AGRUPAMENTO DE ESCOLAS DE
CISTER - ALCOBAÇA

Ano letivo
2016/2017

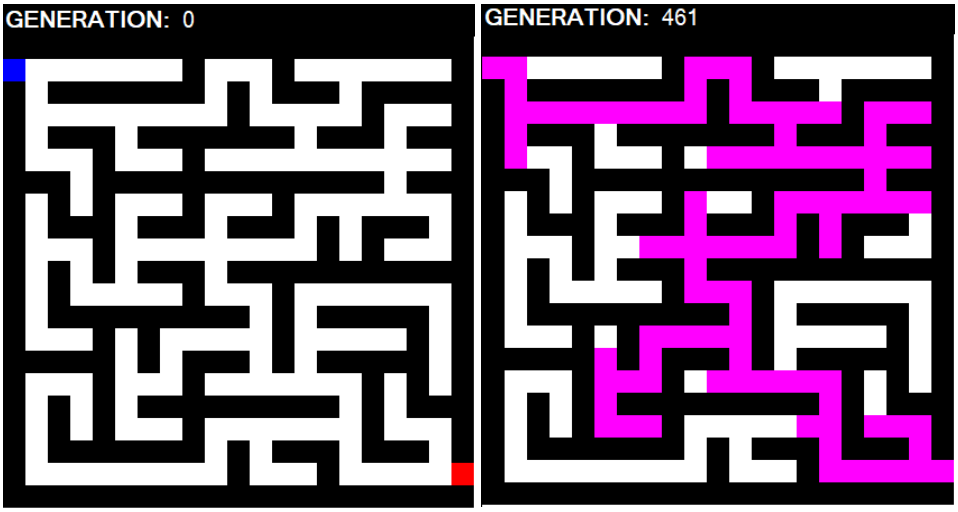
ESCOLA SECUNDÁRIA D. INÊS DE CASTRO

CURSO CIENTÍFICO-HUMANÍSTICO DE

CIÊNCIAS E TECNOLOGIAS

PROJETO DE APLICAÇÕES INFORMÁTICAS B

Algoritmos Genéticos



NOME DO ALUNO: **FRANCISCO MIGUEL LOIRO SERRALHEIRO** **Nº:14**

ANO: 12º **TURMA:** 12ºCT-A

PROFESSOR: Nuno José da Silva Trindade Duarte

DATA: 12-12-2016

Índice

1. Resumo

Este relatório descreve o processo e o trabalho que esteve subjacente à conceção e programação de um algoritmo genético. O algoritmo genético irá ter como principal função a de resolver qualquer labirinto, independentemente do grau de complexidade do mesmo.

Primeiramente, são expostos os problemas de programação, diretamente relacionados com conceitos de algoritmos genéticos, como por exemplo, o problema do *fitness* e o problema do *crossover*.

Posteriormente, os problemas gráficos são resolvidos, para irem de encontro àquilo que é o principal objetivo – apresentar ao utilizador a solução do labirinto.

2. Introdução

O presente relatório é elaborado no âmbito da disciplina de Aplicações Informáticas B, com vista à conclusão de um novo projeto, neste caso, continuando a aprender novas áreas da Programação.
O projeto desenvolveu-se na Escola Secundária D. Inês de Castro, nas aulas da disciplina, em computadores escolares e no meu computador pessoal, durante o 3º Período letivo. O programa utilizado foi apenas o Visual Studio 2015.

O tema abordado foi algoritmos genéticos. *Genetic algorithms* são uma área da inteligência artificial, e o seu principal objetivo é evoluir, através de processos evolutivos baseados na teoria da evolução biológica, uma população de indivíduos abstratos, até atingirem um determinado objetivo.

Assim, com o principal objetivo de aprender o máximo sobre esta *branch* da inteligência artificial, desenvolvi um algoritmo genético que evolui um número de indivíduos de uma mesma população a resolverem um labirinto. Como linguagem de programação escolhi o C#, visto que é a que é a utilizada no ambiente de programação que estou mais familiarizado, o Visual Studio.

3. Descrição do problema

O problema que se pretende resolver é o de programar um algoritmo genético que resolva qualquer labirinto. Isto é, terá de ser programado um algoritmo que traduza o método genético de evoluir uma população, com o objetivo de esta convergir para uma solução do labirinto.

Posto isto, o grande objetivo a que se pretende chegar é um algoritmo que consiga resolver qualquer labirinto, independentemente da complexidade.

4. Recursos utilizados

Neste projeto utilizei variados recursos para o suporte e o desenvolvimento do programa. No que toca ao ambiente de programação utilizei o Visual Studio Community 2015, na versão 14.0.

Para manter o trabalho atualizado entre os diversos aparelhos usei uma pen USB e o Google Drive.

Foram-me disponibilizados equipamentos e ajuda informática pelo professor orientador e pela escola.

5. Desenvolvimento do projeto

Antes de começar qualquer programação, é necessário dividir o problema em etapas, neste caso, as etapas da evolução genética.

Primeiramente, o indivíduo a ser criado em qualquer algoritmo genético é dotado de material genético (DNA) e o *fitness*. O material genético é aquilo que vai sofrer, diretamente, evolução, e é o que vai ser responsável de expôr essa evolução, tal como um ser humano e o seu material genético. O *fitness* é o grau de aptidão/prestação do indivíduo no ambiente em que evolui. É uma "pontuação" que vai aumentando consoante a prestação do indivíduo.

Assim, transcrevendo para o problema do labirinto, o material genético dos invíduos vai ser o movimento deste, isto é, todos os indivíduos vão ter 5 genes diferentes: o que é responsável por mover o indíviduo para a esquerda, para a direita, para cima, para baixo e um gene que é responsável por não mover o indivíduo. No caso do *fitness*, quanto maior for a proximidade do indivíduo ao final, maior o grau de *fitness*.

O próximo passo para programar um algoritmo genético é compreender os mecanismos de evolução: geração de uma população, cálculo de *fitness* dos indivíduos, ordenação decrescente da população por fitness, seleção dos indivíduos para a próxima população, os indivíduos selecionados vão sofrer fenómenos de *crossover* entre o seu material genético e, finalmento, a mutação (ou não) dos indivíduos da população. Este método é iterado por cada geração de indivíduos.

5.1. Problemas de programação

Problema da primeira população

Antes do algoritmo genético evoluir uma população, é necessário haver uma primeira população para começar a evoluçaõ. Por isso, uma primeira população de indivíduos, cujo número pode ser escolhido pelo utilizador no painel de controlo da aplicação, é gerada aleatoriamente, com o número de genes de cada indivíduo a ser também decidido aleatoriamente, entre um mínimo e máximo, escolhidos, também, pelo utilizador.Onde o método sublinhado é um construtor de objetos da classe "Individual", que controi indivíduos com o número de genes compreendido entre "MinGenomeSize" e "MaxGenomeSize".

```
Individual[] InitPopulation()
{
    int MinGenomeSize = Convert.ToInt16(textBoxMinGenome.Text);
    int MaxGenomeSize = Convert.ToInt16(textBoxMaxGenome.Text);
    //reset population
    population = new Individual[PopulationSize];
    for (int i = 0; i < population.Length; i++)
    {
        Individual p = new Individual(MinGenomeSize, MaxGenomeSize);
        population[i] = p;
    }
    return population;
}
```

Problema o cálculo de fitness

Depois, os indivíduos dessa população têm de ter um *fitness* que é calculado:

```
void FitnessEvalAndRanking(Individual[] pop)
{
    foreach (Individual ind in pop)
        ind.evaluate(labirinto);

    Array.Sort(pop);
}
```

O método "FitnessEvalAndRanking" começa um processo iterativo por cada indivíduo, utilizando o método "evaluate" que está à direita. Neste momento, o genoma dos indivíduos vai ser lido, gerando uma posição no labirinto. Depois, o inverso da distância entre essa posição gerada pelo movimento do indivíduo (pelo genoma) e o ponto da saída resulta no *fitness* indivíduo.

```
Point position = labirinto.Entry;

//get DNA
for (int i = 0; i < DNA.Length; i++)
{
    switch (DNA[i])
    {
        case Gene.Left:
            if (position.Y == 0) break;
            if (labirinto.grid[position.X, position.Y - 1] == 0) break;
            position.Y--; break;

        case Gene.Right:
            if (position.Y == labirinto.Width - 1) break;
            if (labirinto.grid[position.X, position.Y + 1] == 0) break;
            position.Y++; break;

        case Gene.Up:
            if (position.X == 0) break;
            if (labirinto.grid[position.X - 1, position.Y] == 0) break;
            position.X--; break;

        case Gene.Down:
            if (position.X == labirinto.Height - 1) break;
            if (labirinto.grid[position.X + 1, position.Y] == 0) break;
            position.X++; break;

    }
}
double distance = Math.Sqrt(Math.Pow(labirinto.Exit.X - position.X, 2)
    + (Math.Pow(labirinto.Exit.Y - position.Y, 2)));
fitness = 1 / distance;
```

Problema da seleção

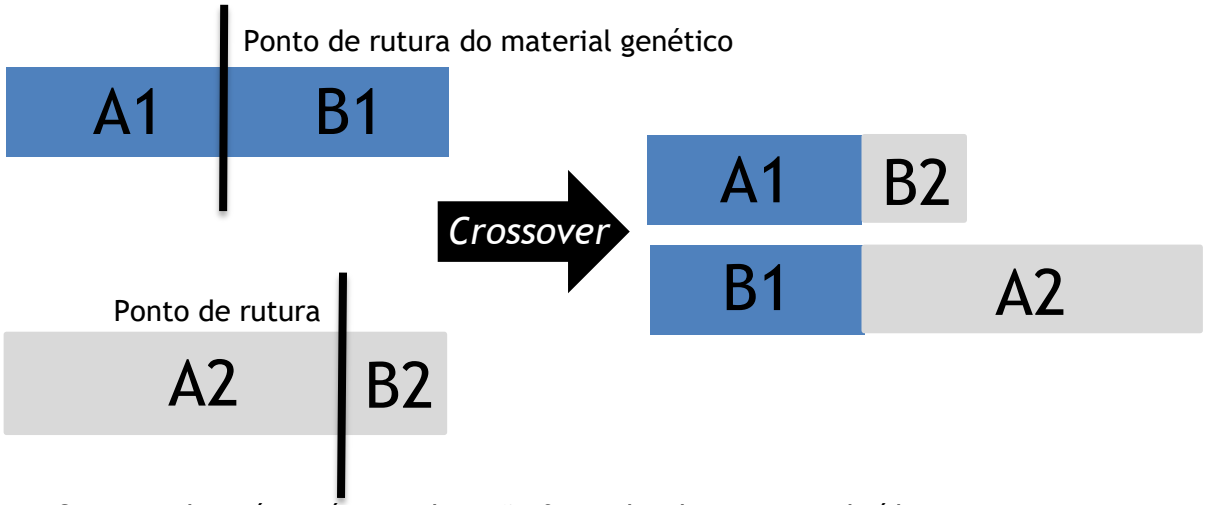
Posteriormente, os melhores indivíduos desta população, pois estes foram organizados anteriormente, através de uma "Icomparable" e do método "Array.Sort", irão ser selecionados para sofrer *crossover* e *mutation*, que são os fenómenos que irão fazer converger a população em indivíduos aptos para solucionar o problema. O utilizador poderá escolher uma probabilidade, a probabilidade de seleção, e a cada indivíduo vai ser testada esta

```
Individual[] Selection(Individual[] population)
{
    Individual[] popselected = new Individual[PopulationSize];

    //ciclo que vai selecionar a próxima pop baseada no ranking
    for (int i = 0; i < popselected.Length; i++)
    {
        for (int j = 0; j < population.Length; j++)
        {
            float prob = rnd.Next(0, 101);
            if (prob < PSelec)
            {
                popselected[i] = population[j];
                break;
            }
        }
        if (popselected[i] == null) popselected[i] = population[population.Length - 1];

    }
    return popselected;
}
```

probabilidade. Se o indíviduo passar, passa diretamente para a próxima étapa e o ciclo é recomeçado, de maneira a que os melhores indivíduos tenham maior probabilidade de ser selecionados. Desta maneira a próxima população irá ter melhores indivíduos (maior *fitness*). Irá também ser garantida a passagem do último indivíduo se todos os indivíduos não conseguirem ser selecionados.

Problema do *crossover*

Quando os indivíduos chegam a esta fase do algoritmo, vão sofrer o *crossover* do materia genético. Este fenómeno é caracterizado por dar variabilidade genética à população. O *crossover* é feito de maneira que o material genético é partido em dois pontos gerados aleatoriamente, depois esses conjuntos de genes são cruzados, de maneira que o esquema expõe:



O material genético é cruzado e são formados dois novos indivíduos

```
Individual[] Crossover(Individual[] matingPool)
{
    List<Individual> mutationpool = new List<Individual>();
    Individual firstcross = null;
    foreach (Individual ind in matingPool)
    {
        //checkforcross
        float prob = rnd.Next(0, 101);

        if (prob < PCross)
        {
            //check if some individual is in standby
            if (firstcross == null)
                firstcross = ind;

            //cross
            else
            {
                Individual[] results = firstcross.cross(ind);

                //add results to pool
                foreach (Individual i in results)
                {
                    mutationpool.Add(i);
                }
                //reset firstcross
                firstcross = null;
            }
        }
        else
            mutationpool.Add(ind);
    }
```

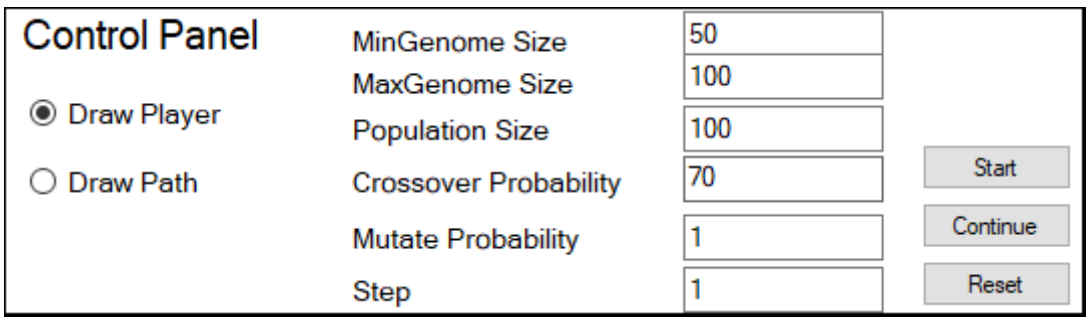
A cada indivíduo vai ser testado esta probabilidade, se passar, o indivíduo irá sofrer *crossover* com outro indivíduo. Senão, o indivíduo irá passar diretamente para a próxima *pool*, que é a *pool* de mutação. Desta maneira, a próxima população irá ter maior variabilidade genética e indivíduos mais aptos.

Problema da mutação

A quarta e última alteração que o genoma dos indivíduos pode sofrer é a mutação. A mutação é mais uma garantia de introduzir variabilidade na população, através da alteração de um gene, numa posição aleatória no genoma, para outro gene aleatório. Desta maneira, se o indivíduo ficar preso num máximo local, é através de mecanismos como a mutação que será possível a escapatória desses máximos. Assim, o utilizador poderá escolher, no painel de controlo, uma probabilidade de mutação. Cada indivíduo será submetido a essa probabilidade. Se passar o indivíduo sofrará mutação.

```
void Mutation(Individual[] pop)
{
    foreach (Individual ind in pop)
    {
        float prob = rnd.Next(0, 101);
        if (prob <= PMutate) ind.mutate();
    }
}
```

```
public void mutate()
{
    if (DNA.Length == 0) return;

    int geneposition = rnd.Next(0, DNA.Length);
    int anothergene = rnd.Next(0, 5);

    switch (anothergene)
    {
        case 0:
            this.DNA[geneposition] = Gene.Left;
            break;
        case 1:
            this.DNA[geneposition] = Gene.Right;
            break;
        case 2:
            this.DNA[geneposition] = Gene.Up;
            break;
        case 3:
            this.DNA[geneposition] = Gene.Down;
            break;
        case 4:
            this.DNA[geneposition] = Gene.Stop;
            break;
    }
}
```



Painel de controlo

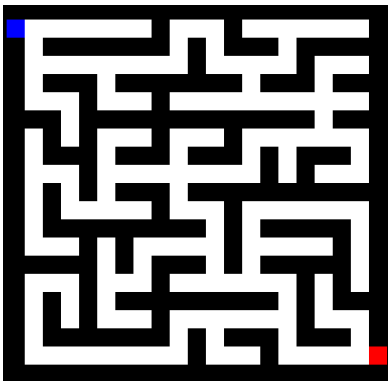
5.2. Problemas Gráficos

Uma grande componente deste projeto é, além do algoritmo genético, a representação gráfica. Não só para fins de *debug*, a representação da evolução de uma população é sempre muito mais interessante do que a análise de variáveis em menus de *debug*. Por isso, era obrigatório este projeto ter a representação gráfica do problema. Primeiro foi preciso resolver o problema da representação do labirinto cujo os indivíduos vão tentar solucionar. Depois é muito apelativa a representação, nesse labirinto, do melhor indivíduo da geração em questão.

Problema do labirinto

Para ser mais leve para a memória do computador, foi decidido que o labirinto seria guardado como um vetor de números inteiros, zeros, uns, dois e três. Para a representação gráfica, esse vetor é lido e por cada 0 é criada uma "PictureBox" preta, que representa uma parede do labirinto, por cada 1 cria uma "PictureBox" branca, que representa o espaço onde o indivíduo pode andar.

```
int[,] l = {
   //----------------------------------------------->y
  {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,},
  {2,1,1,1,1,1,1,1,0,1,1,1,0,1,1,1,1,1,1,1,0,},
  {0,1,0,0,0,0,0,0,0,1,0,1,0,0,0,1,0,0,0,0,0,},
  {0,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,0,1,1,0,},
  {0,1,0,0,0,1,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,},
  {0,1,1,1,0,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,0,},
  {0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,},
  {0,1,0,1,0,1,1,1,0,1,1,1,0,1,1,1,1,1,1,1,0,},
  {0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,1,0,0,0,1,0,},
  {0,1,1,1,0,1,1,1,1,1,1,1,1,0,1,0,1,1,1,0,},
  {0,1,0,1,0,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,},
  {0,1,0,1,1,1,1,1,0,1,1,1,0,1,1,1,1,1,1,0,},
  {0,1,0,0,0,0,0,0,0,0,0,1,0,1,0,0,0,0,1,0,},
  {0,1,1,1,0,1,0,1,1,1,1,1,0,1,1,1,1,0,1,0,},
  {0,0,0,0,0,1,0,1,0,0,0,1,0,1,0,0,0,0,1,0,},
  {0,1,1,1,0,1,1,1,0,1,1,1,1,1,1,1,0,1,0,1,0,},
  {0,1,0,1,0,1,0,0,0,0,0,0,0,0,1,0,1,0,0,0,},
  {0,1,0,1,0,1,1,1,0,1,1,1,1,1,1,1,0,1,1,1,0,},
  {0,1,0,0,0,0,0,0,0,1,0,1,0,0,0,1,0,0,0,1,0,},
  {0,1,1,1,1,1,1,1,1,1,0,1,1,1,0,1,1,1,1,1,3,},
  {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,},
```



```
public void draw(Panel panel, PictureBox[,]pictures) {
    //DRAW LABIRINTO
    for (int i = 0; i < this.Height; i++)
    {
        for (int j = 0; j < this.Width; j++)
        {
            PictureBox pb = new PictureBox();
            pb.Size = new Size(20, 20);
            pb.Location = new Point(j * 20, i * 20);


            switch (this.grid[i, j])
            {
                case 0: pb.BackColor = Color.Black; break;
                case 1: pb.BackColor = Color.White; break;
                case 2: pb.BackColor = Color.Blue; break;
                case 3: pb.BackColor = Color.Red; break;
            }

            pictures[i, j] = pb;
            panel.Controls.Add(pb);

        }
    }
}
```

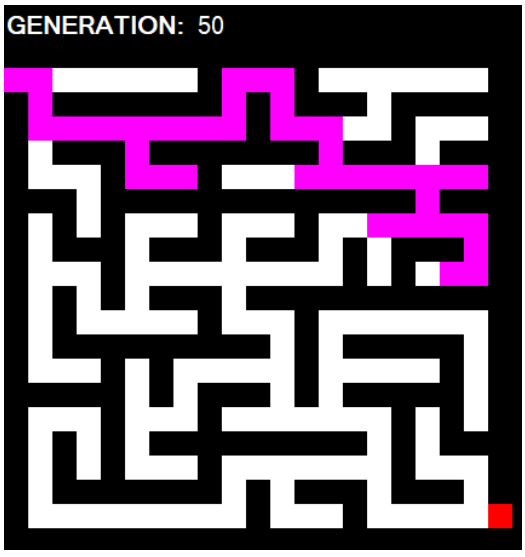
Só há um dois e um três, que representam a entrada e a saída, respetivamente, e têm como cor de fundo o azul e o vermelho, respetivamente. Há também uma classe "Labirinto", onde é guardada o objeto labirinto, os seus pontos de entrada e saída, a sua largura e altura. Além disso, juntamente com o desenho do labirinto, é criado um vetor de "PictureBox" para armazenar as "PictureBox" que pertencem ao labirinto, auxiliando com o desenho do melhor indivíduo.

Problema da representação do melhor indivíduo

Finalmente, para percecionar a evolução de uma maneira mais visual, seria necessário a representação gráfica do melhor indivíduo de uma população que, como é organizado decrescentemente por *fitness*, será o primeiro indivíduo. Para desenhar o caminho que este leva, o material genético é lido, mais uma vez, e a cada posição resultante de cada gene é "pintada" a "PictureBox" respetiva no vetor das imagens, que foi criado no método que desenho o labirinto.

```
//get DNA
for (int i = 0; i < pop[0].DNA.Length; i++)
{
    switch (pop[0].DNA[i])
    {
        case Gene.Left:
            if (position.Y == 0) break;
            if (labirinto.grid[position.X, position.Y - 1] == 0) break;
            position.Y--; break;

        case Gene.Right:
            if (position.Y == labirinto.Width - 1) break;
            if (labirinto.grid[position.X, position.Y + 1] == 0) break;
            position.Y++; break;

        case Gene.Up:
            if (position.X == 0) break;
            if (labirinto.grid[position.X - 1, position.Y] == 0) break;
            position.X--; break;

        case Gene.Down:
            if (position.X == labirinto.Height - 1) break;
            if (labirinto.grid[position.X + 1, position.Y] == 0) break;
            position.X++; break;

    }

    pictures[position.X, position.Y].BackColor = Color.Magenta;

}
```

Representação do melhor indivíduo na geração 50:



6. Conclusões

Concluindo, este projeto cumpriu, minimamente, os objetivos projetados. Para o tempo em que foi desenvolvido o algoritmo resultante conseguiu resolver, em poucas gerações, um labirinto de grau de complexidade média. Mesmo assim, ainda gostaria de testar com muito mais detalhe e pormenor o algoritmo, porque nunca deixou de gerar resultados estranhos, assim como melhorar a interface do utilizador.

7. Reflexão Final

A grande lição que consegui retirar deste projeto foi a de manter a calma e partir um problema muito maior em problemas mais pequeninos. E apesar de eu já o saber, no que envolveu parte gráfica e algoritmo não me consegui concentrar apenas num, tentei resolver todos de uma só vez, o que culminou num programa cheio de bugs e erros e "NullPointerException".

Por isso, sempre que me deparar com um problema que envolva reprentação gráfica, elaboração de um algoritmo ou ambos é sempre importante me relembrar que os problemas devem ser repartidos, resolvidos e testados individualmente. Até que, num todo, seja interligado e testado conseguindo, desta maneira, um projeto seguro e conciso.

8. Referências

1. http://stackoverflow.com/ - Consultado ao longo do projeto para esclarecer as mais variadas dúvidas sobre programação.
2. https://www.youtube.com/watch?v=kHyNqSnzP8Y&t=1636s – Consultado, no inicio do projeto, para estudar a teoria dos algoritmos genéticos.

9. Anexos

Código do Form:

```
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;
using System.IO;

namespace WindowsFormsApplication23
{

    public partial class FormGA : System.Windows.Forms.Form
    {
        int generationnumb = 0;
        int PopulationSize = 10;
        int genomesize = 0;
        bool stop = false;
        int step = 0;
        int PSelec = 70;
        int PCross = 30;
        int PMutate = 1;

        Individual[] population;
        PictureBox[,] pictures;
        Labirinto labirinto;

        private static Random rnd;

        static FormGA()
        {
            rnd = new Random();
        }

        public FormGA()
        {
            InitializeComponent();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            genomesize = Convert.ToInt16(textBoxPopulation.Text);
            PopulationSize = Convert.ToInt16(textBoxPopulation.Text);
            labelGeneration.Text = Convert.ToString(generationnumb);
            PSelec = Convert.ToInt16(textBoxCrossover.Text);
            PMutate = Convert.ToInt16(textBoxMutation.Text);
            step = Convert.ToInt16(textBoxStep.Text);


            //int[,] l = {
            //    //-------------------------------------------------->y
            //    {0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1},//|
            //    {0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},//|
            //    {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1},//|
            //    {0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},//|
            //    {0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},//|
```

```
//    {0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},//|
//    {0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},//|
//    {0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},//|
//    {0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},//|
//    {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0},//V
//    {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,1,1,1,1,3},//x

//};
//int[,] l = {
//    //------------------------------------------------->y
//    {0,0,0,2,0,0,0,0,0,},
//    {0,1,1,1,0,0,0,0,0,},
//    {0,1,0,1,0,0,0,0,0,},
//    {0,1,0,1,1,1,1,1,0,},
//    {0,1,0,0,0,0,1,0,0,},
//    {0,1,0,1,1,1,1,1,0,},
//    {0,0,0,0,1,0,0,0,0,},
//    {0,1,1,1,1,0,0,0,0,},
//    {0,0,0,0,3,0,0,0,0,},
//};
/*int[,] l = {
        {0,0,0,2,0},
        {0,1,1,1,0},
        {0,1,0,0,0},
        {0,1,1,1,0},
        {0,0,0,3,0}


    };*/
int[,] l = {
    //------------------------------------------------->y
  {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,},
  {2,1,1,1,1,1,1,1,0,1,1,1,0,1,1,1,1,1,1,1,0,},
  {0,1,0,0,0,0,0,0,0,1,0,1,0,0,0,1,0,0,0,0,0,},
  {0,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,0,1,1,1,0,},
  {0,1,0,0,0,1,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,},
  {0,1,1,1,0,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,0,},
  {0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,},
  {0,1,0,1,0,1,1,1,0,1,1,1,0,1,1,1,1,1,1,1,0,},
  {0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,1,0,0,0,1,0,},
  {0,1,1,1,0,1,1,1,1,1,1,1,1,1,0,1,0,1,1,1,0,},
  {0,1,0,1,0,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,},
  {0,1,0,1,1,1,1,1,0,1,1,1,0,1,1,1,1,1,1,1,0,},
  {0,1,0,0,0,0,0,0,0,0,1,0,1,0,0,0,0,0,1,0,0,},
  {0,1,1,1,0,1,0,1,1,1,1,1,0,1,1,1,1,1,0,1,0,},
  {0,0,0,0,0,1,0,1,0,0,0,1,0,1,0,0,0,0,0,1,0,},
  {0,1,1,1,0,1,1,1,0,1,1,1,1,1,1,1,0,1,0,1,0,},
  {0,1,0,1,0,1,0,0,0,0,0,0,0,0,0,1,0,1,0,0,0,},
  {0,1,0,1,0,1,1,1,0,1,1,1,1,1,1,1,0,1,1,1,0,},
  {0,1,0,0,0,0,0,0,0,1,0,1,0,0,0,1,0,0,0,1,0,},
  {0,1,1,1,1,1,1,1,1,1,1,0,1,1,1,0,1,1,1,1,1,3,},
  {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,},

};
//labirinto = new Labirinto(l, new Point(0,3), new Point(4,3));
//labirinto = new Labirinto(l, new Point(0, 4), new Point(4, 8));
labirinto = new Labirinto(l, new Point(1, 0), new Point(20, 21));

pictures = new PictureBox[labirinto.Height, labirinto.Width];
population = new Individual[PopulationSize];

panel1.Width = labirinto.Width*20;
panel1.Height= labirinto.Height*20;

labirinto.draw(panel1, pictures);
```

```
}
private void evolute()
{
    //checkstep
    int rest = 0;
    rest = generationnumb % step;
    if (rest == 0)//Stop
        stop = true;
    //dar continuidade à geração
    if (generationnumb == 0)
    {
        population = InitPopulation();
        generationnumb++;
        labelGeneration.Text = Convert.ToString(generationnumb);
    }
    else
    {
        generationnumb++;
        labelGeneration.Text = Convert.ToString(generationnumb);
    }

    FitnessEvalAndRanking(population);

    //representação gráfica
    DrawBestPlayer(population, pictures);

    //evolution
    population = Selection(population);
    population = Crossover(population);
    Mutation(population);
}
private void DrawBestPlayer(Individual[] pop, PictureBox[,] pictures)
{
    labirinto.redraw(panel1, pictures);
    //First pos
    Point position = labirinto.Entry;

    pictures[position.X, position.Y].BackColor = Color.Magenta;

    SuspendLayout();

    //get DNA
    for (int i = 0; i < pop[0].DNA.Length; i++)
    {
        switch (pop[0].DNA[i])
        {
            case Gene.Left:
                if (position.Y == 0) break;
                if (labirinto.grid[position.X, position.Y - 1] == 0) break;
                position.Y--; break;

            case Gene.Right:
                if (position.Y == labirinto.Width - 1) break;
                if (labirinto.grid[position.X, position.Y + 1] == 0) break;
                position.Y++; break;

            case Gene.Up:
                if (position.X == 0) break;
                if (labirinto.grid[position.X - 1, position.Y] == 0) break;
                position.X--; break;

            case Gene.Down:
                if (position.X == labirinto.Height - 1) break;
                if (labirinto.grid[position.X + 1, position.Y] == 0) break;
```

```
                        position.X++; break;

                }

                pictures[position.X, position.Y].BackColor = Color.Magenta;

            }

            ResumeLayout();
        }
private void radioButtonElement_CheckedChanged(object sender, EventArgs e)
        {

        }

        private void radioButtonPath_CheckedChanged(object sender, EventArgs e)
        {

        }
        private void reset()
        {
            generationnumb = 0;
            population = new Individual[PopulationSize];
            evolute();
        }
        private void buttonReset_Click(object sender, EventArgs e)
        {
            reset();
        }

        private void buttonStart_Click(object sender, EventArgs e)
        {
            evolute();
        }

        private void buttonContinue_Click(object sender, EventArgs e)
        {
            stop = false;
            if (buttonContinue.Text.Equals("Stop")) stop = true;
            if (!stop)
            {
                buttonContinue.Text = "Stop";
                while (!stop)
                {
                    evolute();
                    //checkforstop
                    if (stop) buttonContinue.Text = "Continue";
                }
            }
            else buttonContinue.Text = "Continue";


        }

        Individual[] InitPopulation()
        {
            int MinGenomeSize = Convert.ToInt16(textBoxMinGenome.Text);
            int MaxGenomeSize = Convert.ToInt16(textBoxMaxGenome.Text);
            //reset population
            population = new Individual[PopulationSize];
            for (int i = 0; i < population.Length; i++)
            {
                Individual p = new Individual(MinGenomeSize, MaxGenomeSize);
                population[i] = p;
            }
```

```
            return population;
        }

        void FitnessEvalAndRanking(Individual[] pop)
        {
            foreach (Individual ind in pop)
                ind.evaluate(labirinto);

            Array.Sort(pop);
        }

        Individual[] Selection(Individual[] population)
        {
            Individual[] popselected = new Individual[PopulationSize];

            //ciclo que vai selecionar a próxima pop baseada no ranking
            for (int i = 0; i < popselected.Length; i++)
            {
                for (int j = 0; j < population.Length; j++)
                {
                    float prob = rnd.Next(0, 101);
                    if (prob < PSelec)
                    {
                        popselected[i] = population[j];
                        break;
                    }
                }
                if (popselected[i] == null) popselected[i] = population[population.Length -
1];


            }
            return popselected;
        }

        Individual[] Crossover(Individual[] matingPool)
        {
            List<Individual> mutationpool = new List<Individual>();
            Individual firstcross = null;
            foreach (Individual ind in matingPool)
            {
                //checkforcross
                float prob = rnd.Next(0, 101);

                if (prob < PCross)
                {
                    //check if some individual is in standby
                    if (firstcross == null)
                        firstcross = ind;

                    //cross
                    else
                    {
                        Individual[] results = firstcross.cross(ind);

                        //add results to pool
                        foreach (Individual i in results)
                        {
                            mutationpool.Add(i);
                        }
                        //reset firstcross
                        firstcross = null;
                    }
                }
                else
```

```
                mutationpool.Add(ind);
            }
            return mutationpool.ToArray();
        }

        void Mutation(Individual[] pop)
        {
            foreach (Individual ind in pop)
            {
                float prob = rnd.Next(0, 101);
                if (prob <= PMutate) ind.mutate();
            }
        }

        private void textBoxStep_TextChanged(object sender, EventArgs e)
        {
            if (textBoxStep.Text == "") return;
            step = Convert.ToInt16(textBoxStep.Text);
            evolute();

        }

        private void textBoxPopulation_TextChanged(object sender, EventArgs e)
        {
            //reset PopulationSize
            PopulationSize = Convert.ToInt16(textBoxPopulation.Text);

            reset();
        }

        private void textBoxCrossover_TextChanged(object sender, EventArgs e)
        {
            PCross = Convert.ToInt16(textBoxCrossover);
            reset();
        }

        private void textBoxMutation_TextChanged(object sender, EventArgs e)
        {
            PMutate = Convert.ToInt16(textBoxMutation.Text);
            reset();
        }

        private void textBoxMinGenome_TextChanged(object sender, EventArgs e)
        {
            reset();
        }

        private void textBoxMaxGenome_TextChanged(object sender, EventArgs e)
        {
            reset();
        }
    }
}
```

Código da classe Individual:

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

```
using System.Windows.Forms;

namespace WindowsFormsApplication23
{
    enum Gene { Left, Right, Up, Down, Stop };

    class Individual : IComparable
    {
        //location
        public Gene[] DNA { get; set; }
        public double fitness { get; set; }
        private static Random rnd;
        static Individual()
        {
            rnd = new Random();
        }

        public Individual(Gene[] DNA) {
            this.DNA = DNA;
        }

        public Individual(int minGenomeSize, int maxGenomeSize)
        {

            int genomeSize = rnd.Next(minGenomeSize, maxGenomeSize + 1);
            DNA = new Gene[genomeSize];

            for (int i = 0; i < genomeSize; i++)
            {
                int c = rnd.Next(1, 5);
                switch (c)
                {
                    case 1:
                        DNA[i] = Gene.Down;
                        break;
                    case 2:
                        DNA[i] = Gene.Left;
                        break;
                    case 3:
                        DNA[i] = Gene.Right;
                        break;
                    case 4:
                        DNA[i] = Gene.Up;
                        break;
                    case 5:
                        DNA[i] = Gene.Stop;
                        break;
                }
            }
        }

        public void evaluate(Labirinto labirinto)
        {
            //First pos
            Point position = labirinto.Entry;

            //get DNA
            for (int i = 0; i < DNA.Length; i++)
            {
                switch (DNA[i])
                {
                    case Gene.Left:
                        if (position.Y == 0) break;
                        if (labirinto.grid[position.X, position.Y - 1] == 0) break;
                        position.Y--; break;
```

```
                    case Gene.Right:
                        if (position.Y == labirinto.Width - 1) break;
                        if (labirinto.grid[position.X, position.Y + 1] == 0) break;
                        position.Y++; break;

                    case Gene.Up:
                        if (position.X == 0) break;
                        if (labirinto.grid[position.X - 1, position.Y] == 0) break;
                        position.X--; break;

                    case Gene.Down:
                        if (position.X == labirinto.Height - 1) break;
                        if (labirinto.grid[position.X + 1, position.Y] == 0) break;
                        position.X++; break;

                }
            }
            double distance = Math.Sqrt(Math.Pow(labirinto.Exit.X - position.X, 2)
                + (Math.Pow(labirinto.Exit.Y - position.Y, 2)));
            fitness = 1 / distance;
        }

        public void mutate()
        {
            if (DNA.Length == 0) return;

            int geneposition = rnd.Next(0, DNA.Length);
            int anothergene = rnd.Next(0, 5);

            switch (anothergene)
            {
                case 0:
                    this.DNA[geneposition] = Gene.Left;
                    break;
                case 1:
                    this.DNA[geneposition] = Gene.Right;
                    break;
                case 2:
                    this.DNA[geneposition] = Gene.Up;
                    break;
                case 3:
                    this.DNA[geneposition] = Gene.Down;
                    break;
                case 4:
                    this.DNA[geneposition] = Gene.Stop;
                    break;
            }
        }
        public Individual[] cross(Individual ind)
        {
            int thisBreakPoint = rnd.Next(1, this.DNA.Length + 1);
            int indBreakPoint = rnd.Next(1, ind.DNA.Length + 1);

            Gene[] indA1B2 = this.DNA.Take(thisBreakPoint +
1).Concat(ind.DNA.Skip(indBreakPoint)).ToArray();
            Gene[] indA2B1 = ind.DNA.Take(thisBreakPoint +
1).Concat(this.DNA.Skip(indBreakPoint)).ToArray();

            return new Individual[] { new Individual(indA1B2), new Individual(indA2B1)};

        }
        public int CompareTo(object obj)
        {
            Individual other = obj as Individual;
```

```
            if (this.fitness > other.fitness) return -1;
            if (this.fitness < other.fitness) return 1;
            return 0;

        }
    }
}
```

Código da classe Labirinto:

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

namespace WindowsFormsApplication23
{
    class Labirinto
    {
        //para a delimitação
        public int Width{ get; set; }
        public int Height{ get; set; }
        public Point Entry { get; set; }
        public Point Exit { get; set; }

        //array
        public int[,] grid{ get; set; }

        public Labirinto(int[,]labirinto, Point entry, Point exit)
        {
            grid = labirinto;
            Width = labirinto.GetLength(1);
            Height = labirinto.GetLength(0);
            Entry = entry;
            Exit = exit;

        }
        public void draw(Panel panel, PictureBox[,]pictures) {
            //DRAW LABIRINTO
            for (int i = 0; i < this.Height; i++)
            {
                for (int j = 0; j < this.Width; j++)
                {
                    PictureBox pb = new PictureBox();
                    pb.Size = new Size(20, 20);
                    pb.Location = new Point(j * 20, i * 20);


                    switch (this.grid[i, j])
                    {
                        case 0: pb.BackColor = Color.Black; break;
                        case 1: pb.BackColor = Color.White; break;
                        case 2: pb.BackColor = Color.Blue; break;
                        case 3: pb.BackColor = Color.Red; break;
                    }

                    pictures[i, j] = pb;
                    panel.Controls.Add(pb);

                }
            }
```

```
        }

        public void redraw(Panel panel, PictureBox[,] pictures)
        {
            //DRAW LABIRINTO
            for (int i = 0; i < this.Height; i++)
            {
                for (int j = 0; j < this.Width; j++)
                {
                    switch (this.grid[i, j])
                    {
                        case 0: pictures[i, j].BackColor = Color.Black; break;
                        case 1: pictures[i, j].BackColor = Color.White; break;
                        case 2: pictures[i, j].BackColor = Color.Blue; break;
                        case 3: pictures[i, j].BackColor = Color.Red; break;
                    }
                }
            }
        }

    }
}
```