PRÁCTICA 1

Máster en Ingeniería Informática

UNIVERSIDAD DE ALMERIA

COMPUTACIÓN AVANZADA: TÉCNICAS

COMPUTACIÓN EVOLUTIVA

Curso 2015/2016

Alumno/a:

Savíns Puertas Martín

EDITADO CON LaTeX.

Plantilla original de Mathias Legrand (http://www.latextemplates.com).
Modificada por Savíns Puertas Martín.

Índice general

1 Introducción a la práctica 7

1.1 Especificación de la práctica 7

1.2 Lenguaje de programación a utilizar 7

1.3 Resumen del resto de la memoria 8

2 Elementos del problema 9

2.1 Mapa 9

2.1.1 Restricciones 9

2.1.2 Representación 10

2.2 Robot 11

2.2.1 Características 11

2.2.2 Inteligencia 12

3 Algoritmo 13

3.1 Fases del algoritmo 13

3.1.1 Generación 13

3.1.2 Evaluación 13

3.1.3 Ordenación 14

3.1.4 Selección 14

3.1.5 Mutación 15

3.1.6 Cruce 15

3.1.7 Eliminación 17

3.2 Algoritmo principal 17

4 Interfaz gráfica 21
4.1 Interfaz gráfica 21
4.2 Cargar mapa 22
4.3 Aprender mediante un mapa 23
4.4 Cargar inteligencia desde un archivo 24
4.5 Ejecutar un individuo con inteligencia 24
4.6 Cambiar el diseño del mapa 26

5 Tiempos y Resultados 29
5.1 Mapa Fácil 30
5.2 Mapa Medio 31
5.3 Mapa Difícil 32
5.4 Algunos movimientos autoaprendidos 33
5.5 Vídeo 35

6 Conclusión 37

A Comportamiento mejor robot 39

B Ampliación explicación código fuente 43
B.1 Fases del algoritmo 43
B.1.1 Generación 43
B.1.2 Evaluación 43
B.1.3 Ordenación 44
B.1.4 Selección 44
B.1.5 Mutación 45
B.1.6 Cruce 45
B.1.7 Eliminación 48

1. Introducción a la práctica

En este capítulo se define el problema que se va a resolver en esta práctica. Adicionalmente, se detallará la tecnología utilizada en la solución propuesta así como los motivos de su elección. Por último se resumirá el contenido del resto de capítulos de esta memoria.

1.1 Especificación de la práctica

El problema que se plantea en esta práctica es construir un robot con inteligencia heredada que sea capaz de recorrer el perímetro interno de un polígono. Para resolver el problema se debe utilizar un algoritmo evolutivo. En ese sentido, el conocimiento de los individuos más óptimos será el que pase a las generaciones siguientes.

El polígono sobre el que se probará el robot se definirá en un archivo de texto pudiendo cargar cualquier mapa independiente de la forma.

Una vez entrenado el robot, se podrá almacenar su conocimiento para cargarlo en cualquier momento y poder ejecutarlo sobre un mapa.

Se debe poder visualizar el recorrido del robot en el mapa mediante una aplicación gráfica.

1.2 Lenguaje de programación a utilizar

Durante la realización de la práctica se han utilizado tres lenguajes de programación. Sin embargo, los problemas presentados han obligado a desarrollar la aplicación y el algoritmo en el lenguaje *C#*. Los motivos se describen a continuación.

Inicialmente se resolvió el problema con *Python*. La facilidad de programación que ofrece este lenguaje así como la limpieza de su código ayudaban a una implementación rápida. Sin embargo, al ser un lenguaje interpretado encontrar la mejor solución llevaba demasiado tiempo: aproximadamente 9 horas. Por tanto, se decidió reescribir el problema en C++.

Este lenguaje es más complicado de implementar pero su principal ventaja es la rapidez en la ejecución. El problema llegó en el momento de añadirle la interfaz gráfica. Para ello se hizo uso del *framework* Qt así como su IDE. Tras alguna prueba fallida con su oportuna excepción, al intentar ejecutar de nuevo el software, Windows había bloqueado esa aplicación y no permitía ejecutarla salvo que se reiniciase el equipo. Por tanto, no se podía trabajar en esas condiciones.

Llegados a este punto, se decidió buscar otro lenguaje de programación similar en eficiencia a C++ pero con la facilidad de programación de *Python* y así es como se decidió elegir *C#*. Si bien no se ha podido comprobar empíricamente, según distintos *benchmark*, *C#* tenía un rendimiento ligeramente inferior a C++ por lo que servía para solucionar nuestro problema. Además, para la interfaz gráfica se podía utilizar los *Windows Forms* facilitando bastante el trabajo.

1.3 Resumen del resto de la memoria

El resto de este documento se estructura de la siguiente forma:

En el Capítulo 2 se describen los elementos que forman parte del problema: el mapa y el robot. En este capítulo se justificará la forma de definir el mapa así como las restricciones asociadas al diseño del mismo. El robot será tratado de forma similar.

En el Capítulo 3 se describirá el algoritmo que da solución a este problema. En ese sentido primero se detallarán los submétodos y una vez entendidos se mostrará el algoritmo completo.

En el Capítulo 4 se mostrará la interfaz gráfica propuesta para la práctica. Para ello se realizará un pequeño tutorial explicando como ejecutar las distintas funcionalidades del programa.

En el Capítulo 5 se realizará una comparativa en tiempos de ejecución, mapas y parámetros del algoritmo.

Por último, en el Capítulo 6 se expondrán las conclusiones obtenidas de la realización de esta práctica.

2. Elementos del problema

En este capítulo se el espacio de búsqueda sobre el que se moverá el robot. Además se describirá a éste y sus posibles movimientos.

2.1 Mapa

El espacio que intentará recorrer el robot se encuentra definido por un polígono cerrado. Este polígono se encuentra constituido por celdas. Las celdas pueden ser de dos tipos:

- De muro: no se puede situar el robot sobre ella y por tanto actúan de pared (Izq. Fig 2.1).
- Vacía: el robot se puede situar sobre ella libremente (Der. Fig 2.1).

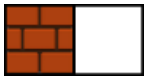


Figura 2.1: Tipos de casillas.

2.1.1 Restricciones

Para el problema no sirve cualquier configuración de mapa sino que hay definidas algunas restricciones. Estas restricciones son:

- El mapa debe ser cerrado. En caso contrario el robot saldría por el exterior del mapa y nunca se encontraría la mejor solución (Figura 2.2).
- No pueden existir celdas de muro en medio del mapa. Podría existir la posibilidad de que el robot llegase a ellas y no saliese de dicha isla (Figura 2.3).

- No pueden existir caminos de una celda de anchura (Figura 2.4).

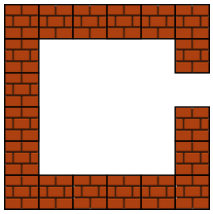


Figura 2.2: Mapa abierto.

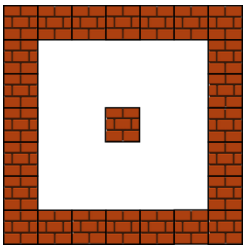


Figura 2.3: Mapa con elementos en el centro.

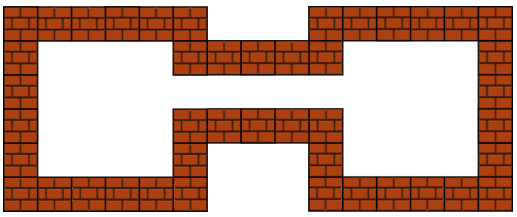


Figura 2.4: Mapa con camino de una celda.

2.1.2 Representación

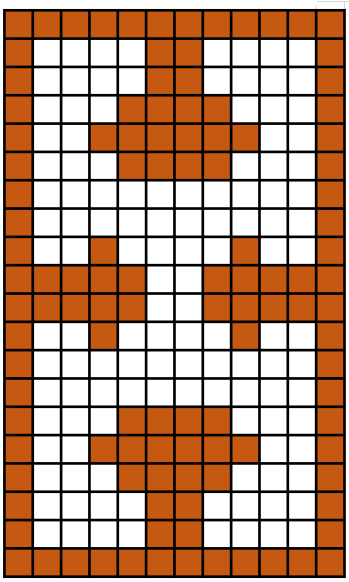
Son dos las formas en las que se representa el mapa. Por un lado está la representación en un archivo de texto que se utilizará para que nuestro algoritmo conozca la configuración del mapa. Por otro lado, para facilitar el trabajo al usuario que utilice la aplicación se representará gráficamente.

Para la representación en un archivo de texto se utilizarán dos caracteres para definir los dos tipos del celdas posibles. Por un lado, las celdas que correspondan a un muro se representarán mediante #. Las celdas que representen una celda vacía harán lo propio con un espacio.

Esta configuración la leerá el programa y la representará en un panel en la interfaz gráfica. Un ejemplo de mapa en un archivo de texto y su representación gráfica se puede ver en la Figura 2.5.

```
############
#     ##     #
#     ##     #
#    ####    #
#   ######   #
#    ####    #
#            #
#            #
#   #     #  #
#####   #####
#####   #####
#   #     #  #
#            #
#            #
#    ####    #
#   ######   #
#    ####    #
#     ##     #
#     ##     #
############
```

Mediante caracteres.



Representación gráfica.

Figura 2.5: Representación del mapa.

2.2 Robot

Es la unidad elemental del problema. Dado que se está trabajando en un algoritmo evolutivo, a lo largo de la memoria también puede que se haga referencia a él con el nombre de individuo (de una especie).

2.2.1 Características

Su tamaño es de una celda del mapa.

Como individuo, recibe información externa a él. La información que capta es el tipo de las celdas que hay a su alrededor. En concreto 8. Se puede ver en la Figura 2.6.

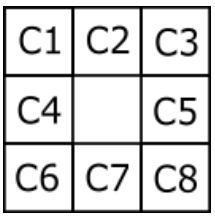


Figura 2.6: Celdas alrededor del robot.

2.2.2 Inteligencia

Dado que hay 8 sensores, el número de posibles entradas es $2^8 = 256$ por tanto para cada una de esas posibilidades el robot debería moverse en un sentido. Hacia arriba, abajo, izquierda y derecha.

La forma de representar esta información que almacena cada robot y que es independiente de cada uno es mediante un diccionario clave-valor. El hecho de utilizar esta estructura se debe a la rapidez de acceso al valor. El formato en el que se almacena cada posible entrada es:

```
Clave: [1,0,0,1,0,1,1,1] - Valor: 1
```

Los 0's significan que la celda se encuentra vacía y los 1's que hay muro. El valor puede tomar uno de cuatro posibles valores: 0, 1, 2 y 3. El significado de cada uno es el siguiente:

- 0: El robot se intenta desplazar una celda hacia arriba.
- 1: El robot se intenta desplazar una celda hacia la derecha.
- 2: El robot se intenta desplazar una celda hacia abajo.
- 3: El robot se intenta desplazar una celda hacia la izquierda.

Una vez dicho esto, gráficamente el ejemplo anterior sería el siguiente (Figura 2.7).

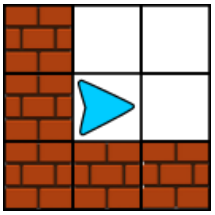


Figura 2.7: Movimiento según clave-valor.

3. Algoritmo

En este capítulo se define el algoritmo que permite mejorar los individuos de la especie. En una primera parte se explicará cada uno de las fases del algoritmo: evaluación, ordenación, selección, cruce, eliminación, mutación. Después se mostrará el pseudocódigo del algoritmo que recoge todas estas operaciones y permite la evolución del robot. De forma adicional en el Anexo B se entrará más en profundidad en la explicación del código fuente.

3.1 Fases del algoritmo

3.1.1 Generación

Este método crea la primera generación de individuos. Se crean tantos como se indique por parámetro de entrada y se almacenan en una lista. Se permite elegir si se desea que todos los individuos se generen en el mismo punto de inicio o por el contrario cada uno tenga un punto distinto. Respecto a su comportamiento, el diccionario se genera de forma aleatoria para cada uno. En ese sentido las posibilidades de que dos individuos tenga el mismo comportamiento es 1 entre 256! ($8x10^{506}$).

3.1.2 Evaluación

Este es uno de los procesos más importantes ya que si no se evalúa o se define correctamente la función objetivo todo lo demás no sirve. Lo que se pretende maximizar es el número de celdas pertenecientes al perímetro interno de mapa. Para realizar este cálculo, en el momento de cargar en memoria el mapa, se calcula de forma dinámica el número de celdas que tiene el perímetro. Por las restricciones del problema, este proceso es sencillo ya que toda celda colindante con una

celda de muro pertenece a la solución.

Por otra parte, cada robot tiene una lista con todos los puntos por donde pasa conforme se va moviendo.

Conociendo ambas partes, se realiza una intersección entre los dos conjuntos de tal forma que solo quedan las celdas que pertenecen a la solución y que ha pasado el robot. De esta forma, la el proceso de evaluación se reduce a contar el número de celdas que ha recorrido el robot.

```
public static List<ETTank> evaluate(List<ETTank> creatures,
   List<Point> mapSolution)
{
    List<ETTank> creaturesTemp = creatures;
    List<Point> pathTravelled = new List<Point>();
    for (int i = 0; i < creaturesTemp.Count; i++)
    {
        pathTravelled = creaturesTemp.ElementAt(i).PathTravelled
        .Intersect(mapSolution).Distinct<Point>().ToList<Point>();
        creaturesTemp.ElementAt(i).PathTravelled = pathTravelled;
    }
    return creaturesTemp;
}
```

Listado 3.1: Método de evaluación.

3.1.3 Ordenación

Este método ordena el listado de individuos en función de la función objetivo. La idea es poder operar con los mejores individuos o con los peores de una forma sencilla.

```
  public static List<ETTank> order(List<ETTank> creatures)
  {
      List<ETTank> creaturesTemp = creatures;
      creaturesTemp = creaturesTemp.OrderByDescending(o =>
   o.PathTravelled.Count).ToList();
      return creaturesTemp;
  }
```

Listado 3.2: Método de ordenación.

3.1.4 Selección

Aquí se eligen a los mejores individuos para el proceso de cruce. En ese sentido se han implementado dos soluciones: elegir los X mejores individuos; o establecer una selección por torneo para así permitir que entre los dos peores resultados, el antepenúltimo sea elegido. Con la primera solución se descartaría directamente.

La solución elegida finalmente es la de la selección de los mejores individuos.

```
public static List<ETTank> select(List<ETTank>
 creatures)
{
    List<ETTank> sublist = creatures.GetRange(0,
     creatures.Count/2);
    return sublist;
}
```

Listado 3.3: Método de selección.

3.1.5 Mutación

Como toda especie en la naturaleza, en ocasiones puede que mute un gen de un individuo. Se ha considerado una probabilidad de 0,05 de que un individuo cambie su comportamiento dada una entrada. El comportamiento cambia aleatoriamente y la entrada elegida es seleccionada de la misma forma.

```
public static List<ETTank> mutation(List<ETTank> creatures)
{
    List<ETTank> creaturesTemp = creatures;
    Random rnd = new Random(DateTime.Now.Millisecond);
    int n = 0;
    string key;
    for (int i = 0; i < creaturesTemp.Count; i++)
    {
        n = rnd.Next(0, 100);
        if (n < 5)
        {
            n = rnd.Next(0, 256);
            key = creaturesTemp.ElementAt(i).Brain.Situations.
            ElementAt(n).Key;
            creaturesTemp.ElementAt(i).Brain.Situations[key] =
            rnd.Next(0, 4);
        }
    }
    return creaturesTemp;
}
```

Listado 3.4: Método de mutación.

3.1.6 Cruce

Este proceso es muy importante ya que es donde se produce el paso de genes de una generación a otra. Dados dos padres seleccionados de los individuos existentes, se elige un punto de división y se mezclan por ese punto los genes de ambos padres generando dos nuevos genes con parte de ambos. Estos dos genes se asignan a dos nuevos hijos.

```
public static List<ETTank> crossing(List<ETTank>
  creatures, Matrix map)
{
    List<ETTank> newGeneration = new List<ETTank>();
    ETTank e1 = null, e2 = null, off1 = null, off2 = null;
    Random rnd = new Random(DateTime.Now.Millisecond);
    IDictionary<string, int> firstHalf, secondHalf,
  firstHalf2, secondHalf2;
    int n = 0;

    while (creatures.Count>0)
    {
        n = rnd.Next(0, creatures.Count);
        e1 = creatures.ElementAt(n);
        creatures.RemoveAt(n);
        n = rnd.Next(0, creatures.Count);
        e2 = creatures.ElementAt(n);
        creatures.RemoveAt(n);
        n = rnd.Next(0, 256);

        off1 = new ETTank(ETTank.InitialPositionMap);
        off2 = new ETTank(ETTank.InitialPositionMap);


        firstHalf =
  e1.Brain.Situations.Take(n).ToDictionary(kv => kv.Key,
  kv => kv.Value);
        secondHalf =
  e1.Brain.Situations.Skip(n).ToDictionary(kv => kv.Key,
  kv => kv.Value);

        firstHalf2 =
  e2.Brain.Situations.Take(n).ToDictionary(kv => kv.Key,
  kv => kv.Value);
        secondHalf2 =
  e2.Brain.Situations.Skip(n).ToDictionary(kv => kv.Key,
  kv => kv.Value);

        secondHalf2.ToList().ForEach(x =>
  firstHalf.Add(x.Key, x.Value));

        secondHalf.ToList().ForEach(x =>
  firstHalf2.Add(x.Key, x.Value));

        off1.Brain.Situations = firstHalf;
        off2.Brain.Situations = firstHalf2;

        newGeneration.Add(off1);
        newGeneration.Add(off2);
    }


    return newGeneration;
}
```

Listado 3.5: Método de cruce.

3.1.7 Eliminación

Si se cruzan constantemente los individuos, la especie no dejaría de crecer. Por tanto es necesario definir un método que elimine aquellos individuos que no interesen. La técnica seguida en este paso ha sido eliminar aquellos con peor valor en la función objetivo.

```
public static List<ETTank> killWorst(List<ETTank> creatures, int
  number)
{
    List<ETTank> sublist = creatures.GetRange(0, number);
    return sublist;
}
```

Listado 3.6: Método de eliminación.

3.2 Algoritmo principal

Una vez ya se conocen las partes del algoritmo, en esa sección se muestra el pseudocódigo y el código en C#. En este último se puede que algunos bucles *for* se paralelizan con la función *Parallel.For* de *C#*.

1: Creación de los individuos
2: Búsqueda de cada individuo de la solución
3: Evaluación de los individuos
4: Ordenación de los individuos
5: **while** Hay evoluciones disponibles **do**
6: Selección de los mejores individuos
7: Cruce de los individuos seleccionados
8: Mutación de los individuos
9: Búsqueda de cada individuo de la solución
10: Evaluación de los individuos
11: Ordenación de los individuos
12: Eliminación de los peores individuos
13: Se guarda el mejor individuo de la evolución para fines de análisis
14: **end while**

```
public void evolution(int numberOfCreatures, int
   numberOfIterations, int numberOfEvolutions, frmMain window,
   bool positionrandom, Point p)
{

    List<ETTank> creatures = new List<ETTank>();
    int n = 0;
    ETTank creature;
    // INICIALIZACION
    if (!positionrandom)
    {
```

```
        while (n < numberOfCreatures)
        {
            creature = new ETTank(p);
            creatures.Add(creature);
            n++;
        }
    }
    else
    {
        while (n < numberOfCreatures)
        {
            creature = new ETTank(window.f_BoardBox.Matrix);
            creatures.Add(creature);
            n++;
        }
    }

    // EVLUACION
    n = 0;
    while (n < numberOfIterations)
    {
        Parallel.For(0, creatures.Count, i =>
        {
            string radar =
                creatures.ElementAt(i).scanner(this.map.m_Buffer);
            int direction =
                creatures.ElementAt(i).Brain.Situations[radar];
            creatures.ElementAt(i).move(direction, this.map);
            });
        n++;
    }

    creatures = ETVenus.evaluate(creatures,
   window.f_BoardBox.tileSolutions);
    creatures = ETVenus.order(creatures);
    n = 0;
    int n2 = 0;
    List<ETTank> creaturesSelected, offsprings;

    while (n < numberOfEvolutions)
    {
        if (creatures.ElementAt(0).PathTravelled.Count ==
            window.f_BoardBox.tileSolutions.Count)
        {
            break;
        }
        //RESET CREATURES
        Parallel.For(0, creatures.Count, i =>
        {
            creatures.ElementAt(i).IndexMapPosition =
                creatures.ElementAt(i).InitialPositionMap;
```

```
        });
        creaturesSelected = ETVenus.select(creatures);
        offsprings = ETVenus.crossing(creaturesSelected, this.map);
        creatures.AddRange(offsprings);
        creatures = ETVenus.mutation(creatures);
        while (n2 < numberOfIterations)
        {
            Parallel.For(0, creatures.Count, i =>
            {
                string radar =
                creatures.ElementAt(i).scanner(this.map.m_Buffer);
                int direction =
                    creatures.ElementAt(i).Brain.Situations[radar];
                creatures.ElementAt(i).move(direction, this.map);
            });
            n2++;
        }
        n2 = 0;
        creatures =  ETVenus.evaluate(creatures,
             window.f_BoardBox.tileSolutions);
        creatures =
            ETVenus.order(creatures);
        creatures =
            ETVenus.killWorst(creatures,
                 (creatures.Count / 3) * 2);
        window.addBestCreatureEvolution(creatures.ElementAt(0), n);

        n++;
        if (n % 100 == 0)
        {
            System.IO.StreamWriter fileTemp = new
             System.IO.StreamWriter(@"C:\maps\evoluciones\solucion"
                 + n + ".txt");
            fileTemp.WriteLine("Distancia recorrida: " +
                 creatures.ElementAt(0).PathTravelled.Count);
            for (int i = 0; i < 256; i++)
            {
                fileTemp.WriteLine(creatures.ElementAt(0).
                    Brain.Situations.ElementAt(i));
            }
            fileTemp.Close();
        }
        window.updateChart(creatures.ElementAt(0).
            PathTravelled.Count);
        Console.WriteLine("Evolucion: " + n);
        }

    //ESCRIBIR LA SOLUCION
    System.IO.StreamWriter file =
        new System.IO.StreamWriter(@"C:\maps\solucion.txt");
    file.WriteLine("Mejores 10 resultados");
```

```
111     for (int i = 0; i < 10; i++)
        {
113         file.WriteLine("Distancia recorrida:" +
                creatures.ElementAt(i).PathTravelled.Count);
115     }

117     file.Close();
        System.IO.StreamWriter file2 = new System.IO.StreamWriter(
119         @"C:\maps\solucionMejor.txt");
        for (int i = 0; i < 256; i++)
121     {
            file2.WriteLine(creatures.ElementAt(0).
123             Brain.Situations.ElementAt(i));
        }
125
        file2.Close();
127
}
```

Listado 3.7: Método principal del algoritmo.

4. Interfaz gráfica

En este capítulo se muestra la interfaz gráfica de la aplicación y los pasos a seguir para ejecutar su funcionalidad.

4.1 Interfaz gráfica

Al iniciar la aplicación, aparece una ventana similar a la de la Figura 4.1. Para explicar cada una de sus partes se ha decidido marcar con cuadros de colores cada una de ellas. Se detallan a continuación:

- Negro: Menú superior. Desde él se puede utilizar las distintas operaciones que hay implementadas como son cargar un mapa, entrenar un robot, probar un robot dada una configuración y cambiar el estilo del mapa.
- Rojo: Tres cuadros de texto para introducir el número de criaturas, de evoluciones y de iteraciones por evolución que se quieren ejecutar.
- Amarillo: Representación gráfica del mapa cargado.
- Verde: Si se desea que la posición inicial de los individuos sea aleatoria e independiente se pulsa el *checkbox*. En caso contrario todos los individuos se generarán en la posición que se indique en los elementos *Row* y *Column*.
- Azul: En este listado aparece la configuración del mejor individuo en cada evolución.
- Rosa: Gráfico que representa en cada evolución, cual es el valor de la función objetivo del mejor individuo.

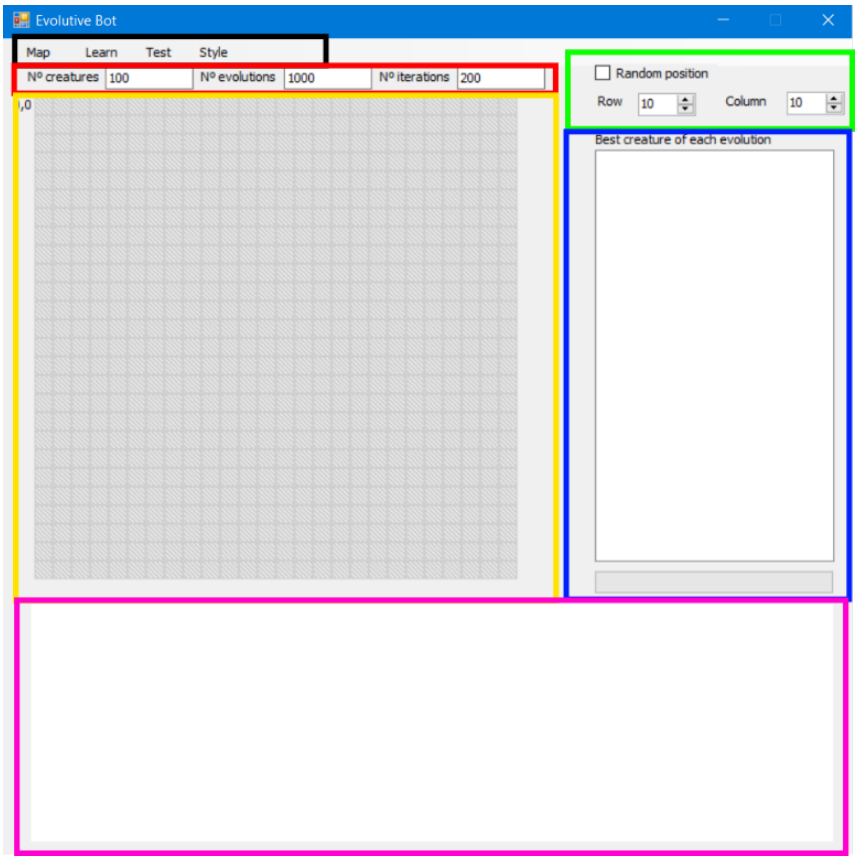


Figura 4.1: Interfaz gráfica.

4.2 Cargar mapa

El el primer paso que se tiene que hacer para utilizar el programa. Para cargarlo hay que clicar sobre *Map* y a continuación *Load* (Figura 4.2).

Figura 4.2: Cargar mapa.

Esto abrirá una ventana con la cual seleccionar el archivo que tiene el mapa. Una vez seleccionado, se cargará en la ventana de la aplicación (Figura 4.3).

Como se indicó en el capítulo del mapa, si la forma del mismo no se encuentra descrita con *almohadillas* y *espacios* el mapa no se cargará y provocará la finalización del programa.

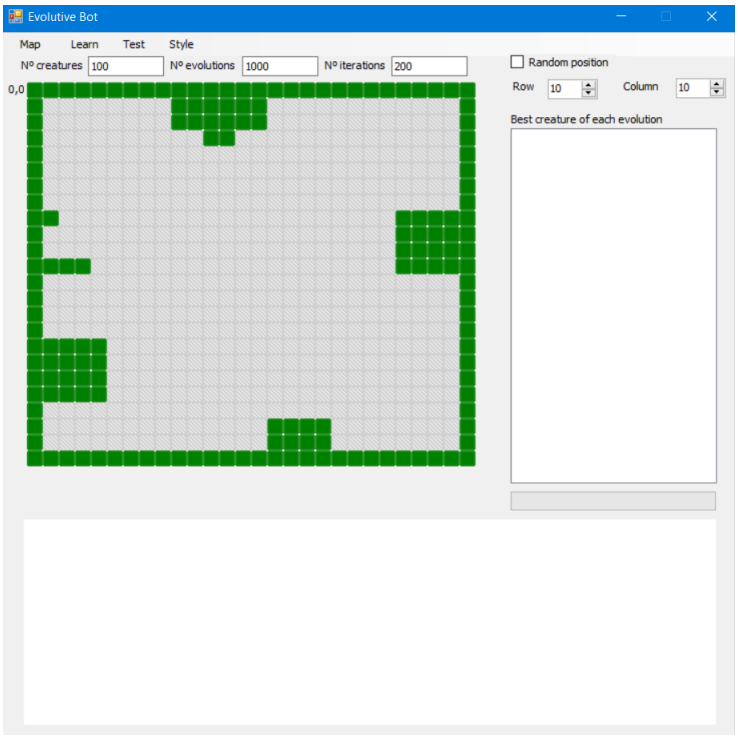


Figura 4.3: Mapa cargado en la aplicación

4.3 Aprender mediante un mapa

Partiendo desde el punto en el que se ha cargado un mapa, el siguiente paso es ajustar los parámetros correspondientes al número de individuos, evoluciones e iteraciones (Figura 4.4).

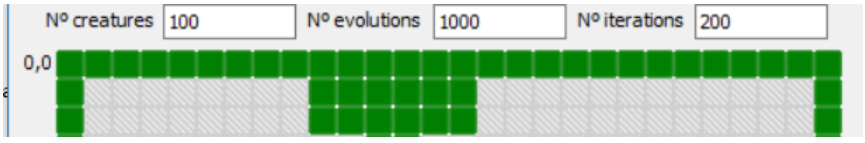


Figura 4.4: Configuración de los parámetros.

Tras esto, para ejecutar el método hay que clicar sobre *Learn* y a continuación en *Start learning*. Entonces empezará a ejecutar el algoritmo evolutivo. En la parte de abajo se muestra una gráfica en la que se indica en cada evolución cual es el valor más alto alcanzado de la función objetivo. Esta gráfica sirve de ayuda porque se sabe en tiempo real cual es la situación de las especies y además la facilidad o dificultad de aprender el mapa. Adicionalmente, en el panel de la derecha se va cargando el mejor individuo de cada evolución.

Por otra parte, en el momento que se desee para el aprendizaje se pulsa en *Learn*, *Stop learning* y se para la ejecución.

El hecho de que se puedan realizar varias acciones de forma paralela se debe a que se han utilizado varios hilos de ejecución mediante la clase *Thread* de *C#*. Un ejemplo de ello se puede en el Listado 4.1.

```
thread = new Thread(() =>
{
    god.evolution(creatures, iterations, evolutions, obj,
  randomCheckBox.Checked, new
  Kernel.Point(Convert.ToByte(rowNumericUpDown.Value),
  Convert.ToByte(columnNumericUpDown.Value)));
});
thread.Start();
```

Listado 4.1: Paralelización de métodos.

Bien porque haya finalizado la ejecución o se haya detenido por parte del usuario, se puede guardar la configuración de los individuos de la lista para poder utilizarla en futuras pruebas u otros mapas. Para guardar dicha configuración hay que seleccionar un individuo, pulsa en *Learn* y por último en *Save creature seleted*. Aparecerá una ventana emergente en la que hay que indicar el fichero donde se guardará la configuración. Tras indicarlo, se pulsa en *Aceptar* y la configuración queda guardada.

4.4 Cargar inteligencia desde un archivo

En caso de haber realizado anteriormente el entrenamiento, si se desea cargar directamente una configuración sobre una especie, hay que clicar sobre *Test* y a continuación *Load Learning*. En ese momento aparecerá en el listado de la derecha el individuo cargado (Figura 4.5).

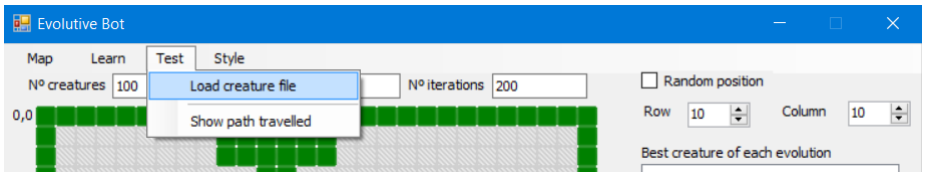


Figura 4.5: Cargar configuración.

4.5 Ejecutar un individuo con inteligencia

Si se desea ejecutar un individuo ya cargado en el panel de la derecha (bien porque se haya cargado desde un archivo o porque haya terminado el aprendizaje), el primer paso es seleccionar el individuo a ejecutar. Después hay que indicar la posición de partida del individuo y por último se pulsa en *Test* y *Show path travelled*. Si se han seguido todos los pasos, aparecerá el recorrido del individuo en el mapa (Figura 4.6). Además se ha añadido una barra de progreso verde para saber cuantas iteraciones le quedan al test.

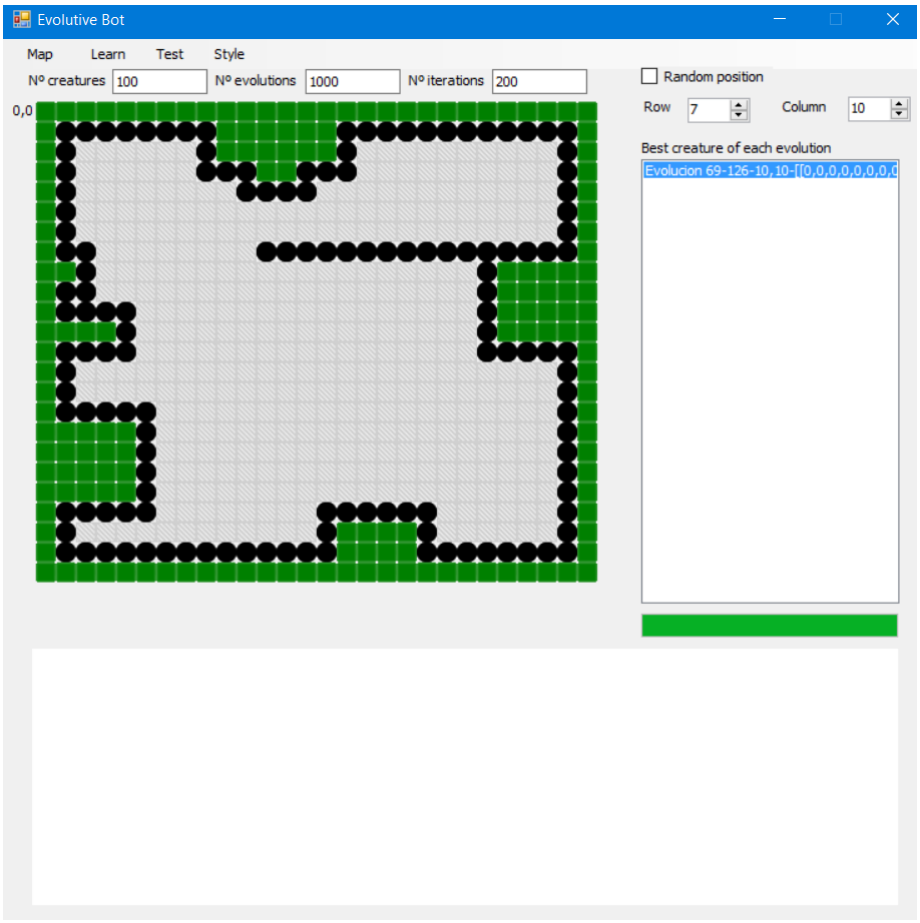


Figura 4.6: Recorrido del individuo.

4.6 Cambiar el diseño del mapa

Como mejora de diseño, se pueden cambiar los colores del mapa. Para ello se pulsa en *Style* y a continuación se elige uno de los tres estilos (Figuras 4.7, 4.8 y 4.9)

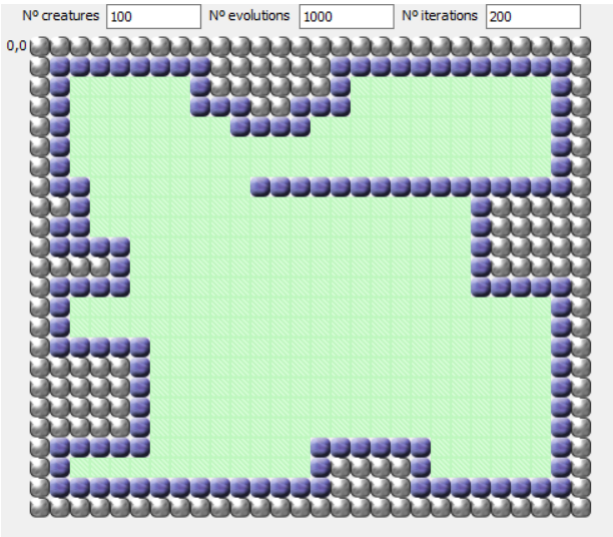


Figura 4.7: Estilo 1.

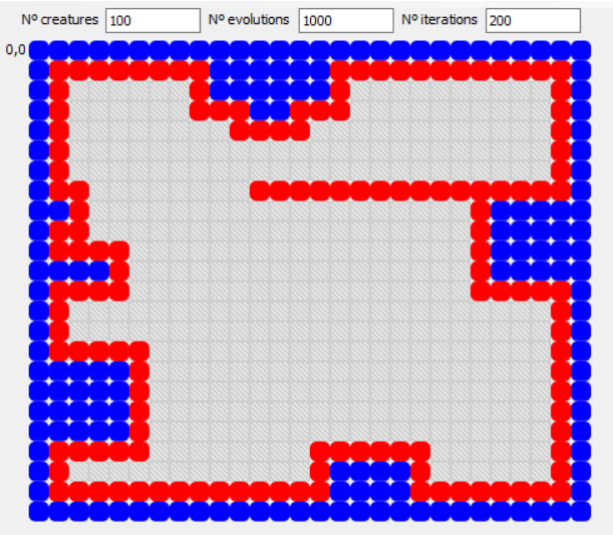


Figura 4.8: Estilo 2.

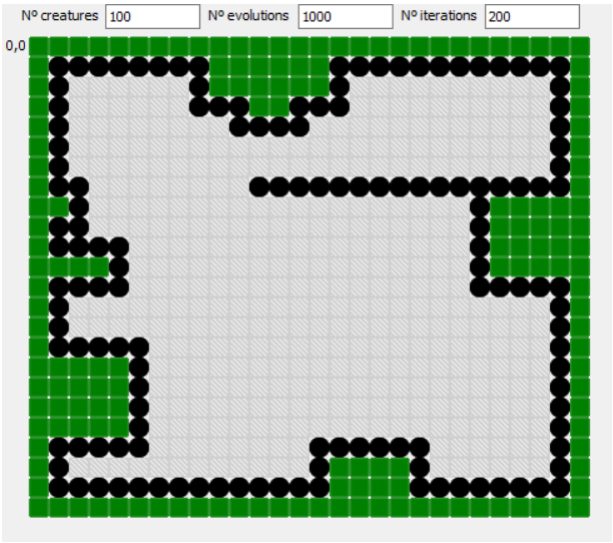


Figura 4.9: Estilo 3 (Por defecto).

5. Tiempos y Resultados

En este capítulo se mostrarán algunos tiempos obtenidos de la ejecución del programa dependiendo de los 4 parámetros que se pueden definir por el usuario: el mapa, el número de especies, el número de evoluciones y el número de iteraciones por evolución. Existiría un quinto parámetro que es la posición inicial de cada robot. Con las pruebas con posición fija, el punto de inicio será 10, 10 y será el mismo para todos los individuos. Si el punto de inicio es aleatorio, no se podrá determinar antes de la ejecución cual es el punto.

En la Figura 5.1 se pueden ver los tres mapas que se van a analizar.

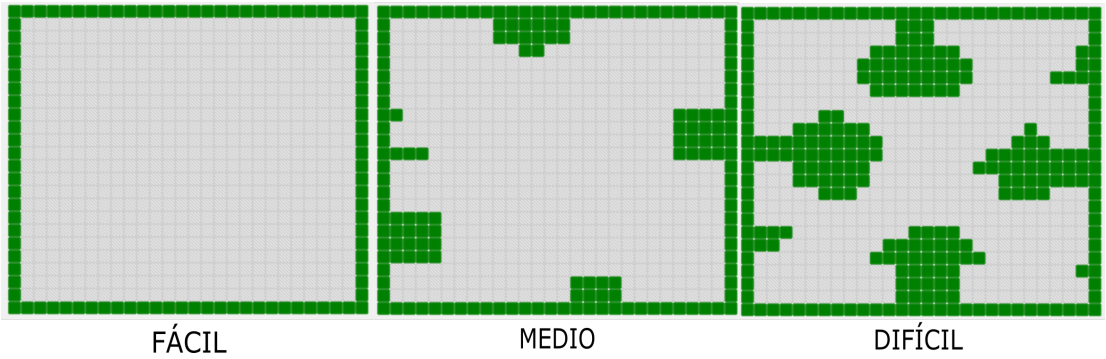


Figura 5.1: Mapas con distinta dificultad.

*Los datos aquí mostrados depende en gran medida del factor suerte ya que es posible que no evolucionen correctamente los individuos debido a que los genes de las generaciones anteriores no transmitan la mejor información, sin embargo esto no se puede controlar. Simplemente se eligen los individuos a priori mejor capacitados.

5.1 Mapa Fácil

Considerando 1000 individuos, 1000 evoluciones y 200 iteraciones por evolución, con 5 evoluciones se encuentra la solución. Además no influye que se generen aleatoriamente los individuos o partan de una posición fija.

Realizando otra prueba con 500 individuos, se han necesitado 10 evoluciones para alcanzar la solución optima.

En cuanto a tiempos de ejecución, en la Tabla 5.1 se puede ver el tiempo de creación de especies, de cada evolución y el tiempo total del problema. En este caso, por la sencillez del problema es más interesante tener menos individuos ya que se opera más rápido y la solución es fácil de obtener.

Tabla 5.1: Tiempos (s) para el mapa fácil

nº de individuos	Tiempo generación	Tiempo evolución	nº evoluciones	Tiempo total
1000	17	5	5	42
500	8	3	10	38

En el listado de la derecha se ve el mejor individuo de cada iteración y seleccionando uno, se puede ver el recorrido en el mapa de la izquierda.

Debajo aparece la evolución por iteraciones. En la primera iteración tenía 64 celdas de la solución final y en la última ya tenía las 92 que conforman la solución.

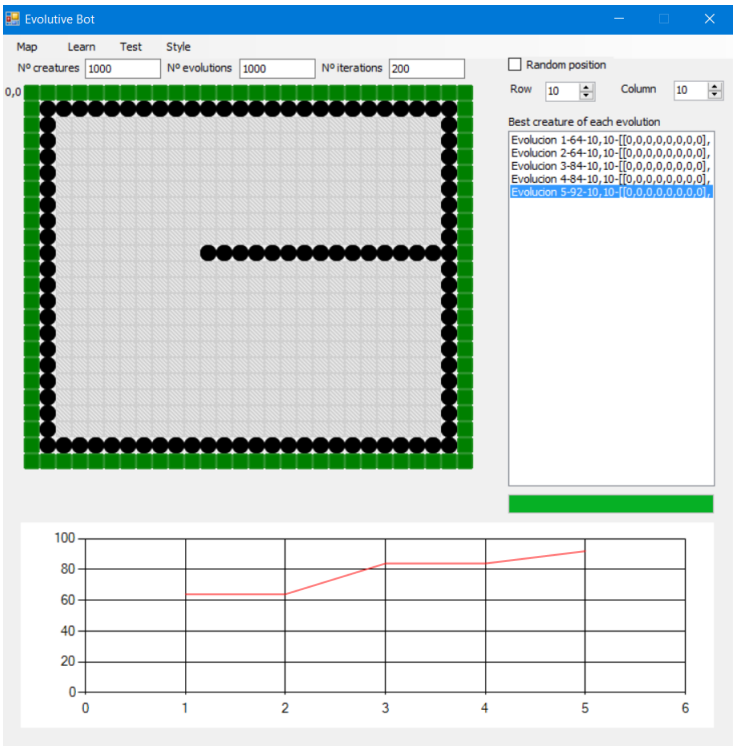


Figura 5.2: Solución con 1000 individuos.

5.2 Mapa Medio

Este mapa presenta una complejidad mayor. Son varias las pruebas que se han realizado y en ellas se puede ver la aleatoriedad de los resultados (Tabla 5.2). En todas ellas se han encontrado la solución final. El segundo y tercer valor corresponde con las Figuras 5.3 y 5.4.

Tabla 5.2: Tiempos (s) para el mapa fácil

Aleatorio	nº de individuos	Tiempo generación	Tiempo evolución	nº evoluciones	Tiempo total (min)
No(10,10)	1000	17	5	69	6
No (10,10)	1000	17	5	206	17
No (10,10)	3000	51	17	93	27
Si	3000	51	17	105	30

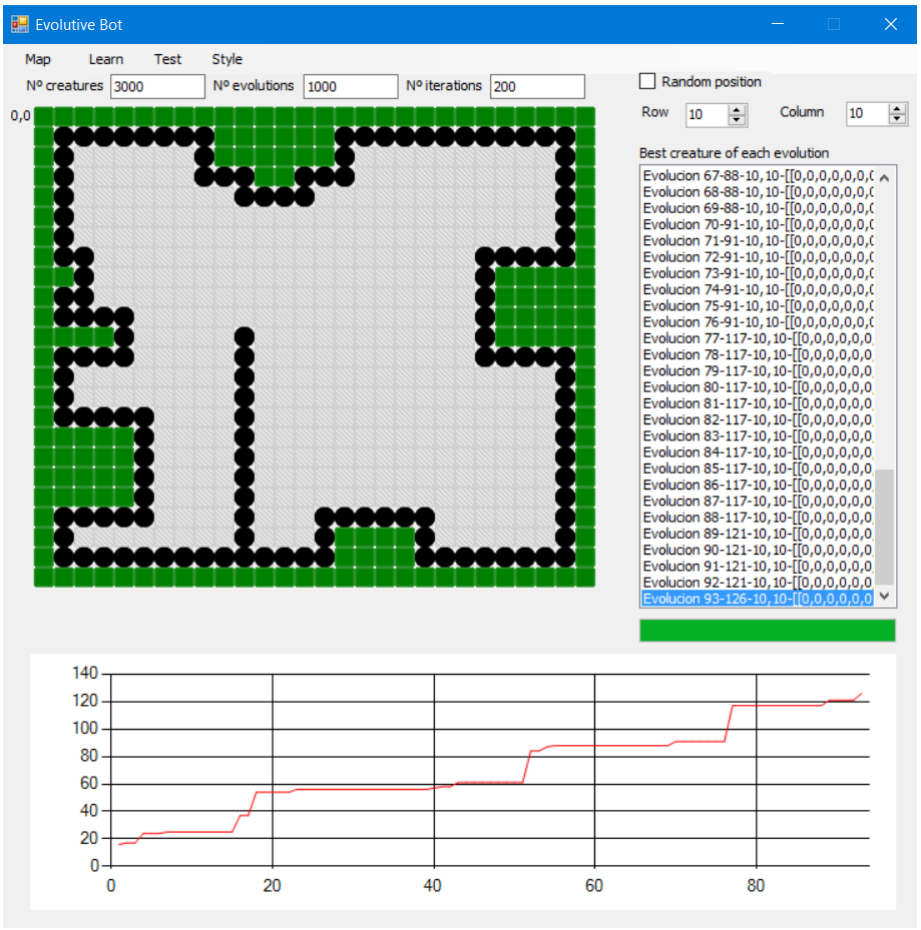


Figura 5.3: Solución con 3000 individuos.

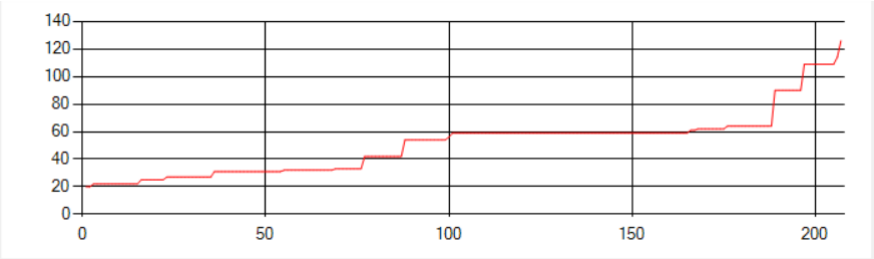


Figura 5.4: Solución con 1000 individuos.

5.3 Mapa Difícil

Este mapa es la versión más difícil planteada. Fueron necesarias 2292 evoluciones en este caso para poder recorrer todo el perímetro (Figura 5.5). En otra ejecución, con 1000 iteraciones no fue suficiente (Figura 5.6). Los tiempos se pueden ver en la Tabla 5.3.

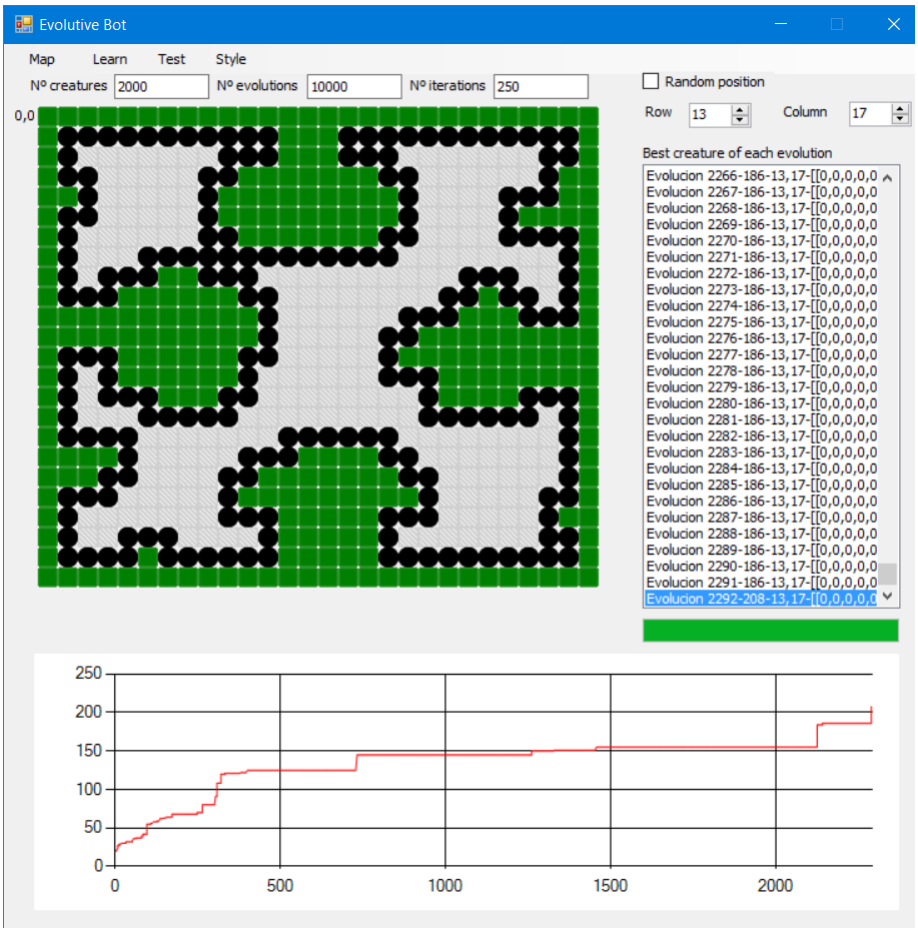


Figura 5.5: Solución con 2000 individuos.

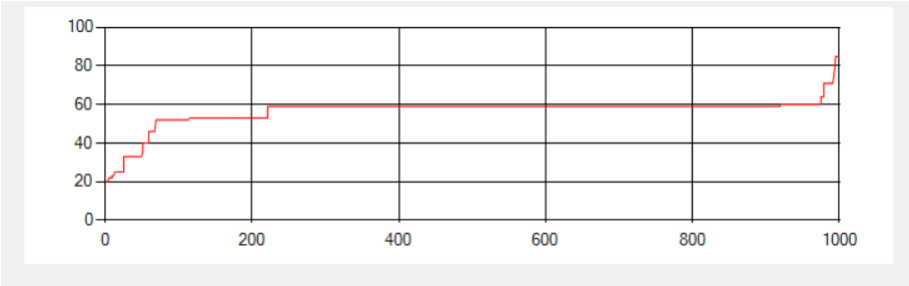


Figura 5.6: Solución incompleta con 2500 individuos.

Tabla 5.3: Tiempos (s) para el mapa fácil

Aleatorio	nº de individuos	Tiempo generación	Tiempo evolución	nº evoluciones	Tiempo total (horas)
Si	2000	35	11	2292	7
No (10,10)	2500	43	14	1000	3.9

En ambos casos se puede ver el problema que se presenta cuando se estanca la especie y no consigue encontrar mejores soluciones. Esta *suerte* puede hacer que se consiga encontrar la solución o por el contrario no se pueda en un tiempo razonable.

En el Anexo A se adjunta el comportamiento de este último robot que encuentra el recorrido en el último mapa y por tanto también es capaz de recorrer los otros dos.

5.4 Algunos movimientos autoaprendidos

En esta sección se mostrarán algunos de los movimientos (Figura 5.7 y 5.8) autoaprendidos por el robot del mapa difícil. Dicho comportamiento se puede contrastar con el anexo A.

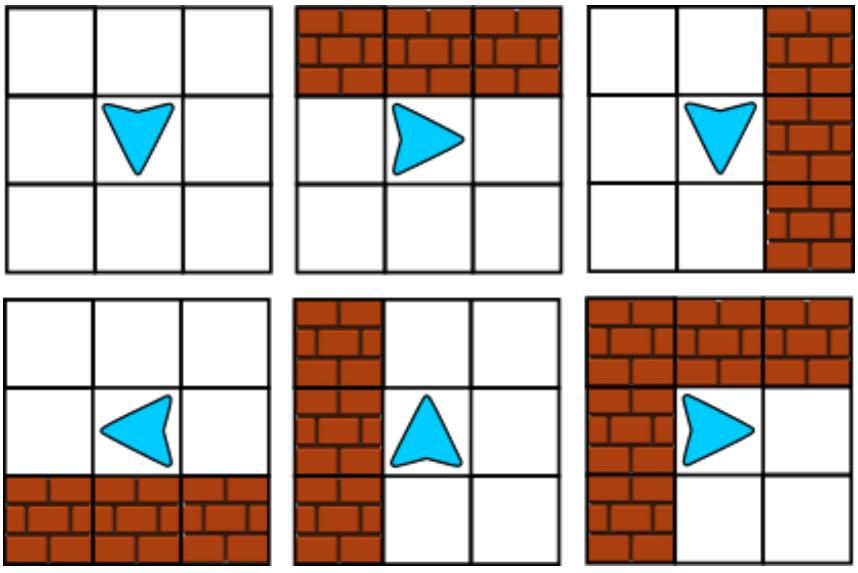


Figura 5.7: Comportamiento de algunas situaciones.

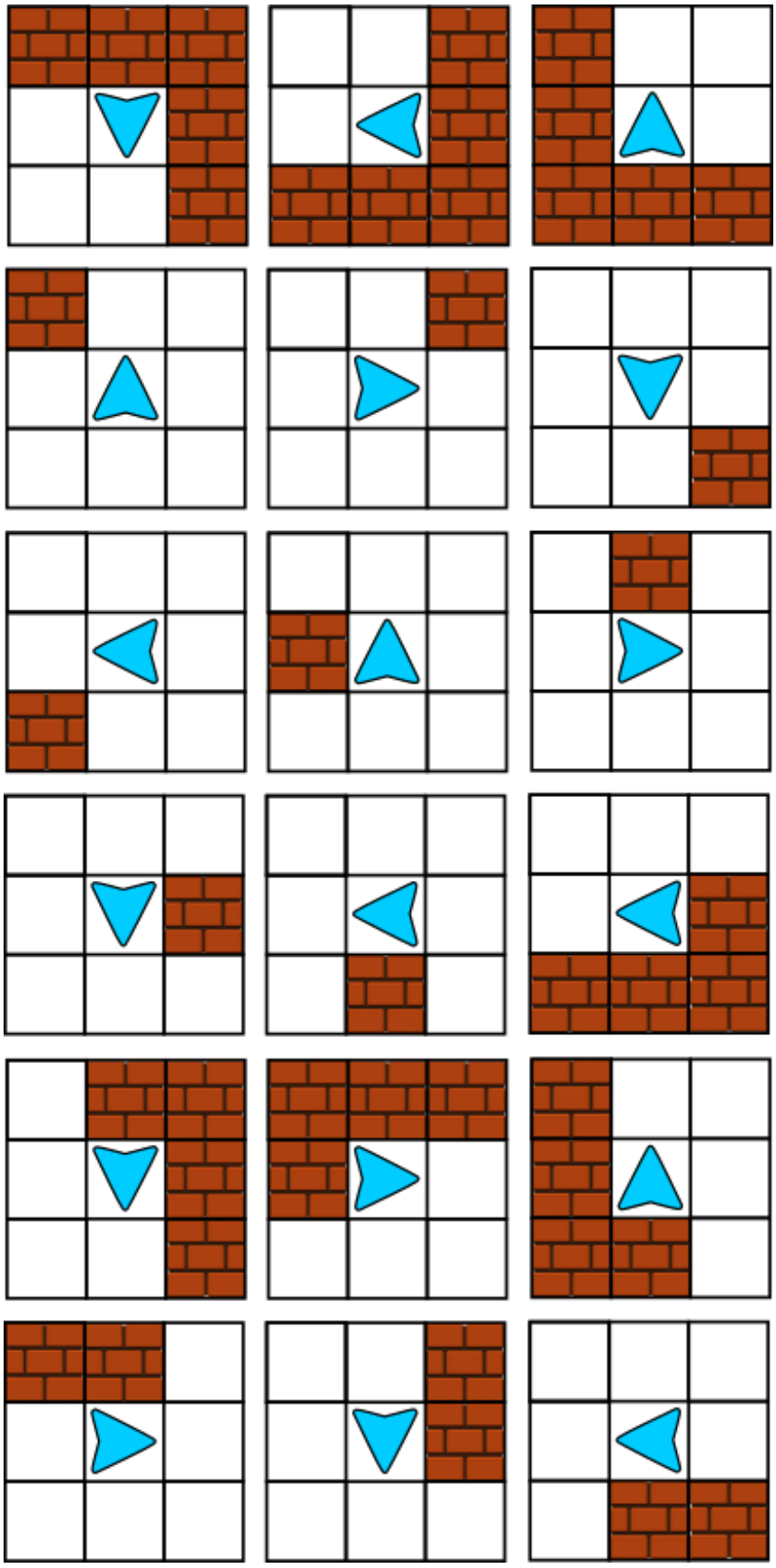


Figura 5.8: Comportamiento de algunas situaciones.

5.5 Vídeo

En la siguiente url se puede ver un vídeo de la aplicación en ejecución:

`https://youtu.be/lNvzCgomrgk`

6. Conclusión

En base a los resultados obtenidos con las pruebas realizadas, la conclusión que se obtiene es la siguiente:

Tomando como punto de referencia el mismo mapa, se pueden introducir manualmente tres parámetros de configuración: número de individuos, número de iteraciones y número de evoluciones. El número de iteraciones debe ser mayor que el resultado final, sino jamás se encontrará la solución correcta, así que este parámetro se puede considerar no variable. Por tanto, ya solo quedan el número de individuos y el número de evoluciones. Respecto a este último, a mayor número, si las especies se estancan, más posibilidades hay de encontrar la solución, pero realmente no influye en la rapidez con la que se encuentra la solución. Esto viene dado por el número de especies. A mayor número de especies, estadísticamente es más probable encontrar la solución antes ya que hay mayor diversidad. Sin embargo, tener un número considerable de especies implica mayor procesamiento y por tanto mayor tiempo dedicado por evolución. La virtud reside en considerar un número elevado de especies pero suficientemente bajo como para que no repercuta demasiado en los tiempos de ejecución.

Adicionalmente se intentó mejorar el tiempo de búsqueda pudiendo decidir si el punto de partida era aleatorio o por el contrario fijo para todas lo individuos. Con los resultados obtenidos se podría decir que a mapas más complejos, la variedad de la posición inicial es mejoría para encontrar antes la solución como sucede en el *Mapa Difícil* donde a mismo número de evoluciones, el mejor robot de una iteración recorría más perímetro. No obstante, en los otros mapas no hay tanta influencia dado su más fácil recorrido.

A nivel de resumen y como solución genérica se podría decir que a mayor dificultad del problema, generar más individuos ya que la variedad ayudará a converger antes. Y dado que no repercute en el tiempo de ejecución, es mejor inicializar los robots con posiciones aleatorias.

A. Comportamiento mejor robot

* Se muestra en dos columnas para ahorra espacio. El archivo real solo tiene una.

```
Evolucion 2292
208
20,10
[[0,0,0,0,0,0,0,0], 2]
[[0,0,0,0,0,0,0,1], 2]
[[0,0,0,0,0,0,1,0], 3]
[[0,0,0,0,0,0,1,1], 3]
[[0,0,0,0,0,1,0,0], 3]
[[0,0,0,0,0,1,0,1], 3]
[[0,0,0,0,0,1,1,0], 3]
[[0,0,0,0,0,1,1,1], 3]
[[0,0,0,0,1,0,0,0], 2]
[[0,0,0,0,1,0,0,1], 2]
[[0,0,0,0,1,0,1,0], 1]
[[0,0,0,0,1,0,1,1], 3]
[[0,0,0,0,1,1,0,0], 1]
[[0,0,0,0,1,1,0,1], 3]
[[0,0,0,0,1,1,1,0], 1]
[[0,0,0,0,1,1,1,1], 3]
[[0,0,0,1,0,0,0,0], 0]
[[0,0,0,1,0,0,0,1], 3]
[[0,0,0,1,0,0,1,0], 2]
[[0,0,0,1,0,0,1,1], 3]
[[0,0,0,1,0,1,0,0], 0]
[[0,0,0,1,0,1,0,1], 1]
[[0,0,0,1,0,1,1,0], 0]
[[0,0,0,1,0,1,1,1], 1]
[[0,0,0,1,1,0,0,0], 2]
[[0,0,0,1,1,0,0,1], 1]
[[0,0,0,1,1,0,1,0], 2]
[[0,0,0,1,1,0,1,1], 1]
[[0,0,0,1,1,1,0,0], 3]
[[0,0,0,1,1,1,0,1], 3]
[[0,0,0,1,1,1,1,0], 1]
[[0,0,0,1,1,1,1,1], 3]
[[0,0,1,0,0,0,0,0], 1]
[[0,0,1,0,0,0,0,1], 1]
[[0,0,1,0,0,0,1,0], 2]
[[0,0,1,0,0,0,1,1], 3]
[[0,0,1,0,0,1,0,0], 0]
[[0,0,1,0,0,1,0,1], 3]
[[0,0,1,0,0,1,1,0], 2]
[[0,0,1,0,0,1,1,1], 2]
[[0,0,1,0,1,0,0,0], 2]
[[0,0,1,0,1,0,0,1], 2]
[[0,0,1,0,1,0,1,0], 3]
[[0,0,1,0,1,0,1,1], 3]
[[0,0,1,0,1,1,0,0], 2]
[[0,0,1,0,1,1,0,1], 1]
[[0,0,1,0,1,1,1,0], 1]
[[0,0,1,0,1,1,1,1], 3]
[[0,0,1,1,0,0,0,0], 3]
```

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

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

```
[[1,1,1,1,0,0,0,1], 1]
[[1,1,1,1,0,0,1,0], 1]
[[1,1,1,1,0,0,1,1], 0]
[[1,1,1,1,0,1,0,0], 1]
[[1,1,1,1,0,1,0,1], 0]
[[1,1,1,1,0,1,1,0], 1]
[[1,1,1,1,0,1,1,1], 2]
[[1,1,1,1,1,0,0,0], 1]
```

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

B. Ampliación explicación código fuente

En este anexo se detalla más en profundidad los métodos mostrados en el capítulo 3. Estos métodos son: generación, evaluación, ordenación, selección, mutación, cruce y eliminación.

B.1 Fases del algoritmo

B.1.1 Generación

Este método crea la primera generación de individuos. Se crean tantos como se indique por parámetro de entrada y se almacenan en una lista. Se permite elegir si se desea que todos los individuos se generen en el mismo punto de inicio o por el contrario cada uno tenga un punto distinto. Respecto a su comportamiento, este se genera de forma aleatoria para cada uno de tal manera que para una misma entrada, el comportamiento del robot puede ser intentar moverse hacia arriba, abajo, izquierda o derecha. En caso de que la casilla a la que intenta moverse es muro se mantendrá en la misma posición. Las posibilidades de que dos individuos tenga el mismo comportamiento es 1 entre 256! ($8x10^{506}$).

B.1.2 Evaluación

Este es uno de los procesos más importantes ya que si no se evalúa o se define correctamente la función objetivo todo lo demás no sirve. Lo que se pretende maximizar es el número de celdas que recorre el robot y que pertenecen al perímetro interno de mapa. Para realizar este cálculo, en el momento de cargar en memoria el mapa, se calcula de forma dinámica el número de celdas que tiene el perímetro. Por las restricciones del problema, este proceso es sencillo ya que toda celda colindante con una celda de muro pertenece a la solución.

Por otra parte, cada robot tiene una lista con todos los puntos por donde pasa conforme se va moviendo. Antes de la evaluación, esta lista tienen tantos puntos como movimientos (iteraciones) haya hecho el robot. Por tanto, aunque solo se haya desplazado en dos casillas, si ha tenido 300 iteraciones tendrá 150 veces una casilla y 150 veces la otra.

Una vez se conocen ambas partes (el listado de puntos de cada robot y los puntos que pertenecen al perímetro, pasados por parámetro) se realiza una intersección entre los dos conjuntos de tal forma que en el listado de cada individuo solo quedan las celdas que cumplen las dos condiciones: que pertenecen a la solución y que haya pasado el robot. De esta forma, el proceso de evaluación se reduce a contar la longitud del listado *PathTravelled* de cada individuo.

```
public static List<ETTank> evaluate(List<ETTank> creatures,
   List<Point> mapSolution)
{
List<ETTank> creaturesTemp = creatures;
List<Point> pathTravelled = new List<Point>();
for (int i = 0; i < creaturesTemp.Count; i++)
{
pathTravelled = creaturesTemp.ElementAt(i).PathTravelled
.Intersect(mapSolution).Distinct<Point>().ToList<Point>();
creaturesTemp.ElementAt(i).PathTravelled = pathTravelled;
}
return creaturesTemp;
}
```

Listado B.1: Método de evaluación.

B.1.3 Ordenación

Una vez se ha filtrado las celdas por las que ha pasado cada criatura, en este método se ordena el conjunto de individuos dejando en la posición 0 el que más celdas haya recorrido. Esta ordenación se realiza utilizando LinQ.

```
public static List<ETTank> order(List<ETTank> creatures)
{
List<ETTank> creaturesTemp = creatures;
creaturesTemp = creaturesTemp.OrderByDescending(o =>
   o.PathTravelled.Count).ToList();
return creaturesTemp;
}
```

Listado B.2: Método de ordenación.

B.1.4 Selección

El hecho de ordenar los individuos tiene su importancia en este método de selección y es que se coge la primera mitad del listado ordenado, es decir, desde el mejor hasta la mitad. De esta manera se elimina la posibilidad de reproducción de los peores individuos.

```
    public static List<ETTank> select(List<ETTank>
   creatures)
    {
    List<ETTank> sublist = creatures.GetRange(0,
    creatures.Count/2);
    return sublist;
    }
```

Listado B.3: Método de selección.

B.1.5 Mutación

Como toda especie en la naturaleza, en ocasiones puede que mute un gen de un individuo. Se ha permitido que todo individuo tenga un 5 % de probabilidad de cambiar uno de sus genes al azar. Esto supondría que dada una entrada cualquier (0,1,1,0,1,1,1,1) cuyo comportamiento era intentar desplazarse hacia arriba cambie a intentar desplazarse hacia cualquiera de las cuatro posibles movimientos pudiéndose quedar con el mismo comportamiento. Esta nueva criatura sustituye a la original.

```
public static List<ETTank> mutation(List<ETTank> creatures)
{
List<ETTank> creaturesTemp = creatures;
Random rnd = new Random(DateTime.Now.Millisecond);
int n = 0;
string key;
for (int i = 0; i < creaturesTemp.Count; i++)
{
n = rnd.Next(0, 100);
if (n < 5)
{
n = rnd.Next(0, 256);
key = creaturesTemp.ElementAt(i).Brain.Situations.
ElementAt(n).Key;
creaturesTemp.ElementAt(i).Brain.Situations[key] =
rnd.Next(0, 4);
}
}
return creaturesTemp;
}
```

Listado B.4: Método de mutación.

B.1.6 Cruce

Este proceso es muy importante ya que es donde se produce el paso de genes de una generación a otra. Dados dos padres seleccionados de la mitad de los mejores del conjunto (estos padres no pueden volver a traspasar sus genes hasta la próxima evolución), se elige un punto

de división de forma aleatorio (entre 0 y 255) y se mezclan por ese punto los genes de ambos padres generando dos nuevos individuos con genes de ambos. En el código se puede ver como se almacena la primera y segunda mitad de cada padre y después se combina la primera mitad de uno con la segunda mitad del otro y se asigna a un hijo y viceversa. Estos hijos se añaden al listado de individuos existente. Esto quiere decir que si la población existente es 1000 tras el cruce será de 1500 ya que a partir de dos padres se generan dos hijos.

```
    public static List<ETTank> crossing(List<ETTank> creatures,
Matrix map)
    {
    List<ETTank> newGeneration = new List<ETTank>();
    ETTank e1 = null, e2 = null, off1 = null, off2 = null;
    Random rnd = new Random(DateTime.Now.Millisecond);
    IDictionary<string, int> firstHalf, secondHalf, firstHalf2,
secondHalf2;
    int n = 0;

    while (creatures.Count>0)
    {
    n = rnd.Next(0, creatures.Count);
    e1 = creatures.ElementAt(n);
    creatures.RemoveAt(n);
    n = rnd.Next(0, creatures.Count);
    e2 = creatures.ElementAt(n);
    creatures.RemoveAt(n);
    n = rnd.Next(0, 256);

    off1 = new ETTank(ETTank.InitialPositionMap);
    off2 = new ETTank(ETTank.InitialPositionMap);


    firstHalf = e1.Brain.Situations.Take(n).ToDictionary(kv =>
kv.Key, kv => kv.Value);
    secondHalf = e1.Brain.Situations.Skip(n).ToDictionary(kv =>
kv.Key, kv => kv.Value);

    firstHalf2 = e2.Brain.Situations.Take(n).ToDictionary(kv =>
kv.Key, kv => kv.Value);
    secondHalf2 = e2.Brain.Situations.Skip(n).ToDictionary(kv =>
kv.Key, kv => kv.Value);

    secondHalf2.ToList().ForEach(x => firstHalf.Add(x.Key,
x.Value));

    secondHalf.ToList().ForEach(x => firstHalf2.Add(x.Key,
x.Value));

    off1.Brain.Situations = firstHalf;
    off2.Brain.Situations = firstHalf2;

    newGeneration.Add(off1);
    newGeneration.Add(off2);
```

```
38      }
40
        return newGeneration;
42      }
```

Listado B.5: Método de cruce.

B.1.7 Eliminación

Por último, para evitar el crecimiento descontrolado de la especie, se eliminan los peores individuos. Cuando se ejecuta este método, se han cruzado los padres por lo que si se continua con el ejemplo actualmente hay 1500 individuos. Por tanto la eliminación consiste en matar los 500 (1/3 partes) peores individuos. Se puede utilizar el método *GetRange* de *C#* porque están ordenados los 1500 individuos.

```
public static List<ETTank> killWorst(List<ETTank> creatures, int
    number)
2 {
List<ETTank> sublist = creatures.GetRange(0, number);
4 return sublist;
}
```

Listado B.6: Método de eliminación.