin embargo, el hecho de que existieran múltiples presas dentro de la simulación me llevo a pensar en los nuevos casos que podían presentarse, que sucede cuando más de una presa es consumida por un mismo depredador, o que pasa cuando ya no se quiere continuar trabajando con una presa. Con respecto a lo primero, decidí implementarles a los depredadores una lista de presas consumidas, la cual me permite llevar un control con respecto a las presas que aún continúan en la simulación y las que no, esta es limpiada cada vez que las presas son eliminadas de la lista de presas. Y con respecto a lo segundo, decidí que, al no especificarle un nuevo destino a una presa, esta pasa a ser descartada y; por lo tanto, eliminada de la lista de presas. Adicionalmente, decidí modificar el “Form” en el que se especifica el origen de la presa para que ahora incluyera de una vez el destino, esto lo hice por cuestiones de simplicidad; ya que considero que es menos tedioso al usuario insertar de una vez origen y destino.

Todas están modificaciones a la interfaz del sistema puede que hayan sido un poco excesivas; ya que sin estas el sistema aun contaba con todos los requerimientos necesarios para realizar la entrega; sin embargo, considero que mi rediseño resulto agradable y simple, tanto que un usuario que no tenga conocimiento previo del sistema puede hacer uso de el.

Apéndices

Apéndice A. Clase Agent

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Drawing;

namespace Final_Project
{
    class Agent
    {
        // CONSTANTS
        const int LinePositionIncrement = 20;

        // ATTRIBUTES
        private int id;
        private Vertex originVertex;
        private List<Point> path;
        private List<Vertex> visitedVertexList;
        private List<Edge> visitedEdgeList;
        private int linePosition;

        public Agent(int newId, Vertex newOriginVertex)
        {
            id = newId;

            originVertex = newOriginVertex;

            path = new List<Point>();

            visitedVertexList = new List<Vertex>();

            visitedEdgeList = new List<Edge>();

            linePosition = 0;
        }

        public int ID
        {
            get { return id; }
            set { id = value; }
        }

        public Vertex OriginVertex
        {
            get { return originVertex; }
            set { originVertex = value; }
        }

        public List<Point> Path
        {
            get { return path; }
            set { path = value; }
        }

        public List<Edge> VisitedEdgeList
        {
            get { return visitedEdgeList; }
            set { visitedEdgeList = value; }
```

```
        }

        public List<Vertex> VisitedVertexList
        {
            get { return visitedVertexList; }
            set { visitedVertexList = value; }
        }

        public int LinePosition
        {
            get { return linePosition; }
            set { linePosition = value; }
        }

        public void GenerateMSTPath()
        {
            List<Vertex> vertexesOnPath = new List<Vertex>() { originVertex };

            path.AddRange(GetMSTEdgesPath(originVertex, vertexesOnPath));
        }

        public List<Point> GetMSTEdgesPath(Vertex currentVertex, List<Vertex>
vertexesOnPath)
        {
            List<Point> points = new List<Point>();
            Edge returnEdge = null;

            foreach (Edge currentEdge in currentVertex.EdgeList)
            {
                if (!vertexesOnPath.Contains(currentEdge.Destination))
                {
                    vertexesOnPath.Add(currentEdge.Destination);

                    visitedEdgeList.Add(currentEdge);

                    points.AddRange(currentEdge.LinePoints);

                    points.AddRange(GetMSTEdgesPath(currentEdge.Destination,
vertexesOnPath));
                }
                else
                {
                    returnEdge = currentEdge;
                }
            }

            if (returnEdge != null)
            {
                visitedEdgeList.Add(returnEdge);

                points.AddRange(returnEdge.LinePoints);
            }

            return points;
        }

        public void GenerateRandomPath(Random randomNumberGenerator)
        {
            Vertex currentVertex = originVertex;
            Edge randomEdge;
            bool fullPath = false;
```

```
        visitedVertexList.Add(originVertex);

        while (!fullPath)
        {
            randomEdge = GetRandomEdge(randomNumberGenerator, currentVertex);

            if (randomEdge != null)
            {
                path.AddRange(randomEdge.LinePoints);

                currentVertex = randomEdge.Destination;

                if (!visitedVertexList.Contains(currentVertex))
                {
                    visitedVertexList.Add(currentVertex);
                }
            }
            else
            {
                fullPath = true;
            }
        }
    }

    private Edge GetRandomEdge(Random randomNumberGenerator, Vertex currentVertex)
    {
        List<Edge> availableEdges = new List<Edge>();
        Edge randomEdge;
        int randomEdgeIndex;

        foreach (Edge currentEdge in currentVertex.EdgeList)
        {
            if (!visitedEdgeList.Contains(currentEdge))
            {
                availableEdges.Add(currentEdge);
            }
        }

        if (availableEdges.Count != 0)
        {
            randomEdgeIndex = randomNumberGenerator.Next(0, availableEdges.Count);
            randomEdge = availableEdges[randomEdgeIndex];

            visitedEdgeList.Add(randomEdge);

            return randomEdge;
        }

        return null;
    }

    public void UpdateLinePosition()
    {
        if (linePosition == path.Count - 1)
        {
            return;
        }

        linePosition += LinePositionIncrement;

        if (linePosition >= path.Count)
        {
```

```
                linePosition = path.Count - 1;
            }
        }

        public double GetDistanceTraveled()
        {
            double distanceTraveled = 0;

            foreach (Edge currentEdge in visitedEdgeList)
            {
                distanceTraveled += currentEdge.Weight;
            }

            return distanceTraveled;
        }
    }
}
```

Apéndice B. Clase Circle

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Drawing;

namespace Final_Project
{
    class Circle
    {
        // Attributes
        private int id;
        private Point centre;
        private int radius;

        //Methods
        public Circle()
        {
            id = -1;
            centre.X = -1;
            centre.Y = -1;
            radius = -1;
        }

        public Circle(int newId, Point newCentre, int newRadius)
        {
            id = newId;
            centre.X = newCentre.X;
            centre.Y = newCentre.Y;
            radius = newRadius;
        }

        public int ID
        {
            get { return id; }
            set { id = value; }
        }

        public Point Centre
        {
            get { return centre; }
            set { centre = value; }
        }
```

```
        public int Radius
        {
            get { return radius; }
            set { radius = value; }
        }
    }
}
```

Apéndice C. Clase CirclePair

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Drawing;

namespace Final_Project
{
    class CirclePair
    {
        private Circle originCirle;
        private Circle destinationCircle;
        private List<Point> linePoints;

        public CirclePair()
        {
            originCirle = new Circle();
            destinationCircle = new Circle();
            linePoints = new List<Point>();
        }

        public CirclePair(Circle newOriginCircle, Circle newDestinationCircle,
List<Point> newLinePoints)
        {
            originCirle = newOriginCircle;
            destinationCircle = newDestinationCircle;
            linePoints = newLinePoints;
        }

        public Circle OriginCircle
        {
            get { return originCirle; }
            set { originCirle = value; }
        }

        public Circle DestinationCircle
        {
            get { return destinationCircle; }
            set { destinationCircle = value; }
        }

        public List<Point> LinePoints
        {
            get { return linePoints; }
            set { linePoints = value; }
        }
    }
}
```

Apéndice D. Clase ConnectedComponent

```
using System;
using System.Collections.Generic;
```

```
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Final_Project
{
    class ConnectedComponent
    {
        // ATTRIBUTES
        public List<Vertex> VertexList { get; set; }

        // METHODS
        public ConnectedComponent(Vertex firstVertex)
        {
            VertexList = new List<Vertex>
            {
                firstVertex
            };
        }
    }
}
```

Apéndice E. Clase Edge

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Drawing;

namespace Final_Project
{
    class Edge
    {
        public Vertex Origin { get; set; }
        public Vertex Destination { get; set; }
        public double Weight { get; set; }
        public List<Point> LinePoints { get; set; }

        public List<Prey> Preys { get; set; }
        public int NumberOfPredators { get; set; }

        public Edge()
        {
            Origin = new Vertex();
            Destination = new Vertex();
            LinePoints = new List<Point>();

            Preys = new List<Prey>();
            NumberOfPredators = 0;
        }

        public Edge(Vertex newOrigin, Vertex newDestination, List<Point>
newLinePoints)
        {
            Origin = newOrigin;
            Destination = newDestination;
            Weight = Math.Sqrt(Math.Pow(newDestination.VertexCircle.Centre.X -
newOrigin.VertexCircle.Centre.X, 2)
                + Math.Pow(newDestination.VertexCircle.Centre.Y -
newOrigin.VertexCircle.Centre.Y, 2));
            LinePoints = newLinePoints;
```

```
            Preys = new List<Prey>();
            NumberOfPredators = 0;
        }

        public Edge Reverse()
        {
            return Destination.EdgeList.Find(Edge => Edge.Destination == Origin);
        }

        public override string ToString()
        {
            return Origin.VertexCircle.ID.ToString() + " → " +
Destination.VertexCircle.ID.ToString() + " : " + Weight.ToString();
        }

        public override bool Equals(object obj)
        {
            if (obj == null || GetType() != obj.GetType())
            {
                return false;
            }

            Edge compareEdge = (Edge)obj;

            if (Origin == compareEdge.Origin)
            {
                if (Destination == compareEdge.Destination)
                {
                    return true;
                }

                return false;
            }

            if (Origin == compareEdge.Destination)
            {
                if (Destination == compareEdge.Origin)
                {
                    return true;
                }

                return false;
            }

            return false;
        }

        public override int GetHashCode()
        {
            return Origin.GetHashCode() ^ Destination.GetHashCode();
        }
    }

    class CompareEdgesByWeight : IComparer<Edge>
    {
        public int Compare(Edge edgeOne, Edge edgeTwo)
        {
            if (edgeOne.Weight > edgeTwo.Weight)
            {
                return 1;
            }
            else if (edgeOne.Weight < edgeTwo.Weight)
```

```
            {
                return -1;
            }
            else
            {
                return 0;
            }
        }
    }
}
```

Apéndice F. Clase Graph

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Drawing;

namespace Final_Project
{
    class Graph
    {
        // ATTRIBUTES
        public List<Vertex> VertexList { get; set; }

        // METHODS
        public Graph()
        {
            VertexList = new List<Vertex>();
        }

        public Graph(List<Circle> circleList, List<CirclePair> circlePairList)
        {
            VertexList = new List<Vertex>();

            Vertex newVertex, originVertex, destinationVertex;

            foreach (Circle currentCircle in circleList)
            {
                newVertex = new Vertex(currentCircle);

                VertexList.Add(newVertex);
            }

            foreach (CirclePair currentCirclePair in circlePairList)
            {
                originVertex = VertexList.Find(Vertex =>
Vertex.VertexCircle.Equals(currentCirclePair.OriginCircle));
                destinationVertex = VertexList.Find(Vertex =>
Vertex.VertexCircle.Equals(currentCirclePair.DestinationCircle));

                AddEdge(originVertex, destinationVertex,
currentCirclePair.LinePoints);
            }
        }

        public Graph(List<Vertex> newVertexList, List<Edge> newEdgeList)
        {
            Vertex originVertex, destinationVertex;

            VertexList = newVertexList;
```

```
            foreach (Edge newEdge in newEdgeList)
            {
                originVertex = VertexList.Find(Vertex =>
Vertex.Equals(newEdge.Origin));

                if (originVertex != null)
                {
                    destinationVertex = VertexList.Find(Vertex =>
Vertex.Equals(newEdge.Destination));

                    AddEdge(originVertex, destinationVertex, newEdge.LinePoints);
                }
            }
        }

        public void Clear()
        {
            VertexList.Clear();
        }

        private void AddEdge(Vertex originVertex, Vertex destinationVertex,
List<Point> linePoints)
        {
            List<Point> reversedLinePoints;
            Edge newEdge;

            newEdge = new Edge(originVertex, destinationVertex, linePoints);
            originVertex.EdgeList.Add(newEdge);

            reversedLinePoints = new List<Point>(linePoints);
            reversedLinePoints.Reverse();

            newEdge = new Edge(destinationVertex, originVertex, reversedLinePoints);
            destinationVertex.EdgeList.Add(newEdge);
        }

        public bool HasEdges()
        {
            foreach (Vertex currentVertex in VertexList)
            {
                if (currentVertex.EdgeList.Count != 0)
                {
                    return true;
                }
            }

            return false;
        }

        public List<Edge> GetEdges()
        {
            List<Edge> allEdges = new List<Edge>();

            foreach (Vertex currentVertex in VertexList)
            {
                foreach (Edge currentEdge in currentVertex.EdgeList)
                {
                    if (!allEdges.Contains(currentEdge))
                    {
                        allEdges.Add(currentEdge);
                    }
                }
```

```
            }

            return allEdges;
        }

        public double GetTotalWeight()
        {
            List<Edge> allEdges = GetEdges();
            double totalWeight = 0;

            foreach (Edge currentEdge in allEdges)
            {
                totalWeight += currentEdge.Weight;
            }

            return totalWeight;
        }
    }
}
```

Apéndice G. Form MainForm

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.Threading;

namespace Final_Project
{
    public partial class FormFinalProject : Form
    {
        // CONSTANTS
        private enum SideLists { FirstSide = 1, SecondSide }
        private enum SurroundingPixelMovement { UpperSide, RightSide, LowerSide,
LeftSide }
        private enum MSTAlgorithms { Kruskal, Prim }
        private enum Tabs { Graph, Points, Random, Trees, Preys, Breadth }
        const int KruskalMSTPenWidth = 12;
        const int PrimMSTPenWidth = 4;

        // VARIABLES
        private Graph circleGraph;
        private List<Circle> circleList;
        private List<CirclePair> circlePairList;
        private Graph kruskalMST;
        private Graph primMST;
        private List<Agent> MSTAgents;
        private List<Prey> globalPreys;
        private List<Predator> globalPredators;
        private Graph dijkstraGraph;
        private Bitmap processingImage;
        private Bitmap preysAndPredatorsImage;
        private Bitmap closestPairOfPointsImage;
        private Bitmap MSTImage;
        private Bitmap MSTAgentsImage;
        private Bitmap BFSTreeImage;
        private Bitmap agentsWithRandomPathImage;
```

```
        private Graphics imageGraphics;
        private Random numberGenerator;
        private Stopwatch bruteForceOnImageStopwatch;
        private Stopwatch bruteForceOnGraphStopwatch;
        private Stopwatch divideAndConquerOnImageStopwatch;
        private List<Point[]> treeByBreadthPoints;
        private List<Agent> agentsWithRandomPath;

        // FORM METHODS
        public FormFinalProject()
        {
            InitializeComponent();

            OpenFileDialogLoadImage.Filter = "Image files (*.jpg, *.jpeg, *.jpe,
*.jfif, *.png) | *.jpg; *.jpeg; *.jpe; *.jfif; *.png";
            circleList = new List<Circle>();
            circlePairList = new List<CirclePair>();
            globalPredators = new List<Predator>();
            treeByBreadthPoints = new List<Point[]>();
            numberGenerator = new Random();
            agentsWithRandomPath = new List<Agent>();
            MSTAgents = new List<Agent>();
            globalPreys = new List<Prey>();

            // Graph Creation Button
            ButtonBuildGraph.Enabled = false;

            // Points Tab
            ButtonFindClosestPairOfPoints.Enabled = false;

            // Random Tab
            NumericUpDownAgentsWithRandomPath.Enabled = false;
            ButtonCreateAgentsWithRandomPath.Enabled = false;

            // Trees Tab
            ButtonGenerateMST.Enabled = false;
            NumericUpDownMSTAgents.Enabled = false;
            RadioButtonKruskal.Checked = true;
            RadioButtonKruskal.Enabled = false;
            RadioButtonPrim.Enabled = false;
            CheckBoxRandomOrigins.Enabled = false;
            ButtonCreateMSTAgents.Enabled = false;

            // Preys Tab
            NumericUpDownPreys.Enabled = false;
            ButtonAddPreys.Enabled = false;
            NumericUpDownPredators.Enabled = false;
            ButtonAddPredators.Enabled = false;
            ButtonStart.Enabled = false;

            // Breadth Tab
            ButtonGenerateTreeByBreadth.Enabled = false;
        }

        private void ButtonLoadImage_Click(object sender, EventArgs e)
        {
            if (OpenFileDialogLoadImage.ShowDialog() == DialogResult.OK)
            {
                ResetToolsAndAbstractDataTypes();

                processingImage = new Bitmap(OpenFileDialogLoadImage.FileName);
```

```
                preysAndPredatorsImage = new Bitmap(processingImage.Width,
processingImage.Height);
                closestPairOfPointsImage = new Bitmap(processingImage.Width,
processingImage.Height);
                MSTImage = new Bitmap(processingImage);
                BFSTreeImage = new Bitmap(processingImage.Width,
processingImage.Height);
                agentsWithRandomPathImage = new Bitmap(processingImage.Width,
processingImage.Height);
                MSTAgentsImage = new Bitmap(processingImage.Width,
processingImage.Height);

                PictureBoxImage.BackgroundImage = processingImage;
                /**PictureBoxImage.Image = animationImage;**/

                // Graph Creation Button
                ButtonBuildGraph.Enabled = true;
            }
        }

        private void ButtonBuildGraph_Click(object sender, EventArgs e)
        {
            ProcessBitmap();
            SetValidCirclePairs();

            circleGraph = new Graph(circleList, circlePairList);
            UpdateGraphDetails();

            imageGraphics = Graphics.FromImage(processingImage);
            DrawCircleConnections();
            DrawCircleCentres();

            // Graph Creation Button
            ButtonBuildGraph.Enabled = false;

            // Points Tab
            ButtonFindClosestPairOfPoints.Enabled = true;

            // Random Tab
            NumericUpDownAgentsWithRandomPath.Maximum = circleList.Count;
            NumericUpDownAgentsWithRandomPath.Enabled = true;
            ButtonCreateAgentsWithRandomPath.Enabled = true;

            // Trees Tab
            MSTImage = new Bitmap(processingImage);
            ButtonGenerateMST.Enabled = true;
            /**NumericUpDownMSTAgents.Maximum = circleList.Count;**/

            ButtonAddPreys.Enabled = true;
            ButtonGenerateTreeByBreadth.Enabled = true;
            NumericUpDownPreys.Maximum = circleList.Count;
            NumericUpDownPreys.Enabled = true;
        }

        private void TabControlCapabilities_Click(object sender, EventArgs e)
        {
            if (TimerAgentsWithRandomPathMovement.Enabled)
            {
                TimerAgentsWithRandomPathMovement.Enabled = false;

                TabControlCapabilities.SelectedIndex = (int)Tabs.Random;
                ShowErrorMessageBox("Hay una simulación en curso.");
```

```
            TimerAgentsWithRandomPathMovement.Enabled = true;
        }
        else if (TimerMSTAgentsMovement.Enabled)
        {
            TimerMSTAgentsMovement.Enabled = false;

            TabControlCapabilities.SelectedIndex = (int)Tabs.Trees;
            ShowErrorMessageBox("Hay una simulación en curso.");

            TimerMSTAgentsMovement.Enabled = true;
        }
        else if (TimerPreysAndPredatorsMovement.Enabled)
        {
            TimerPreysAndPredatorsMovement.Enabled = false;

            TabControlCapabilities.SelectedIndex = (int)Tabs.Preys;
            ShowErrorMessageBox("Hay una simulación en curso.");

            TimerPreysAndPredatorsMovement.Enabled = true;
        }

        switch (TabControlCapabilities.SelectedIndex)
        {
            case (int)Tabs.Graph:
                PictureBoxImage.BackgroundImage = processingImage;
                PictureBoxImage.Image = null;
                break;

            case (int)Tabs.Points:
                PictureBoxImage.BackgroundImage = processingImage;
                PictureBoxImage.Image = closestPairOfPointsImage;
                break;

            case (int)Tabs.Random:
                PictureBoxImage.BackgroundImage = processingImage;
                PictureBoxImage.Image = agentsWithRandomPathImage;
                break;

            case (int)Tabs.Trees:
                PictureBoxImage.BackgroundImage = MSTImage;
                PictureBoxImage.Image = MSTAgentsImage;
                break;

            case (int)Tabs.Preys:
                PictureBoxImage.BackgroundImage = processingImage;
                PictureBoxImage.Image = preysAndPredatorsImage;
                break;

            case (int)Tabs.Breadth:
                PictureBoxImage.BackgroundImage = processingImage;
                PictureBoxImage.Image = BFSTreeImage;
                break;
        }
    }

    private void ButtonFindClosestPairOfPoints_Click(object sender, EventArgs e)
    {
        if (circleList.Count <= 1)
        {
            ShowErrorMessageBox("No se encontró más de un punto.");
```

```
                return;
            }

            /**PictureBoxImage.Image = closestPairOfPointsImage;**/
            imageGraphics = Graphics.FromImage(closestPairOfPointsImage);

            FindClosestPairOfPointsByBruteForceOnImage();
            LabelBruteForceOnImageExecutionTimeValue.Text =
bruteForceOnImageStopwatch.ElapsedMilliseconds.ToString() + " ms";

            FindClosestPairOfPointsByBruteForceOnGraph();
            LabelBruteForceOnGraphExecutionTimeValue.Text =
bruteForceOnGraphStopwatch.ElapsedMilliseconds.ToString() + " ms";

            FindClosestPairOfPointsByDivideAndConquerOnImage();
            LabelDivideAndConquerOnImageExecutionTimeValue.Text =
divideAndConquerOnImageStopwatch.ElapsedMilliseconds.ToString() + " ms";

            ButtonFindClosestPairOfPoints.Enabled = false;
        }

        private void ButtonGenerateMST_Click(object sender, EventArgs e)
        {
            if (!circleGraph.HasEdges())
            {
                ShowErrorMessageBox("El grafo no tiene aristas.");

                return;
            }

            int selectedOriginVertexIndex = 0;

            using (FormVertexSelection PrimMSTOriginVertexForm = new
FormVertexSelection(circleGraph.VertexList, "Origen"))
            {
                if (PrimMSTOriginVertexForm.ShowDialog() == DialogResult.OK)
                {
                    selectedOriginVertexIndex =
PrimMSTOriginVertexForm.GetSelectedVertex();
                }
                else
                {
                    return;
                }
            }

            imageGraphics = Graphics.FromImage(MSTAgentsImage);
            imageGraphics.Clear(Color.Transparent);

            TreeViewMSTAgentDetails.Nodes.Clear();

            MSTImage = new Bitmap(processingImage);
            PictureBoxImage.BackgroundImage = MSTImage;

            imageGraphics = Graphics.FromImage(MSTImage);

            DrawMSTWithKruskal();
            DrawMSTWithPrim(circleGraph.VertexList[selectedOriginVertexIndex]);

            /**PictureBoxImage.Refresh();**/

            // Trees Tab
```

```
            NumericUpDownMSTAgents.Maximum = circleList.Count;
            NumericUpDownMSTAgents.Enabled = true;
            RadioButtonKruskal.Enabled = true;
            RadioButtonPrim.Enabled = true;
            CheckBoxRandomOrigins.Enabled = true;
            ButtonCreateMSTAgents.Enabled = true;

            /**ButtonGenerateMST.Enabled = false;
            NumericUpDownMSTAgents.Maximum = circleList.Count;
            NumericUpDownMSTAgents.Enabled = true;
            ButtonCreateMSTAgents.Enabled = true;
            ButtonAddPreys.Enabled = false;
            NumericUpDownAgentsWithRandomPath.Enabled = false;
            ButtonCreateAgentsWithRandomPath.Enabled = false;
            ButtonFindClosestPairOfPoints.Enabled = false;
            ButtonGenerateTreeByBreadth.Enabled = false;**/
        }

        private void ButtonCreateMSTAgents_Click(object sender, EventArgs e)
        {
            if (NumericUpDownMSTAgents.Value == 0)
            {
                ShowErrorMessageBox("El número de agentes debe ser mayor a cero.");

                return;
            }

            List<Vertex> remainingKruskalMSTVertexes = new
List<Vertex>(kruskalMST.VertexList);
            List<Vertex> remainingPrimMSTVertexes = new
List<Vertex>(primMST.VertexList);
            Vertex originVertex;
            int numberOfMSTAgents = (int)NumericUpDownMSTAgents.Value;

            imageGraphics = Graphics.FromImage(MSTAgentsImage);

            MSTAgents.Clear();
            TreeViewMSTAgentDetails.Nodes.Clear();

            if (CheckBoxRandomOrigins.Checked)
            {
                for (int i = 0; i < numberOfMSTAgents; i++)
                {
                    if (RadioButtonKruskal.Checked)
                    {
                        originVertex =
remainingKruskalMSTVertexes[numberGenerator.Next(0,
remainingKruskalMSTVertexes.Count)];

                        MSTAgents.Add(new Agent(i + 1,
kruskalMST.VertexList.Find(Vertex => Vertex == originVertex)));

                        remainingKruskalMSTVertexes.Remove(originVertex);
                    }
                    else
                    {
                        originVertex =
remainingPrimMSTVertexes[numberGenerator.Next(0, remainingPrimMSTVertexes.Count)];

                        MSTAgents.Add(new Agent(i + 1, primMST.VertexList.Find(Vertex
=> Vertex == originVertex)));
```

```
                        remainingPrimMSTVertexes.Remove(originVertex);
                    }

                    MSTAgents[MSTAgents.Count - 1].GenerateMSTPath();
                }
            }
            else
            {
                remainingKruskalMSTVertexes.Sort((vertexOne, vertexTwo) =>
vertexOne.VertexCircle.ID.CompareTo(vertexTwo.VertexCircle.ID));
                remainingPrimMSTVertexes.Sort((vertexOne, vertexTwo) =>
vertexOne.VertexCircle.ID.CompareTo(vertexTwo.VertexCircle.ID));

                for (int i = 0; i < numberOfMSTAgents; i++)
                {
                    if (RadioButtonKruskal.Checked)
                    {
                        using (FormVertexSelection agentOriginVertexForm = new
FormVertexSelection(remainingKruskalMSTVertexes, "Origen con Kruskal"))
                        {
                            if (agentOriginVertexForm.ShowDialog() == DialogResult.OK)
                            {
                                originVertex =
remainingKruskalMSTVertexes[agentOriginVertexForm.GetSelectedVertex()];

                                MSTAgents.Add(new Agent(i + 1,
kruskalMST.VertexList.Find(Vertex => Vertex == originVertex)));

                                remainingKruskalMSTVertexes.Remove(originVertex);
                            }
                            else
                            {
                                return;
                            }
                        }
                    }
                    else
                    {
                        using (FormVertexSelection agentOriginVertexForm = new
FormVertexSelection(remainingPrimMSTVertexes, "Origen con Prim"))
                        {
                            if (agentOriginVertexForm.ShowDialog() == DialogResult.OK)
                            {
                                originVertex =
remainingPrimMSTVertexes[agentOriginVertexForm.GetSelectedVertex()];

                                MSTAgents.Add(new Agent(i + 1,
primMST.VertexList.Find(Vertex => Vertex == originVertex)));

                                remainingPrimMSTVertexes.Remove(originVertex);
                            }
                            else
                            {
                                return;
                            }
                        }
                    }

                    MSTAgents[MSTAgents.Count - 1].GenerateMSTPath();
                }
            }
```

```
            /**PictureBoxImage.BackgroundImage = MSTImage;**/
            /**PictureBoxImage.Image = animationImage;**/

            imageGraphics = Graphics.FromImage(MSTAgentsImage);

            // Trees Tab
            ButtonGenerateMST.Enabled = false;
            NumericUpDownMSTAgents.Enabled = false;
            RadioButtonKruskal.Enabled = false;
            RadioButtonPrim.Enabled = false;
            CheckBoxRandomOrigins.Enabled = false;
            ButtonCreateMSTAgents.Enabled = false;

            TimerMSTAgentsMovement.Enabled = true;
        }

        private void TimerMSTAgentsMovement_Tick(object sender, EventArgs e)
        {
            int numberOfAgentsHalted = 0;

            imageGraphics.Clear(Color.Transparent);

            foreach (Agent currentAgent in MSTAgents)
            {
                DrawAgent(currentAgent.ID.ToString(),
currentAgent.Path[currentAgent.LinePosition], Color.Firebrick);

                if (currentAgent.LinePosition == currentAgent.Path.Count - 1)
                {
                    numberOfAgentsHalted++;

                    if (numberOfAgentsHalted == MSTAgents.Count)
                    {
                        TimerMSTAgentsMovement.Enabled = false;

                        // Tree Tab
                        ButtonGenerateMST.Enabled = true;
                        NumericUpDownMSTAgents.Enabled = true;
                        RadioButtonKruskal.Enabled = true;
                        RadioButtonPrim.Enabled = true;
                        CheckBoxRandomOrigins.Enabled = true;
                        ButtonCreateMSTAgents.Enabled = true;

                        /**PictureBoxImage.BackgroundImage = processingImage;**/

                        UpdateMSTAgentsDetails();
                    }
                }

                currentAgent.UpdateLinePosition();
            }

            PictureBoxImage.Refresh();
        }

        private void ButtonAddPrey_Click(object sender, EventArgs e)
        {
            if (!circleGraph.HasEdges())
            {
                ShowErrorMessageBox("El grafo no tiene aristas.");

                return;
```

```
            }
            else if (NumericUpDownPreys.Value == 0)
            {
                ShowErrorMessageBox("El número de presas debe ser mayor a cero.");

                return;
            }

            List<Vertex> remainingVertexes;
            Prey newPrey;
            bool validPrey;
            int numberOfPreys = (int)NumericUpDownPreys.Value;

            dijkstraGraph = new Graph(circleList, circlePairList);
            remainingVertexes = new List<Vertex>(dijkstraGraph.VertexList);

            /**PictureBoxImage.Image = animationImage;**/

            imageGraphics = Graphics.FromImage(preysAndPredatorsImage);
            imageGraphics.Clear(Color.Transparent);

            globalPreys.Clear();

            for (int i = 0; i < numberOfPreys; i++)
            {
                newPrey = new Prey();
                validPrey = false;

                using (FormPreyVertexSelection preyVertexSelectionForm = new
FormPreyVertexSelection(remainingVertexes, dijkstraGraph.VertexList, (i +
1).ToString()))
                {
                    while (!validPrey)
                    {
                        if (preyVertexSelectionForm.ShowDialog() == DialogResult.OK)
                        {
                            newPrey.OriginVertex =
dijkstraGraph.VertexList.Find(Vertex => Vertex ==
remainingVertexes[preyVertexSelectionForm.GetSelectedOriginVertex()]);

                            if (newPrey.OriginVertex ==
dijkstraGraph.VertexList[preyVertexSelectionForm.GetSelectedDestinationVertex()])
                            {
                                ShowErrorMessageBox("El vértice destino no puede ser
igual al origen.");
                            }
                            else
                            {
                                try
                                {
                                    newPrey.GeneratePath(dijkstraGraph.VertexList,
dijkstraGraph.VertexList[preyVertexSelectionForm.GetSelectedDestinationVertex()]);

                                    newPrey.ID = i + 1;
                                    globalPreys.Add(newPrey);

                                    DrawAgent(newPrey.ID.ToString(),
newPrey.OriginVertex.VertexCircle.Centre, Color.SeaGreen);
                                    PictureBoxImage.Refresh();

                                    validPrey = true;
```

```
remainingVertexes.RemoveAt(preyVertexSelectionForm.GetSelectedOriginVertex());
                                }
                                catch (NullReferenceException)
                                {
                                    ShowErrorMessageBox("No existe una ruta entre el
vértice origen y el destino.");

                                    newPrey.DijkstraElements.Clear();
                                }
                            }
                        }
                        else
                        {
                            return;
                        }
                    }
                }
            }

            // Preys Tab
            NumericUpDownPreys.Enabled = false;
            ButtonAddPreys.Enabled = false;
            NumericUpDownPredators.Maximum = dijkstraGraph.VertexList.Count -
numberOfPreys;
            NumericUpDownPredators.Enabled = true;
            ButtonAddPredators.Enabled = true;
        }

        private void ButtonAddPredators_Click(object sender, EventArgs e)
        {
            List<Vertex> availableVertexes = new List<Vertex>();
            Vertex chosenVertex;
            int numberOfPredators = (int)NumericUpDownPredators.Value;

            foreach (Vertex currentVertex in dijkstraGraph.VertexList)
            {
                if (globalPreys.Find(Prey => Prey.OriginVertex == currentVertex) ==
null)
                {
                    availableVertexes.Add(currentVertex);
                }
            }

            imageGraphics = Graphics.FromImage(preysAndPredatorsImage);

            globalPredators.Clear();

            for (int i = 0; i < numberOfPredators; i++)
            {
                chosenVertex = availableVertexes[numberGenerator.Next(0,
availableVertexes.Count)];

                globalPredators.Add(new Predator(i + 1, chosenVertex));

                DrawAgent("D", chosenVertex.VertexCircle.Centre, Color.Firebrick);

                availableVertexes.Remove(chosenVertex);
            }

            PictureBoxImage.Refresh();
```

```
            // Preys Tab
            NumericUpDownPredators.Enabled = false;
            ButtonAddPredators.Enabled = false;
            ButtonStart.Enabled = true;
        }

        private void ButtonStart_Click(object sender, EventArgs e)
        {
            foreach (Prey currentPrey in globalPreys)
            {
                currentPrey.IsActive = true;
            }

            foreach (Predator currentPredator in globalPredators)
            {
                currentPredator.GeneratePath();
            }

            imageGraphics = Graphics.FromImage(preysAndPredatorsImage);

            // Preys Tab
            ButtonStart.Enabled = false;

            TimerPreysAndPredatorsMovement.Enabled = true;
        }

        private void TimerPreysAndPredatorsMovement_Tick(object sender, EventArgs e)
        {
            Prey globalPrey;

            imageGraphics.Clear(Color.Transparent);

            for (int i = 0; i < globalPreys.Count; i++)
            {
                globalPrey = globalPreys[i];

                if (!globalPrey.IsActive)
                {
                    bool validDestination = false;

                    TimerPreysAndPredatorsMovement.Enabled = false;

                    globalPreys[i] = new Prey(globalPrey.ID,
globalPrey.CurrentEdge.Destination);
                    globalPrey = globalPreys[i];

                    while (!validDestination)
                    {
                        using (FormVertexSelection preyDestinationVertexSelectionForm
= new FormVertexSelection(dijkstraGraph.VertexList, "Destino Para Presa " +
globalPrey.ID.ToString()))
                        {
                            if (preyDestinationVertexSelectionForm.ShowDialog() ==
DialogResult.OK)
                            {
                                if
(dijkstraGraph.VertexList[preyDestinationVertexSelectionForm.GetSelectedVertex()] ==
globalPrey.OriginVertex)
                                {
                                    ShowErrorMessageBox("El vértice destino no puede
ser igual al origen.");
                                }
```

```
                                else
                                {
                                    try
                                    {

globalPrey.GeneratePath(dijkstraGraph.VertexList,
dijkstraGraph.VertexList[preyDestinationVertexSelectionForm.GetSelectedVertex()]);

                                        globalPrey.IsActive = true;
                                        TimerPreysAndPredatorsMovement.Enabled = true;
                                        validDestination = true;
                                    }
                                    catch (NullReferenceException)
                                    {
                                        ShowErrorMessageBox("No existe una ruta entre
el vértice origen y el destino.");

                                        globalPrey.DijkstraElements.Clear();
                                    }
                                }
                            }
                            else
                            {
                                if (globalPreys.Count == 1)
                                {
                                    DrawAgent(globalPrey.ID.ToString(),
globalPrey.OriginVertex.VertexCircle.Centre, Color.SeaGreen);
                                }

                                globalPreys.Remove(globalPrey);
                                i--;

                                validDestination = true;

                                if (globalPreys.Count == 0)
                                {
                                    foreach (Predator globalPredator in
globalPredators)
                                    {
                                        DrawAgent("D",
globalPredator.CurrentEdge.LinePoints[globalPredator.EdgePosition], Color.Firebrick);
                                    }

                                    // Preys Tab
                                    NumericUpDownPreys.Enabled = true;
                                    ButtonAddPreys.Enabled = true;

                                    PictureBoxImage.Refresh();

                                    return;
                                }
                                else
                                {
                                    TimerPreysAndPredatorsMovement.Enabled = true;
                                }
                            }
                        }
                    }
                }
                else
                {
                    globalPrey.UpdateEdgePosition();
```

```
                    DrawAgent(globalPrey.ID.ToString(),
globalPrey.CurrentEdge.LinePoints[globalPrey.EdgePosition], Color.SeaGreen);
                }
            }

            foreach (Predator globalPredator in globalPredators)
            {
                globalPredator.UpdateEdgePosition();

                // Not sure if this should be here, but let's give it a go
                if (globalPredator.HuntingMode)
                {
                    if (globalPredator.TryHunting())
                    {
                        foreach (Prey huntedPrey in globalPredator.HuntedPreys)
                        {
                            globalPreys.Remove(huntedPrey);
                        }

                        globalPredator.HuntedPreys.Clear();

                        if (globalPreys.Count == 0)
                        {
                            TimerPreysAndPredatorsMovement.Enabled = false;

                            // Preys Tab
                            NumericUpDownPreys.Enabled = true;
                            ButtonAddPreys.Enabled = true;

                            imageGraphics.Clear(Color.Transparent);

                            DrawAgent("D",
globalPredator.CurrentEdge.LinePoints[globalPredator.EdgePosition], Color.Firebrick);

                            PictureBoxImage.Refresh();

                            return;
                        }
                    }
                }

                DrawAgent("D",
globalPredator.CurrentEdge.LinePoints[globalPredator.EdgePosition], Color.Firebrick);
            }

            PictureBoxImage.Refresh();
        }

        private void ButtonGenerateTreeByBreadth_Click(object sender, EventArgs e)
        {
            if (!circleGraph.HasEdges())
            {
                ShowErrorMessageBox("El grafo no tiene aristas.");

                return;
            }

            foreach (Vertex graphVertex in circleGraph.VertexList)
            {
                treeByBreadthPoints.Clear();
```

```
                if (BreadthFirstSearch(graphVertex))
                {
                    PictureBoxImage.Image = BFSTreeImage;

                    imageGraphics = Graphics.FromImage(BFSTreeImage);

                    foreach(Point[] pointPair in treeByBreadthPoints)
                    {
                        DrawLine(pointPair[0], pointPair[1], Color.MidnightBlue, 5);
                    }

                    DrawCircle(graphVertex.VertexCircle.Centre, 15, Color.Aqua);
                    PictureBoxImage.Refresh();

                    // Breadth Tab
                    ButtonGenerateTreeByBreadth.Enabled = false;

                    return;
                }
            }

            ShowErrorMessageBox("No se encontró un arbol.");
        }

        private void ButtonCreateAgentsWithRandomPath_Click(object sender, EventArgs
e)
        {
            if (!circleGraph.HasEdges())
            {
                ShowErrorMessageBox("El grafo no cuenta con aristas.");
            }
            else if (NumericUpDownAgentsWithRandomPath.Value == 0)
            {
                ShowErrorMessageBox("El número de agentes debe ser mayor a cero.");
            }
            else
            {
                List<Vertex> remainingVertexList = new
List<Vertex>(circleGraph.VertexList);
                Agent newAgent;
                int originVertexIndex;

                agentsWithRandomPath.Clear();
                TreeViewAgentsWithRandomPathDetails.Nodes.Clear();

                for (int i = 0; i < NumericUpDownAgentsWithRandomPath.Value; i++)
                {
                    originVertexIndex = numberGenerator.Next(0,
remainingVertexList.Count);

                    newAgent = new Agent(i + 1, circleGraph.VertexList.Find(Vertex =>
Vertex == remainingVertexList[originVertexIndex]));

                    newAgent.GenerateRandomPath(numberGenerator);

                    agentsWithRandomPath.Add(newAgent);

                    remainingVertexList.RemoveAt(originVertexIndex);
                }

                /**PictureBoxImage.Image = animationImage;**/
                imageGraphics = Graphics.FromImage(agentsWithRandomPathImage);
```

```
                // Random Tab
                NumericUpDownAgentsWithRandomPath.Enabled = false;
                ButtonCreateAgentsWithRandomPath.Enabled = false;

                TimerAgentsWithRandomPathMovement.Enabled = true;
            }
        }

        private void TimerAgentsWithRandomPathMovement_Tick(object sender, EventArgs
e)
        {
            int numberOfAgentsHalted = 0;

            imageGraphics.Clear(Color.Transparent);

            foreach (Agent currentAgent in agentsWithRandomPath)
            {
                DrawCircle(currentAgent.Path[currentAgent.LinePosition], 30,
Color.Firebrick);

                imageGraphics.DrawString(currentAgent.ID.ToString(), new Font("Arial",
20), new SolidBrush(Color.White),
                    currentAgent.Path[currentAgent.LinePosition].X - 11,
currentAgent.Path[currentAgent.LinePosition].Y - 15);

                if (currentAgent.LinePosition == currentAgent.Path.Count - 1)
                {
                    numberOfAgentsHalted++;
                }

                currentAgent.UpdateLinePosition();
            }

            PictureBoxImage.Refresh();

            if (numberOfAgentsHalted == agentsWithRandomPath.Count)
            {
                TimerAgentsWithRandomPathMovement.Enabled = false;

                // Random Tab
                NumericUpDownAgentsWithRandomPath.Enabled = true;
                ButtonCreateAgentsWithRandomPath.Enabled = true;

                UpdateAgentsWithRandomPathDetails();
            }
        }

        // USER METHODS
        private void ShowErrorMessageBox(string error)
        {
            MessageBox.Show(error, "Error", MessageBoxButtons.OK,
MessageBoxIcon.Error);
        }

        private void DrawAgent(string ID, Point location, Color color)
        {
            DrawCircle(location, 30, color);

            imageGraphics.DrawString(ID, new Font("Arial", 20), new
SolidBrush(Color.White), location.X - 11, location.Y - 15);
        }
```

```
        private void DrawMSTWithKruskal()
        {
            List<Edge> solutionEdges = BuildMSTWithKruskal();

            foreach (Edge currentEdge in solutionEdges)
            {
                DrawLine(currentEdge.Origin.VertexCircle.Centre,
currentEdge.Destination.VertexCircle.Centre, Color.MediumVioletRed,
KruskalMSTPenWidth);
            }

            LabelKruskalMSTTotalWeightValue.Text =
kruskalMST.GetTotalWeight().ToString();

            PictureBoxImage.Refresh();

            using (FormMSTSolutionSet MSTSolutionSetForm = new
FormMSTSolutionSet(solutionEdges, "Kruskal"))
            {
                MSTSolutionSetForm.ShowDialog();
            }
        }

        private List<Edge> BuildMSTWithKruskal()
        {
            List<Edge> candidateEdges = circleGraph.GetEdges(), promisingEdges = new
List<Edge>();
            List<ConnectedComponent> connectedComponents = new
List<ConnectedComponent>();
            List<Vertex> emptyVertexes = new List<Vertex>();
            Edge selectedEdge;
            ConnectedComponent originConnectedComponent,
destinationConnectedComponent;

            foreach (Circle currentCircle in circleList)
            {
                connectedComponents.Add(new ConnectedComponent(new
Vertex(currentCircle)));
            }

            candidateEdges.Sort((edgeOne, edgeTwo) =>
edgeOne.Weight.CompareTo(edgeTwo.Weight));

            while (promisingEdges.Count != circleGraph.VertexList.Count - 1)
            {
                selectedEdge = candidateEdges[0];

                originConnectedComponent = connectedComponents.Find(ConnectedComponent
=> ConnectedComponent.VertexList.Contains(selectedEdge.Origin));
                destinationConnectedComponent =
connectedComponents.Find(ConnectedComponent =>
ConnectedComponent.VertexList.Contains(selectedEdge.Destination));

                if (originConnectedComponent != destinationConnectedComponent)
                {
                    promisingEdges.Add(selectedEdge);


originConnectedComponent.VertexList.AddRange(destinationConnectedComponent.VertexList)
;
```

```
                    connectedComponents.Remove(destinationConnectedComponent);
                }

                candidateEdges.RemoveAt(0);

                if (candidateEdges.Count == 0)
                {
                    break;
                }
            }

            foreach (ConnectedComponent currentConnectedComponent in
connectedComponents)
            {
                emptyVertexes.AddRange(currentConnectedComponent.VertexList);
            }

            kruskalMST = new Graph(emptyVertexes, promisingEdges);

            LabelNumberOfMSTSubgraphsValue.Text =
connectedComponents.Count.ToString();

            return promisingEdges;
        }

        private void DrawMSTWithPrim(Vertex originVertex)
        {
            List<Edge> solutionEdges = BuildMSTWithPrim(originVertex);

            foreach (Edge currentEdge in solutionEdges)
            {
                DrawLine(currentEdge.Origin.VertexCircle.Centre,
currentEdge.Destination.VertexCircle.Centre, Color.MediumTurquoise, PrimMSTPenWidth);
            }

            LabelPrimMSTTotalWeightValue.Text = primMST.GetTotalWeight().ToString();

            PictureBoxImage.Refresh();

            using (FormMSTSolutionSet MSTSolutionSetForm = new
FormMSTSolutionSet(solutionEdges, "Prim"))
            {
                MSTSolutionSetForm.ShowDialog();
            }
        }

        private List<Edge> BuildMSTWithPrim(Vertex originVertex)
        {
            List<Edge> candidateEdges = originVertex.GetEdges(), promisingEdges = new
List<Edge>();
            List<Vertex> MSTVertexes = new List<Vertex>() { new
Vertex(originVertex.VertexCircle) };
            Edge selectedEdge;
            int numberOfSubgraphs = 1;

            candidateEdges.Sort(new CompareEdgesByWeight());

            while (promisingEdges.Count != circleGraph.VertexList.Count - 1)
            {
                if (candidateEdges.Count == 0)
                {
```

```
                    List<Vertex> remainingVertexes =
circleGraph.VertexList.Except(MSTVertexes).ToList();

                    if (remainingVertexes.Count == 0)
                    {
                        break;
                    }

                    numberOfSubgraphs++;

                    AddEdgesToCandidateEdges(candidateEdges,
remainingVertexes[0].GetEdges());

                    MSTVertexes.Add(new Vertex(remainingVertexes[0].VertexCircle));
                }

                selectedEdge = candidateEdges[0];

                if (ValidEdgeOnPrim(selectedEdge, MSTVertexes))
                {
                    promisingEdges.Add(selectedEdge);

                    if (MSTVertexes.Contains(selectedEdge.Origin))
                    {
                        MSTVertexes.Add(new
Vertex(selectedEdge.Destination.VertexCircle));

                        AddEdgesToCandidateEdges(candidateEdges,
selectedEdge.Destination.EdgeList);
                    }
                    else
                    {
                        MSTVertexes.Add(new Vertex(selectedEdge.Origin.VertexCircle));

                        AddEdgesToCandidateEdges(candidateEdges,
selectedEdge.Origin.EdgeList);
                    }
                }

                candidateEdges.Remove(selectedEdge);
            }

            primMST = new Graph(MSTVertexes, promisingEdges);

            LabelNumberOfMSTSubgraphsValue.Text = numberOfSubgraphs.ToString();

            return promisingEdges;
        }


        private void AddEdgesToCandidateEdges(List<Edge> candidateEdges, List<Edge>
newCandidateEdges)
        {
            int currentEdgeIndex;

            foreach (Edge currentEdge in newCandidateEdges)
            {
                currentEdgeIndex = candidateEdges.BinarySearch(currentEdge, new
CompareEdgesByWeight());

                if (currentEdgeIndex < 0)
                {
```

```
                    candidateEdges.Insert(~currentEdgeIndex, currentEdge);
                }
                else
                {
                    if (!candidateEdges.Contains(currentEdge))
                    {
                        candidateEdges.Insert(currentEdgeIndex, currentEdge);
                    }
                }
            }
        }

        private void FindClosestPairOfPointsByBruteForceOnImage()
        {
            CirclePair closestCirclePair = new CirclePair();
            double possibleMinimumDistance, minimumDistance = -1;

            bruteForceOnImageStopwatch = Stopwatch.StartNew();

            for (int i = 0; i < circleList.Count - 1; i++)
            {
                for (int j = i + 1; j < circleList.Count; j++)
                {
                    Thread.Sleep(1);

                    possibleMinimumDistance = EuclideanDistance(circleList[i].Centre,
circleList[j].Centre);

                    if (possibleMinimumDistance < minimumDistance || minimumDistance
== -1)
                    {
                        closestCirclePair.OriginCircle = circleList[i];
                        closestCirclePair.DestinationCircle = circleList[j];

                        minimumDistance = possibleMinimumDistance;
                    }
                }
            }

            bruteForceOnImageStopwatch.Stop();

            /**imageGraphics = Graphics.FromImage(closestPairOfPointsImage);**/

            DrawLine(closestCirclePair.OriginCircle.Centre,
closestCirclePair.DestinationCircle.Centre, Color.LawnGreen, 20);

            DrawCircle(closestCirclePair.OriginCircle.Centre, 30, Color.LawnGreen);
            DrawCircle(closestCirclePair.DestinationCircle.Centre, 30,
Color.LawnGreen);

            PictureBoxImage.Refresh();

            LabelBruteForceOnImageDistanceValue.Text =
EuclideanDistance(closestCirclePair.OriginCircle.Centre,
closestCirclePair.DestinationCircle.Centre).ToString();
        }

        private void FindClosestPairOfPointsByBruteForceOnGraph()
        {
            CirclePair closestCirclePair = new CirclePair();
            double possibleMinimumDistance, minimumDistance = -1;
```

```
            bruteForceOnGraphStopwatch = Stopwatch.StartNew();

            foreach (Vertex currentVertex in circleGraph.VertexList)
            {
                foreach (Edge currentEdge in currentVertex.EdgeList)
                {
                    if (currentEdge.Destination.VertexCircle.ID >
currentVertex.VertexCircle.ID)
                    {
                        Thread.Sleep(1);

                        possibleMinimumDistance = currentEdge.Weight;

                        if (possibleMinimumDistance < minimumDistance ||
minimumDistance == -1)
                        {
                            closestCirclePair.OriginCircle =
currentVertex.VertexCircle;
                            closestCirclePair.DestinationCircle =
currentEdge.Destination.VertexCircle;

                            minimumDistance = possibleMinimumDistance;
                        }
                    }
                }
            }

            bruteForceOnGraphStopwatch.Stop();

            /**imageGraphics = Graphics.FromImage(closestPairOfPointsImage);**/

            DrawLine(closestCirclePair.OriginCircle.Centre,
closestCirclePair.DestinationCircle.Centre, Color.Fuchsia, 10);

            DrawCircle(closestCirclePair.OriginCircle.Centre, 20, Color.Fuchsia);
            DrawCircle(closestCirclePair.DestinationCircle.Centre, 20, Color.Fuchsia);

            PictureBoxImage.Refresh();

            LabelBruteForceOnGraphDistanceValue.Text =
EuclideanDistance(closestCirclePair.OriginCircle.Centre,
closestCirclePair.DestinationCircle.Centre).ToString();
        }

        private void FindClosestPairOfPointsByDivideAndConquerOnImage()
        {
            List<Point> graphPoints = new List<Point>();
            Point[] closestPairOfPoints = new Point[2];

            foreach (Circle graphCircle in circleList)
            {
                graphPoints.Add(graphCircle.Centre);
            }

            graphPoints.Sort((pointOne, pointTwo) =>
pointOne.X.CompareTo(pointTwo.X));

            divideAndConquerOnImageStopwatch = Stopwatch.StartNew();

            closestPairOfPoints = DivideAndConquerGraphPoints(graphPoints);

            divideAndConquerOnImageStopwatch.Stop();
```

```
            /**imageGraphics = Graphics.FromImage(closestPairOfPointsImage);**/

            DrawLine(closestPairOfPoints[0], closestPairOfPoints[1], Color.Yellow, 3);

            DrawCircle(closestPairOfPoints[0], 10, Color.Yellow);
            DrawCircle(closestPairOfPoints[1], 10, Color.Yellow);

            PictureBoxImage.Refresh();

            LabelDivideAndConquerOnImageDistanceValue.Text =
EuclideanDistance(closestPairOfPoints[0], closestPairOfPoints[1]).ToString();
        }

        private Point[] DivideAndConquerGraphPoints(List<Point> graphPoints)
        {
            Point[] closestPairOfPoints;
            double minimumDistance = double.PositiveInfinity, possibleMinimumDistance;

            // C
            if (graphPoints.Count <= 4)
            {
                closestPairOfPoints = new Point[2];

                for (int i = 0; i < graphPoints.Count - 1; i++)
                {
                    for (int j = i + 1; j < graphPoints.Count; j++)
                    {
                        Thread.Sleep(1);

                        possibleMinimumDistance = EuclideanDistance(graphPoints[i],
graphPoints[j]);

                        if (possibleMinimumDistance < minimumDistance)
                        {
                            closestPairOfPoints[0] = graphPoints[i];
                            closestPairOfPoints[1] = graphPoints[j];

                            minimumDistance = possibleMinimumDistance;
                        }
                    }
                }

                return closestPairOfPoints;
            }

            List<Point> dGraphPoints = new List<Point>();
            Point[] leftSideClosestPairOfPoints, rightSideClosestPairOfPoints;
            Point middlePoint;
            int pointsCount, middle = graphPoints.Count / 2;
            double closestPairOfPointsDistance;

            middlePoint = graphPoints[middle];

            leftSideClosestPairOfPoints =
DivideAndConquerGraphPoints(graphPoints.GetRange(0, middle));

            if (graphPoints.Count % 2 == 0)
            {
                pointsCount = middle - 1;
            }
            else
```

```
            {
                pointsCount = middle;
            }

            rightSideClosestPairOfPoints =
DivideAndConquerGraphPoints(graphPoints.GetRange(middle + 1, pointsCount));

            if (EuclideanDistance(leftSideClosestPairOfPoints[0],
leftSideClosestPairOfPoints[1]) < EuclideanDistance(rightSideClosestPairOfPoints[0],
rightSideClosestPairOfPoints[1]))
            {
                closestPairOfPoints = leftSideClosestPairOfPoints;
            }
            else
            {
                closestPairOfPoints = rightSideClosestPairOfPoints;
            }

            closestPairOfPointsDistance = EuclideanDistance(closestPairOfPoints[0],
closestPairOfPoints[1]);

            // O(n)
            for (int i = 0; i < graphPoints.Count; i++)
            {
                if (Math.Abs(graphPoints[i].X - middlePoint.X) <
closestPairOfPointsDistance)
                {
                    dGraphPoints.Add(graphPoints[i]);
                }
            }

            // O(nlog(n))
            dGraphPoints.Sort((pointOne, pointTwo) =>
pointOne.Y.CompareTo(pointTwo.Y));

            minimumDistance = closestPairOfPointsDistance;

            // O(6n)
            for (int i = 0; i < dGraphPoints.Count - 1; i++)
            {
                for (int j = i + 1; j < dGraphPoints.Count; j++)
                {
                    if (dGraphPoints[j].Y - dGraphPoints[i].Y < minimumDistance)
                    {
                        Thread.Sleep(1);

                        possibleMinimumDistance = EuclideanDistance(dGraphPoints[i],
dGraphPoints[j]);

                        if (possibleMinimumDistance < minimumDistance)
                        {
                            closestPairOfPoints[0] = dGraphPoints[i];
                            closestPairOfPoints[1] = dGraphPoints[j];

                            minimumDistance = possibleMinimumDistance;
                        }
                    }
                }
            }

            return closestPairOfPoints;
        }
```

```
        private bool BreadthFirstSearch(Vertex originVertex)
        {
            List<Vertex> visitedVertexes = new List<Vertex>() { originVertex };
            List<bool> treeVertexesNewLevel = new List<bool>() { false };
            Queue<Vertex> vertexQueue = new Queue<Vertex>();
            Vertex currentVertex, destinationVertex;
            int treeVertexesIndex = 0, vertexesAdded = 0;

            vertexQueue.Enqueue(originVertex);

            // O(N)
            while (vertexQueue.Count != 0)
            {
                currentVertex = vertexQueue.Dequeue();

                // O(N - 1)
                foreach (Edge currentEdge in currentVertex.EdgeList)
                {
                    destinationVertex = currentEdge.Destination;

                    if (!visitedVertexes.Contains(destinationVertex))
                    {
                        treeByBreadthPoints.Add(new Point[2] {
currentVertex.VertexCircle.Centre, destinationVertex.VertexCircle.Centre });

                        visitedVertexes.Add(destinationVertex);
                        vertexQueue.Enqueue(destinationVertex);

                        vertexesAdded++;

                        if (!treeVertexesNewLevel[treeVertexesIndex])
                        {
                            treeVertexesNewLevel[treeVertexesIndex] = true;
                        }
                    }
                }

                if (treeVertexesIndex != 0)
                {
                    if (treeVertexesNewLevel[treeVertexesIndex - 1] !=
treeVertexesNewLevel[treeVertexesIndex])
                    {
                        return false;
                    }
                }

                treeVertexesIndex++;

                if (treeVertexesIndex == treeVertexesNewLevel.Count)
                {
                    treeVertexesNewLevel.Clear();

                    for (int i = 0; i < vertexesAdded; i++)
                    {
                        treeVertexesNewLevel.Add(false);
                    }

                    vertexesAdded = 0;
                    treeVertexesIndex = 0;
                }
            }
```

```
            return true;
        }

        private void ResetToolsAndAbstractDataTypes()
        {
            PictureBoxImage.BackgroundImage = null;

            // Main Graph
            circleGraph = null;

            circleList.Clear();
            circlePairList.Clear();

            TreeViewGraphDetails.Nodes.Clear();

            // Graph Tab
            TabControlCapabilities.SelectedIndex = (int)Tabs.Graph;

            // Points Tab
            if (closestPairOfPointsImage != null)
            {
                imageGraphics = Graphics.FromImage(closestPairOfPointsImage);
                imageGraphics.Clear(Color.Transparent);
            }

            ButtonFindClosestPairOfPoints.Enabled = false;

            LabelBruteForceOnImageExecutionTimeValue.Text = "0 ms";
            LabelBruteForceOnImageDistanceValue.Text = "0";
            LabelBruteForceOnGraphExecutionTimeValue.Text = "0 ms";
            LabelBruteForceOnGraphDistanceValue.Text = "0";
            LabelDivideAndConquerOnImageExecutionTimeValue.Text = "0 ms";
            LabelDivideAndConquerOnImageDistanceValue.Text = "0";

            // Random Tab
            if (agentsWithRandomPathImage != null)
            {
                imageGraphics = Graphics.FromImage(agentsWithRandomPathImage);
                imageGraphics.Clear(Color.Transparent);
            }

            TimerAgentsWithRandomPathMovement.Enabled = false;

            TreeViewAgentsWithRandomPathDetails.Nodes.Clear();

            NumericUpDownAgentsWithRandomPath.Value =
NumericUpDownAgentsWithRandomPath.Minimum;
            NumericUpDownAgentsWithRandomPath.Enabled = false;

            ButtonCreateAgentsWithRandomPath.Enabled = false;

            // Trees Tab
            if (MSTImage != null)
            {
                imageGraphics = Graphics.FromImage(MSTImage);
                imageGraphics.Clear(Color.Transparent);
            }

            if (MSTAgentsImage != null)
            {
                imageGraphics = Graphics.FromImage(MSTAgentsImage);
```

```
    imageGraphics.Clear(Color.Transparent);
}

TimerMSTAgentsMovement.Enabled = false;

TreeViewMSTAgentDetails.Nodes.Clear();

NumericUpDownMSTAgents.Value = NumericUpDownMSTAgents.Minimum;
NumericUpDownMSTAgents.Enabled = false;

RadioButtonKruskal.Checked = true;
RadioButtonKruskal.Enabled = false;
RadioButtonPrim.Enabled = false;

CheckBoxRandomOrigins.Checked = false;
CheckBoxRandomOrigins.Enabled = false;

ButtonGenerateMST.Enabled = false;
ButtonCreateMSTAgents.Enabled = false;

LabelPrimMSTTotalWeightValue.Text = "0";
LabelKruskalMSTTotalWeightValue.Text = "0";
LabelNumberOfMSTSubgraphsValue.Text = "0";

// Preys Tab
if (preysAndPredatorsImage != null)
{
    imageGraphics = Graphics.FromImage(preysAndPredatorsImage);
    imageGraphics.Clear(Color.Transparent);
}

TimerPreysAndPredatorsMovement.Enabled = false;

NumericUpDownPreys.Value = NumericUpDownPreys.Minimum;
NumericUpDownPreys.Enabled = false;
NumericUpDownPredators.Value = NumericUpDownPredators.Minimum;
NumericUpDownPredators.Enabled = false;

ButtonAddPreys.Enabled = false;
ButtonAddPredators.Enabled = false;
ButtonStart.Enabled = false;

// Breadth Tab
if (BFSTreeImage != null)
{
    imageGraphics = Graphics.FromImage(BFSTreeImage);
    imageGraphics.Clear(Color.Transparent);
}

ButtonGenerateTreeByBreadth.Enabled = false;

/**if (PictureBoxImage.BackgroundImage != null)
{
    PictureBoxImage.BackgroundImage = null;
}

if (TimerAgentMovement.Enabled)
{
    TimerAgentMovement.Enabled = false;
}

ButtonFindClosestPairOfPoints.Enabled = false;
```

```
            ButtonGenerateMST.Enabled = false;
            ButtonCreateMSTAgents.Enabled = false;

            NumericUpDownPredators.Value = NumericUpDownPredators.Minimum;
            NumericUpDownPredators.Enabled = false;
            ButtonAddPredators.Enabled = false;
            ButtonAddPreys.Enabled = false;
            ButtonStart.Enabled = false;

            ButtonGenerateTreeByBreadth.Enabled = false;

            TreeViewGraphDetails.Nodes.Clear();
            TreeViewMSTAgentDetails.Nodes.Clear();

            LabelNumberOfMSTSubgraphs.Text = "0";
            LabelKruskalMSTTotalWeightNumber.Text = "0";
            LabelPrimMSTTotalWeightNumber.Text = "0";

            NumericUpDownAgentsWithRandomPath.Value =
NumericUpDownAgentsWithRandomPath.Minimum;
            NumericUpDownAgentsWithRandomPath.Enabled = false;
            ButtonCreateAgentsWithRandomPath.Enabled = false;
            TimerAgentsWithRandomPathMovement.Enabled = false;
            TreeViewAgentsWithRandomPathDetails.Nodes.Clear();

            MSTAgents.Clear();
            NumericUpDownMSTAgents.Value = NumericUpDownMSTAgents.Minimum;
            NumericUpDownMSTAgents.Enabled = false;

            globalPreys.Clear();
            NumericUpDownPreys.Value = NumericUpDownPreys.Minimum;
            NumericUpDownPreys.Enabled = false;

            if (circleGraph != null)
            {
                circleGraph = null;
            }

            if (kruskalMST != null)
            {
                kruskalMST = null;
            }

            if (primMST != null)
            {
                primMST = null;
            }

            if (dijkstraGraph != null)
            {
                dijkstraGraph = null;
            }

            circleList.Clear();
            circlePairList.Clear();
            globalPredators.Clear();**/
        }

        private void ProcessBitmap()
        {
            Point currentCoordinates = new Point();
```

```
            for (currentCoordinates.Y = 0; currentCoordinates.Y <
processingImage.Height; currentCoordinates.Y++)
            {
                for (currentCoordinates.X = 0; currentCoordinates.X <
processingImage.Width; currentCoordinates.X++)
                {
                    if (processingImage.GetPixel(currentCoordinates.X,
currentCoordinates.Y).ToArgb() == Color.Black.ToArgb())
                    {
                        FindCircleCentre(ref currentCoordinates);
                    }
                }
            }
        }

        private void FindCircleCentre(ref Point currentCoordinates)
        {
            bool validCircleCentreSearch = true;
            int lastBlackPixelOnX = currentCoordinates.X + 1, middlePointOnX = 0,
lastBlackPixelOnY = currentCoordinates.Y + 1, middlePointOnY = 0;

            try
            {
                while (processingImage.GetPixel(lastBlackPixelOnX,
currentCoordinates.Y).ToArgb() == Color.Black.ToArgb())
                {
                    lastBlackPixelOnX++;
                }

                lastBlackPixelOnX--;

                middlePointOnX = (lastBlackPixelOnX + currentCoordinates.X) / 2;

                currentCoordinates.X = lastBlackPixelOnX;

                if (processingImage.GetPixel(middlePointOnX, currentCoordinates.Y -
1).ToArgb() == Color.Black.ToArgb())
                {
                    validCircleCentreSearch = false;
                }
                else
                {
                    while (processingImage.GetPixel(middlePointOnX,
lastBlackPixelOnY).ToArgb() == Color.Black.ToArgb())
                    {
                        lastBlackPixelOnY++;
                    }

                    lastBlackPixelOnY--;

                    middlePointOnY = (lastBlackPixelOnY + currentCoordinates.Y) / 2;
                }
            }
            catch (ArgumentOutOfRangeException)
            {
                validCircleCentreSearch = false;
            }

            if (validCircleCentreSearch)
            {
                Point circleCentre = new Point(middlePointOnX, middlePointOnY);
```

```
                int circleRadius = (lastBlackPixelOnY - currentCoordinates.Y) / 2;

                if (IsFullCircle(circleCentre, circleRadius))
                {
                    Circle newCircle = new Circle(circleList.Count + 1, circleCentre,
circleRadius);

                    circleList.Add(newCircle);
                }
            }

            currentCoordinates.X = lastBlackPixelOnX;
        }

        private bool IsFullCircle(Point circleCentre, int circleRadius)
        {
            try
            {
                if (processingImage.GetPixel(circleCentre.X - circleRadius - 2,
circleCentre.Y).ToArgb() == Color.Black.ToArgb())
                {
                    return false;
                }

                if (processingImage.GetPixel(circleCentre.X + circleRadius + 2,
circleCentre.Y).ToArgb() == Color.Black.ToArgb())
                {
                    return false;
                }

                if (processingImage.GetPixel(circleCentre.X, circleCentre.Y -
circleRadius - 2).ToArgb() == Color.Black.ToArgb())
                {
                    return false;
                }

                if (processingImage.GetPixel(circleCentre.X, circleCentre.Y +
circleRadius + 2).ToArgb() == Color.Black.ToArgb())
                {
                    return false;
                }
            }
            catch (ArgumentOutOfRangeException)
            {
                return false;
            }

            return true;
        }

        private void DrawCircleCentres()
        {
            Circle currentCircle;

            /**imageGraphics = Graphics.FromImage(processingImage);**/

            for (int i = 0; i < circleList.Count; i++)
            {
                currentCircle = circleList[i];

                DrawCircle(currentCircle.Centre, 8, Color.DarkOrange);
```

```
                imageGraphics.DrawString((i + 1).ToString(), new Font("Arial", 20),
new SolidBrush(Color.Black),
                    currentCircle.Centre.X - currentCircle.Radius - 20,
currentCircle.Centre.Y - currentCircle.Radius - 20);
            }

            PictureBoxImage.BackgroundImage = null;
            PictureBoxImage.BackgroundImage = processingImage;
        }

        private void DrawCircle(Point circleCentre, float circleDiameter, Color
circleColor)
        {
            float circleRadius = circleDiameter / 2;

            imageGraphics.FillEllipse(new SolidBrush(circleColor), circleCentre.X -
circleRadius, circleCentre.Y - circleRadius, circleDiameter, circleDiameter);
        }

        private void SetValidCirclePairs()
        {
            List<Point> linePoints;
            CirclePair validCirclePair;

            for (int i = 0; i < circleList.Count - 1; i++)
            {
                for (int j = i + 1; j < circleList.Count; j++)
                {
                    linePoints = new List<Point>();

                    if (ValidDrawLine(circleList[i].Centre, circleList[j].Centre,
linePoints))
                    {
                        validCirclePair = new CirclePair(circleList[i], circleList[j],
linePoints);

                        circlePairList.Add(validCirclePair);
                    }
                }
            }
        }

        private bool ValidDrawLine(Point originCentre, Point destinationCentre,
List<Point> linePoints)
        {
            Point currentPixel, previousPixel, nextPixel;
            bool obstructionDetected = true, reversedOrigin = false;
            int originY, destinationY, originX, destinationX, numberOfObstructions =
1;

            if (originCentre.X == destinationCentre.X)
            {
                if (originCentre.Y < destinationCentre.Y)
                {
                    originY = originCentre.Y + 1;
                    destinationY = destinationCentre.Y;
                }
                else
                {
                    originY = destinationCentre.Y + 1;
                    destinationY = originCentre.Y;
```

```
                    reversedOrigin = true;
                }

                while (originY < destinationY)
                {
                    previousPixel = new Point(originCentre.X, originY - 1);
                    currentPixel = new Point(originCentre.X, originY);
                    nextPixel = new Point(originCentre.X, originY + 1);

                    if (ItsObstruction(previousPixel, currentPixel, nextPixel, ref
obstructionDetected, ref numberOfObstructions))
                    {
                        return false;
                    }

                    linePoints.Add(currentPixel);

                    originY++;
                }
            }
            else
            {
                float m = (float)(destinationCentre.Y - originCentre.Y) /
(destinationCentre.X - originCentre.X);
                float b = originCentre.Y - (m * originCentre.X);

                if (m > 1 || m < -1)
                {
                    if (originCentre.Y < destinationCentre.Y)
                    {
                        originY = originCentre.Y + 1;
                        destinationY = destinationCentre.Y;
                    }
                    else
                    {
                        originY = destinationCentre.Y + 1;
                        destinationY = originCentre.Y;

                        reversedOrigin = true;
                    }

                    while (originY < destinationY)
                    {
                        previousPixel = new Point((int)Math.Round((originY - 1 - b) /
m), originY - 1);
                        currentPixel = new Point((int)Math.Round((originY - b) / m),
originY);
                        nextPixel = new Point((int)Math.Round((originY + 1 - b) / m),
originY + 1);

                        if (ItsObstruction(previousPixel, currentPixel, nextPixel, ref
obstructionDetected, ref numberOfObstructions))
                        {
                            return false;
                        }

                        linePoints.Add(currentPixel);

                        originY++;
                    }
                }
                else
```

```
                {
                    if (originCentre.X < destinationCentre.X)
                    {
                        originX = originCentre.X + 1;
                        destinationX = destinationCentre.X;
                    }
                    else
                    {
                        originX = destinationCentre.X + 1;
                        destinationX = originCentre.X;

                        reversedOrigin = true;
                    }

                    while (originX < destinationX)
                    {
                        previousPixel = new Point(originX - 1, (int)Math.Round((m *
(originX - 1)) + b));
                        currentPixel = new Point(originX, (int)Math.Round((m *
originX) + b));
                        nextPixel = new Point(originX + 1, (int)Math.Round((m *
(originX + 1)) + b));

                        if (ItsObstruction(previousPixel, currentPixel, nextPixel, ref
obstructionDetected, ref numberOfObstructions))
                        {
                            return false;
                        }

                        linePoints.Add(currentPixel);

                        originX++;
                    }
                }
            }

            if (reversedOrigin)
            {
                linePoints.Reverse();
            }

            return true;
        }

        private bool ItsObstruction(Point previousPixel, Point currentPixel, Point
nextPixel, ref bool obstructionDetected, ref int numberOfObstructions)
        {
            if (numberOfObstructions > 2)
            {
                return true;
            }

            List<Point> firstSide = new List<Point>(), secondSide = new List<Point>();
            Point surroundingPixel = new Point(currentPixel.X - 1, currentPixel.Y -
1), startingPixel = surroundingPixel;
            int movementCase = (int)SurroundingPixelMovement.UpperSide, listNumber =
(int)SideLists.FirstSide;

            do
            {
                if (surroundingPixel == previousPixel || surroundingPixel ==
nextPixel)
```

```
        {
            if (listNumber == (int)SideLists.FirstSide)
            {
                listNumber = (int)SideLists.SecondSide;
            }
            else
            {
                listNumber = (int)SideLists.FirstSide;
            }
        }
        else
        {
            if (listNumber == (int)SideLists.FirstSide)
            {
                firstSide.Add(surroundingPixel);
            }
            else
            {
                secondSide.Add(surroundingPixel);
            }
        }

        switch (movementCase)
        {
            case (int)SurroundingPixelMovement.UpperSide:

                surroundingPixel.X++;

                if (surroundingPixel.X == currentPixel.X + 1)
                {
                    movementCase++;
                }

                break;

            case (int)SurroundingPixelMovement.RightSide:

                surroundingPixel.Y++;

                if (surroundingPixel.Y == currentPixel.Y + 1)
                {
                    movementCase++;
                }

                break;

            case (int)SurroundingPixelMovement.LowerSide:

                surroundingPixel.X--;

                if (surroundingPixel.X == currentPixel.X - 1)
                {
                    movementCase++;
                }

                break;

            case (int)SurroundingPixelMovement.LeftSide:
                surroundingPixel.Y--;
                break;
        }
    }
```

```
            while (surroundingPixel != startingPixel);

            if (obstructionDetected)
            {
                if (ValidCentreAndSurroundings(currentPixel, firstSide, secondSide))
                {
                    obstructionDetected = false;
                }
            }
            else
            {
                if (!ValidCentreAndSurroundings(currentPixel, firstSide, secondSide))
                {
                    numberOfObstructions++;

                    obstructionDetected = true;
                }
            }

            return false;
        }

        private bool ValidCentreAndSurroundings(Point currentPixel, List<Point>
firstSide, List<Point> secondSide)
        {
            if (processingImage.GetPixel(currentPixel.X, currentPixel.Y).ToArgb() !=
Color.White.ToArgb())
            {
                return false;
            }

            foreach (Point firstSidePoint in firstSide)
            {
                if (processingImage.GetPixel(firstSidePoint.X,
firstSidePoint.Y).ToArgb() != Color.White.ToArgb())
                {
                    foreach (Point secondSidePoint in secondSide)
                    {
                        if (processingImage.GetPixel(secondSidePoint.X,
secondSidePoint.Y).ToArgb() != Color.White.ToArgb())
                        {
                            return false;
                        }
                    }
                }
            }

            return true;
        }

        private void UpdateGraphDetails()
        {
            int currentVertexNumber = 0, currentEdgeNumber;

            TreeViewGraphDetails.BeginUpdate();

            foreach (Vertex currentVertex in circleGraph.VertexList)
            {
                TreeViewGraphDetails.Nodes.Add(currentVertex.ToString());

                currentEdgeNumber = 0;
```

```
                foreach (Edge currentEdge in currentVertex.EdgeList)
                {

TreeViewGraphDetails.Nodes[currentVertexNumber].Nodes.Add(currentEdge.Destination.ToSt
ring());

TreeViewGraphDetails.Nodes[currentVertexNumber].Nodes[currentEdgeNumber].Nodes.Add(cur
rentEdge.Weight.ToString());

                    currentEdgeNumber++;
                }

                currentVertexNumber++;
            }

            TreeViewGraphDetails.EndUpdate();
        }

        private void UpdateMSTAgentsDetails()
        {
            Agent currentAgent;
            string agentPath;

            TreeViewMSTAgentDetails.BeginUpdate();

            for (int i = 0; i < MSTAgents.Count; i++)
            {
                currentAgent = MSTAgents[i];

                TreeViewMSTAgentDetails.Nodes.Add(currentAgent.ID.ToString());

                agentPath =
currentAgent.VisitedEdgeList[0].Origin.VertexCircle.ID.ToString();

                foreach (Edge currentEdge in currentAgent.VisitedEdgeList)
                {
                    agentPath += " → " +
currentEdge.Destination.VertexCircle.ID.ToString();
                }

                TreeViewMSTAgentDetails.Nodes[i].Nodes.Add(agentPath);
            }

            TreeViewMSTAgentDetails.EndUpdate();

            /*string agentPath;

            TreeViewMSTAgentDetails.BeginUpdate();

            TreeViewMSTAgentDetails.Nodes.Add(globalAgent.ID.ToString());

            agentPath =
globalAgent.VisitedEdgeList[0].Origin.VertexCircle.ID.ToString();

            foreach (Edge currentEdge in globalAgent.VisitedEdgeList)
            {
                agentPath += " → " +
currentEdge.Destination.VertexCircle.ID.ToString();
            }

            TreeViewMSTAgentDetails.Nodes[0].Nodes.Add(agentPath);
```

```
            TreeViewMSTAgentDetails.EndUpdate();*/
        }

        private void UpdateAgentsWithRandomPathDetails()
        {
            string agentPath;
            int currentAgentNumber = 0, agentWithTheMostVisitedVertexesIndex = 0;

            TreeViewAgentsWithRandomPathDetails.BeginUpdate();

            foreach (Agent currentAgent in agentsWithRandomPath)
            {

TreeViewAgentsWithRandomPathDetails.Nodes.Add(currentAgent.ID.ToString() + ".
Vértices: " + currentAgent.VisitedVertexList.Count + ". Distancia: " +
currentAgent.GetDistanceTraveled());

                agentPath =
currentAgent.VisitedEdgeList[0].Origin.VertexCircle.ID.ToString();

                foreach (Edge currentEdge in currentAgent.VisitedEdgeList)
                {
                    agentPath += " → " +
currentEdge.Destination.VertexCircle.ID.ToString();
                }


TreeViewAgentsWithRandomPathDetails.Nodes[currentAgentNumber].Nodes.Add(agentPath);

                currentAgentNumber++;

                if (currentAgent.VisitedVertexList.Count >
agentsWithRandomPath[agentWithTheMostVisitedVertexesIndex].VisitedVertexList.Count)
                {
                    agentWithTheMostVisitedVertexesIndex = currentAgent.ID - 1;
                }
                else if (currentAgent.VisitedVertexList.Count ==
agentsWithRandomPath[agentWithTheMostVisitedVertexesIndex].VisitedVertexList.Count)
                {
                    if (currentAgent.GetDistanceTraveled() >
agentsWithRandomPath[agentWithTheMostVisitedVertexesIndex].GetDistanceTraveled())
                    {
                        agentWithTheMostVisitedVertexesIndex = currentAgent.ID - 1;
                    }
                }
            }


TreeViewAgentsWithRandomPathDetails.Nodes[agentWithTheMostVisitedVertexesIndex].Text
+= " ← Mayor.";

            TreeViewAgentsWithRandomPathDetails.EndUpdate();
        }

        private void DrawCircleConnections()
        {
            /**imageGraphics = Graphics.FromImage(processingImage);**/

            foreach (CirclePair currentCirclePair in circlePairList)
            {
                DrawLine(currentCirclePair.OriginCircle.Centre,
currentCirclePair.DestinationCircle.Centre, Color.OrangeRed, 2);
```

```
            }

            PictureBoxImage.BackgroundImage = null;
            PictureBoxImage.BackgroundImage = processingImage;
        }

        private void DrawLine(Point originPoint, Point destinationPoint, Color
lineColor, float penWidth)
        {
            imageGraphics.DrawLine(new Pen(lineColor, penWidth), originPoint,
destinationPoint);
        }

        private double EuclideanDistance(Point originPoint, Point destinationPoint)
        {
            return Math.Sqrt(Math.Pow(destinationPoint.X - originPoint.X, 2) +
Math.Pow(destinationPoint.Y - originPoint.Y, 2));
        }
    }
}
```

Apéndice H. Form MSTSolutionSet

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Final_Project
{
    partial class FormMSTSolutionSet : Form
    {
        public FormMSTSolutionSet(List<Edge> solutionSet, string algorithm)
        {
            InitializeComponent();

            Text += algorithm;

            DataGridViewSolutionSet.DataSource = solutionSet;

            DataGridViewSolutionSet.Columns[3].Visible = false;
        }

        private void ButtonContinue_Click(object sender, EventArgs e)
        {
            DialogResult = DialogResult.OK;
        }
    }
}
```

Apéndice I. Clase Predator

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Final_Project
{
```

```
    class Predator
    {
        // CONSTANTS
        private const int EdgePositionIncrement = 20;

        // ATTRIBUTES
        public Vertex OriginVertex { get; set; }
        public Queue<Edge> Path { get; set; }
        public Edge CurrentEdge { get; set; }
        public int EdgePosition { get; set; }

        public bool HuntingMode { get; set; }
        public List<Prey> HuntedPreys { get; set; }

        // METHODS
        public Predator(int newID, Vertex newOriginVertex)
        {
            OriginVertex = newOriginVertex;
            Path = new Queue<Edge>();
            HuntingMode = false;
            HuntedPreys = new List<Prey>();
        }

        public void GeneratePath()
        {
            Stack<Vertex> vertexStack = new Stack<Vertex>();
            Stack<Edge> edgeStack = new Stack<Edge>();
            List<Vertex> visitedVertexes = new List<Vertex>();
            Queue<Edge> returnEdges = new Queue<Edge>();

            vertexStack.Push(OriginVertex);
            visitedVertexes.Add(OriginVertex);

            DepthFirstSearch(vertexStack, visitedVertexes, edgeStack, returnEdges);

            CurrentEdge = Path.Dequeue();
            CurrentEdge.NumberOfPredators++;
        }

        // (N - 1)(N - 2) = N^2 - 2N - N + 2 = N^2 - 3N + 2 = O(N^2)
        private void DepthFirstSearch(Stack<Vertex> vertexStack, List<Vertex>
visitedVertexes, Stack<Edge> edgeStack, Queue<Edge> returnEdges)
        {
            Vertex currentVertex = vertexStack.Peek();

            foreach (Edge currentVertexEdge in currentVertex.EdgeList)
            {
                if (!visitedVertexes.Contains(currentVertexEdge.Destination))
                {
                    if (returnEdges.Count != 0)
                    {
                        while (returnEdges.Count != 0)
                        {
                            Path.Enqueue(returnEdges.Dequeue());
                        }
                    }

                    Path.Enqueue(currentVertexEdge);

                    visitedVertexes.Add(currentVertexEdge.Destination);
                    vertexStack.Push(currentVertexEdge.Destination);
```

```
                    DepthFirstSearch(vertexStack, visitedVertexes, edgeStack,
returnEdges);
                }
                else
                {
                    if (vertexStack.Count != 1)
                    {
                        if (vertexStack.ElementAt(1) == currentVertexEdge.Destination)
                        {
                            edgeStack.Push(currentVertexEdge);
                        }
                    }
                }
            }

            if (edgeStack.Count != 0)
            {
                returnEdges.Enqueue(edgeStack.Pop());
            }

            vertexStack.Pop();
        }

        public void UpdateEdgePosition()
        {
            if (Path.Count == 0 && EdgePosition == CurrentEdge.LinePoints.Count - 1)
            {
                return;
            }

            EdgePosition += EdgePositionIncrement;

            if (EdgePosition >= CurrentEdge.LinePoints.Count)
            {
                CurrentEdge.NumberOfPredators--;

                if (Path.Count != 0)
                {
                    CurrentEdge = Path.Dequeue();
                    CurrentEdge.NumberOfPredators++;

                    if (CurrentEdge.Preys.Count != 0)
                    {
                        HuntingMode = true;
                    }
                    else if (CurrentEdge.Reverse().Preys.Count != 0)
                    {
                        HuntingMode = true;
                    }
                    else
                    {
                        if (HuntingMode)
                        {
                            HuntingMode = false;
                        }
                    }

                    /*if (CurrentEdge.Preys.Count != 0)
                    {
                        HuntingMode = true;

                        HuntingModeCase = (int)HuntingModeType.SameEdge;
```

```
                    }
                    else if (CurrentEdge.Reverse().Preys.Count != 0)
                    {
                        HuntingMode = true;

                        HuntingModeCase = (int)HuntingModeType.ReverseEdge;
                    }
                    else
                    {
                        if (HuntingMode)
                        {
                            HuntingMode = false;
                        }
                    }*/

                    EdgePosition = 0;
                }
                else
                {
                    EdgePosition = CurrentEdge.LinePoints.Count - 1;
                }
            }
        }

        // The name is pure rubbish, I should change it later
        // Can't get my G if he's in a safe zone
        public bool TryHunting()
        {
            if (CurrentEdge.Preys.Count == 0 && CurrentEdge.Reverse().Preys.Count ==
0)
            {
                HuntingMode = false;

                return false;
            }

            foreach (Prey currentPrey in CurrentEdge.Preys)
            {
                if (EdgePosition >= currentPrey.EdgePosition)
                {
                    HuntedPreys.Add(currentPrey);
                }
            }

            foreach (Prey currentPrey in CurrentEdge.Reverse().Preys)
            {
                if (CurrentEdge.LinePoints.Count - 1 - EdgePosition <=
currentPrey.EdgePosition)
                {
                    HuntedPreys.Add(currentPrey);
                }
            }

            if (HuntedPreys.Count != 0)
            {
                return true;
            }

            return false;
        }
    }
}
```

Apéndice J. Clase Prey

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Final_Project
{
    class Prey
    {
        // INTERNAL CLASS
        public class DijkstraElement
        {
            // ATTRIBUTES
            public Vertex IDVertex { get; set; }
            public double AccumulatedDistance { get; set; }
            public DijkstraElement PreviousDijkstraElement { get; set; }

            // METHODS
            public DijkstraElement(Vertex newIDVertex)
            {
                IDVertex = newIDVertex;

                AccumulatedDistance = double.PositiveInfinity;
            }
        }

        class CompareDijkstraElementsByAccumulatedDistance :
IComparer<DijkstraElement>
        {
            public int Compare(DijkstraElement dijkstraElementOne, DijkstraElement
dijkstraElementTwo)
            {
                if (dijkstraElementOne.AccumulatedDistance >
dijkstraElementTwo.AccumulatedDistance)
                {
                    return 1;
                }
                else if (dijkstraElementOne.AccumulatedDistance <
dijkstraElementTwo.AccumulatedDistance)
                {
                    return -1;
                }
                else
                {
                    return 0;
                }
            }
        }

        // CONSTANTS
        private const int EdgePositionIncrement = 20;

        // ATTRIBUTES
        public int ID { get; set; }
        public bool IsActive { get; set; }
        public Vertex OriginVertex { get; set; }
        public Queue<Edge> Path { get; set; }
        public Edge CurrentEdge { get; set; }
        public int EdgePosition { get; set; }
        public List<DijkstraElement> DijkstraElements { get; set; }
```

```
        public bool OnSafeZone { get; set; }
        public bool SurvivalMode { get; set; }

        // METHODS
        public Prey()
        {
            IsActive = false;
            Path = new Queue<Edge>();
            EdgePosition = 0;
            DijkstraElements = new List<DijkstraElement>();

            OnSafeZone = true;
            SurvivalMode = false;
        }

        public Prey(int newID, Vertex newOriginVertex)
        {
            ID = newID;
            OriginVertex = newOriginVertex;

            IsActive = false;
            Path = new Queue<Edge>();
            EdgePosition = 0;
            DijkstraElements = new List<DijkstraElement>();

            OnSafeZone = true;
            SurvivalMode = false;
        }

        public void GeneratePath(List<Vertex> graphVertexes, Vertex destinationVertex)
        {
            DijkstraElement pathDijkstraElement;
            Edge pathEdge;

            GetShortestPathsWithDijkstra(graphVertexes, destinationVertex);

            pathDijkstraElement = DijkstraElements.Find(DijkstraElement =>
DijkstraElement.IDVertex.Equals(destinationVertex));

            do
            {
                pathEdge =
pathDijkstraElement.PreviousDijkstraElement.IDVertex.EdgeList.Find(Edge =>
Edge.Destination.Equals(pathDijkstraElement.IDVertex));

                if (pathEdge != null)
                {
                    Path.Enqueue(pathEdge);

                    pathDijkstraElement = pathDijkstraElement.PreviousDijkstraElement;
                }
            }
            while (!pathDijkstraElement.IDVertex.Equals(OriginVertex));

            Path = new Queue<Edge>(Path.Reverse());

            CurrentEdge = Path.Dequeue();
        }

        // N + log(N) + N(N^2) = N + log(N) + N^3 = O(N^3)
```

```
        private void GetShortestPathsWithDijkstra(List<Vertex> graphVertexes, Vertex
destinationVertex)
        {
            int definitiveDijkstraElements = 0, originVertexIndex =
OriginVertex.VertexCircle.ID - 1;

            foreach (Vertex currentVertex in graphVertexes)
            {
                DijkstraElements.Add(new DijkstraElement(currentVertex));
            }

            DijkstraElements[originVertexIndex].AccumulatedDistance = 0;
            DijkstraElements[originVertexIndex].PreviousDijkstraElement =
DijkstraElements[originVertexIndex];

            RelocateDijkstraElement(originVertexIndex);

            while (definitiveDijkstraElements != DijkstraElements.Count)
            {
                if (DijkstraElements[definitiveDijkstraElements].AccumulatedDistance
== double.PositiveInfinity || DijkstraElements[definitiveDijkstraElements].IDVertex ==
destinationVertex)
                {
                    break;
                }


UpdateDijkstraElements(DijkstraElements[definitiveDijkstraElements++]);
            }
        }

        // (N - 1)(N + log(N)) = (N^2 + Nlog(N) - N - log(N)) = O(N^2)
        private void UpdateDijkstraElements(DijkstraElement definitiveDijkstraElement)
        {
            int destinationIndex;

            foreach (Edge currentVertexEdge in
definitiveDijkstraElement.IDVertex.EdgeList)
            {
                destinationIndex = DijkstraElements.FindIndex(DijkstraElement =>
DijkstraElement.IDVertex == currentVertexEdge.Destination);

                if (currentVertexEdge.Weight +
definitiveDijkstraElement.AccumulatedDistance <
DijkstraElements[destinationIndex].AccumulatedDistance)
                {
                    DijkstraElements[destinationIndex].AccumulatedDistance =
currentVertexEdge.Weight + definitiveDijkstraElement.AccumulatedDistance;
                    DijkstraElements[destinationIndex].PreviousDijkstraElement =
definitiveDijkstraElement;

                    RelocateDijkstraElement(destinationIndex);
                }
            }
        }

        // log(N)
        private void RelocateDijkstraElement(int index)
        {
            DijkstraElement temporalDijkstraElement = DijkstraElements[index];
            int newIndex;
```

```
            DijkstraElements.RemoveAt(index);

            newIndex = DijkstraElements.BinarySearch(temporalDijkstraElement, new
CompareDijkstraElementsByAccumulatedDistance());

            if (newIndex < 0)
            {
                DijkstraElements.Insert(~newIndex, temporalDijkstraElement);
            }
            else
            {
                DijkstraElements.Insert(newIndex, temporalDijkstraElement);
            }
        }

        public void UpdateEdgePosition()
        {
            // Check if there's a predator on the current Edge
            if (OnSafeZone)
            {
                if (CurrentEdge.NumberOfPredators != 0)
                {
                    return;
                }
                else
                {
                    if (CurrentEdge.Reverse().NumberOfPredators != 0)
                    {
                        return;
                    }
                }

                OnSafeZone = false;
                CurrentEdge.Preys.Add(this);
            }

            // Check if a Predator appeared, backtrack on the same Edge if it did
            if (CurrentEdge.Reverse().NumberOfPredators != 0)
            {
                SurvivalMode = true;
            }

            if (SurvivalMode)
            {
                EdgePosition -= EdgePositionIncrement;

                if (EdgePosition <= 0)
                {
                    CurrentEdge.Preys.Remove(this);

                    OnSafeZone = true;
                    SurvivalMode = false;
                }
            }
            else
            {
                EdgePosition += EdgePositionIncrement;

                if (EdgePosition >= CurrentEdge.LinePoints.Count)
                {
                    CurrentEdge.Preys.Remove(this);
```

```
                    if (Path.Count != 0)
                    {
                        CurrentEdge = Path.Dequeue();

                        OnSafeZone = true;

                        EdgePosition = 0;
                    }
                    else
                    {
                        EdgePosition = CurrentEdge.LinePoints.Count - 1;

                        IsActive = false;
                    }
                }
            }
        }
    }
}
```

Apéndice K. Form PreyVertexSelection

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Final_Project
{
    partial class FormPreyVertexSelection : Form
    {
        public FormPreyVertexSelection(List<Vertex> originVertexes, List<Vertex>
destinationVertexes, string preyID)
        {
            InitializeComponent();

            LabelPrey.Text += preyID;

            ComboBoxOriginVertex.DataSource = originVertexes;
            ComboBoxDestinationVertex.DataSource = destinationVertexes;
        }

        private void ButtonAccept_Click(object sender, EventArgs e)
        {
            DialogResult = DialogResult.OK;
        }

        private void ButtonCancel_Click(object sender, EventArgs e)
        {
            DialogResult = DialogResult.Cancel;
        }

        public int GetSelectedOriginVertex()
        {
            return ComboBoxOriginVertex.SelectedIndex;
        }

        public int GetSelectedDestinationVertex()
        {
```

```
            return ComboBoxDestinationVertex.SelectedIndex;
        }
    }
}
```

Apéndice L. Clase Program

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Final_Project
{
    static class Program
    {
        /// <summary>
        /// Punto de entrada principal para la aplicación.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new FormFinalProject());
        }
    }
}
```

Apéndice M. Clase Vertex

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Final_Project
{
    class Vertex
    {
        private Circle vertexCircle;
        private List<Edge> edgeList;

        public Vertex()
        {
            vertexCircle = new Circle();
            edgeList = new List<Edge>();
        }

        public Vertex(Circle newVertexCircle)
        {
            vertexCircle = newVertexCircle;
            edgeList = new List<Edge>();
        }

        public Circle VertexCircle
        {
            get { return vertexCircle; }
            set { vertexCircle = value; }
        }

        public List<Edge> EdgeList
        {
```

```
            get { return edgeList; }
            set { edgeList = value; }
        }

        public List<Edge> GetEdges()
        {
            List<Edge> allEdges = new List<Edge>();

            foreach (Edge currentEdge in edgeList)
            {
                allEdges.Add(currentEdge);
            }

            return allEdges;
        }

        public override string ToString()
        {
            return vertexCircle.ID.ToString() + ' ' + vertexCircle.Centre.ToString();
        }

        public override bool Equals(object obj)
        {
            if (obj == null || GetType() != obj.GetType())
            {
                return false;
            }

            Vertex compareVertex = (Vertex)obj;

            if (vertexCircle.ID == compareVertex.VertexCircle.ID)
            {
                return true;
            }

            return false;
        }

        public override int GetHashCode()
        {
            return vertexCircle.GetHashCode();
        }
    }
}
```

Apéndice N. Form VertexSelection

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Final_Project
{
    partial class FormVertexSelection : Form
    {
        public FormVertexSelection(List<Vertex> vertexList, string vertexLabel)
        {
            InitializeComponent();
```

```
            Text += vertexLabel;
            LabelVertex.Text += vertexLabel + ":";

            ComboBoxVertexes.DataSource = vertexList;
        }

        private void ButtonAccept_Click(object sender, EventArgs e)
        {
            DialogResult = DialogResult.OK;
        }

        private void ButtonCancel_Click(object sender, EventArgs e)
        {
            DialogResult = DialogResult.Cancel;
        }

        public int GetSelectedVertex()
        {
            return ComboBoxVertexes.SelectedIndex;
        }
    }
}
```