Slovenská technická univerzita
Fakulta informatiky a informačných technológií

AI project 2

b) Treasure hunt

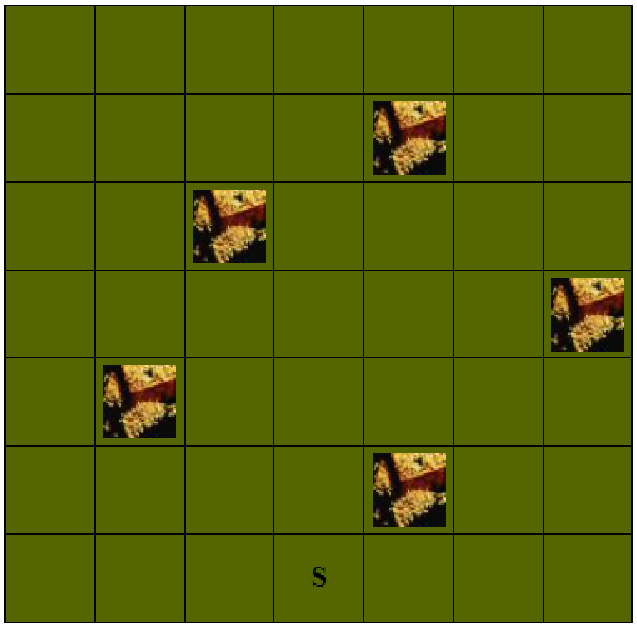
Márk Bartalos

Seminars with**: Ing. Martin Komák, PhD.**
Seminars on**: Monday 9:00-10:50**

Table of Content

Introduction

As task I had to create a genetic evolutionary algorithm that solves the problem of finding all treasures on a n x n map. The main entity in the algorithm was a virtual machine, which had 64 memory cells. Each memory cell contained a 2-bit instruction and an address to which the instruction pointed to. Each memory cell has to have an address.

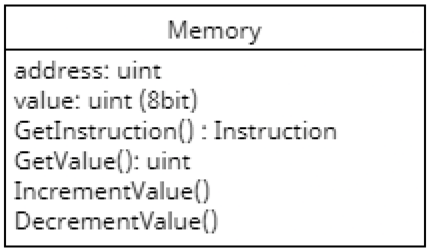


Definition of the instructions:

00 – increment
01 – decrement
10 – jump
11 – print

Definition of the data structures

Memory

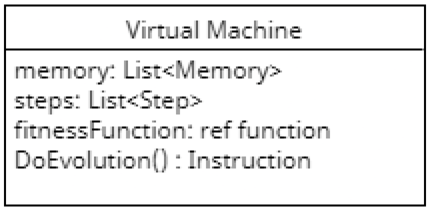


Memory

address: uint
value: uint (8bit)
GetInstruction() : Instruction
GetValue(): uint
IncrementValue()
DecrementValue()

Let's start from the most basic. The most basic data structure is a memory cell object. These memory cells are called genes. They contain an address, a value and methods for incrementing, decrementing the value, printing out the step based on the value or returning the instruction from the value. The Incrementation Decrementation and all methods that use the value property get and modify the appropriate part of the property using binary operations.

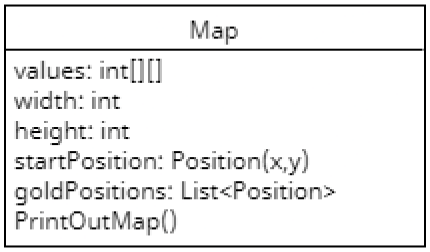
Virtual machine



Virtual machine is the entity itself in the algorithm. It contains a total of 64 memory cells which are addressed from 0-63. Also, it contains a list of steps which are generated during runtime, when calling the DoEvolution method. When the DoEvolution method is called the machine runs code from the memory cells starting from 0 all the way to the end or until 500 instructions hasn't been completed. If the instruction is a PRINT a step is generated based on the memory cell it points to. If the step is outside the map the algorithm deletes the step and stops the machine.

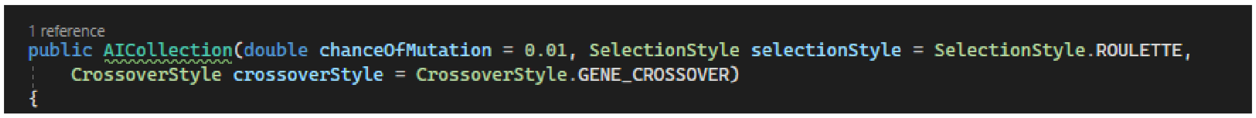
Map

The map data strucure is generated when starting the application from a text file. It contains the width, height of the map, positions of the treasures and the starting position. Also the map is used to control the virtual machine's position based on its steps, the collected gold and check the bounds.



Main part of the algorithm

The main part of the algorithm is contained in a class called AICollection. This class contains the Fitness Evolution, Selection and gene Crossover methods. During initialization this class takes 3 arguments as configuration parameters. These parameters are initialized by default.



```
1 reference
public AICollection(double chanceOfMutation = 0.01, SelectionStyle selectionStyle = SelectionStyle.ROULETTE,
    CrossoverStyle crossoverStyle = CrossoverStyle.GENE_CROSSOVER)
{
```

What these parameters mean?

Chance of mutation

During each generation each successful virtual machine and their each memory cell goes through a process called mutation. The process of mutation goes through every bit in every memory cell and the by chance a mutation can occur. This chance is determined by the chance of mutation parameter. If a mutation occurs the current bit is flipped in that memory cell.

Algorithm

```
private List<Memory> Mutate(List<Memory> completeMemory)
        {
            var random = new Random();
            var newMemory = new List<Memory>();

            foreach(var memory in completeMemory)
            {
                for(int i = 0; i < 8; i++)
                {
                    if (random.NextDouble() <= ChanceOfMutation)
                    {
                        if (!memory.Value.IsOneAtPosition(i))
                        {
                            memory.Value = (memory.Value | (uint)Math.Pow(2, 8 - i -
1));
                        }
                        else
                            memory.Value = (memory.Value & (255-(uint)Math.Pow(2, 8-
i-1)));
                    }
                }
                newMemory.Add(memory);
            }
            return newMemory;
        }
```

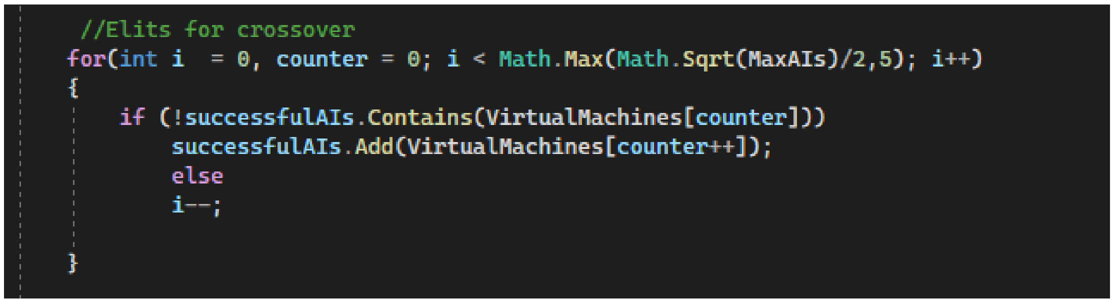
Selection style

The selection style is one of the most important parameters. The selection style determines how the new successful entities are selected.

There are two selection algorithms implemented:

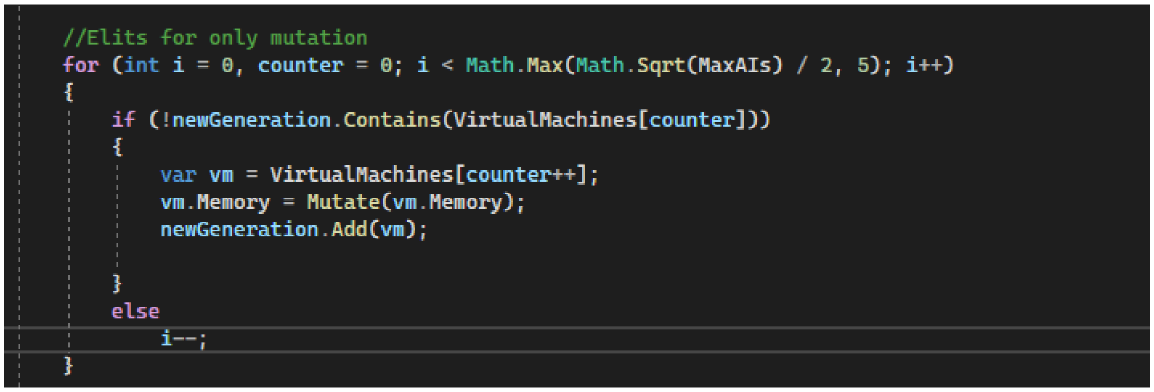
Roulette selection

Roulette selection uses success to determine how much chance has an entity to be picked. Here I used two types of elitism to be sure that the most successful entities are passed forward to the next generation. First type of elitism is that the SQRT(maxAICount)/2 or 5 (depending on which is larger) elite entities are selected and added to the successful list, where they will be crossed over to create new children.



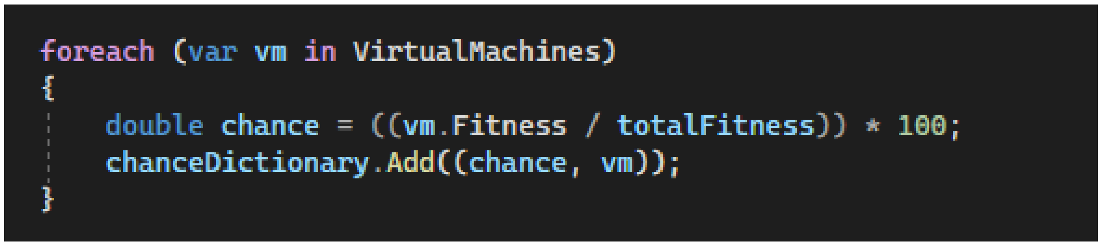
```
 //Elits for crossover
for(int i  = 0, counter = 0; i < Math.Max(Math.Sqrt(MaxAIs)/2,5); i++)
{
    if (!successfulAIs.Contains(VirtualMachines[counter]))
        successfulAIs.Add(VirtualMachines[counter++]);
        else
        i--;

}
```

The second type of elitism is that the elite entities are added straight to the new generation and are only mutated not crossed over and they do not create new children. Their count is also the same.



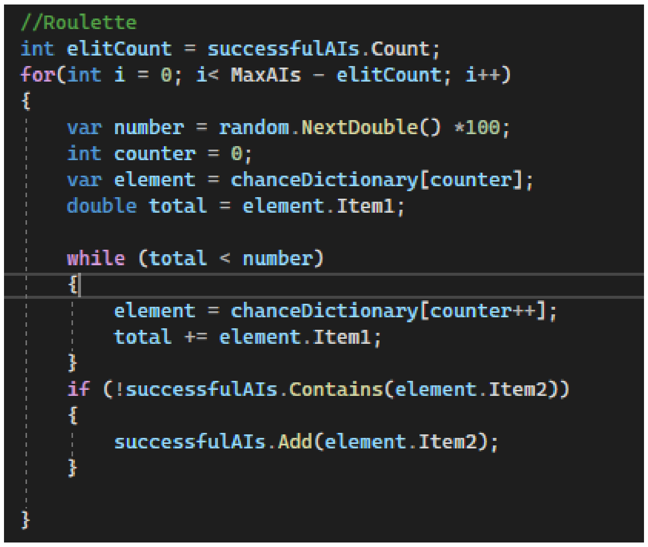
```
//Elits for only mutation
for (int i = 0, counter = 0; i < Math.Max(Math.Sqrt(MaxAIs) / 2, 5); i++)
{
    if (!newGeneration.Contains(VirtualMachines[counter]))
    {
        var vm = VirtualMachines[counter++];
        vm.Memory = Mutate(vm.Memory);
        newGeneration.Add(vm);

    }
    else
        i--;
}
```

The normally selected entities are selected by chance according to their success score. Their success score is calculated first and compared to the total success score to determine their relative success to others.



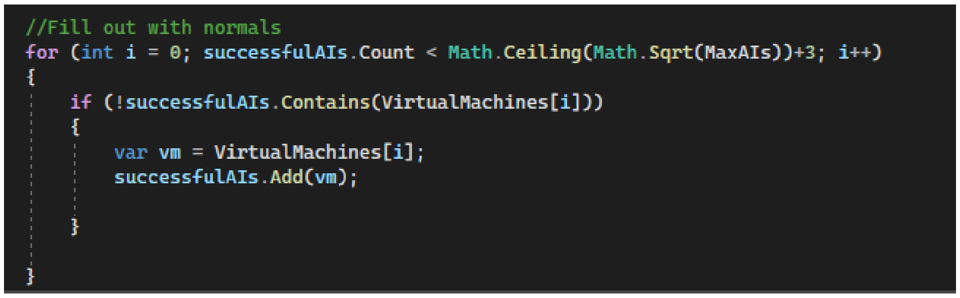
```
foreach (var vm in VirtualMachines)
{
    double chance = ((vm.Fitness / totalFitness)) * 100;
    chanceDictionary.Add((chance, vm));
}
```

After the chances have been determined a roulette selects the successful entities for the crossover based on chance and success.



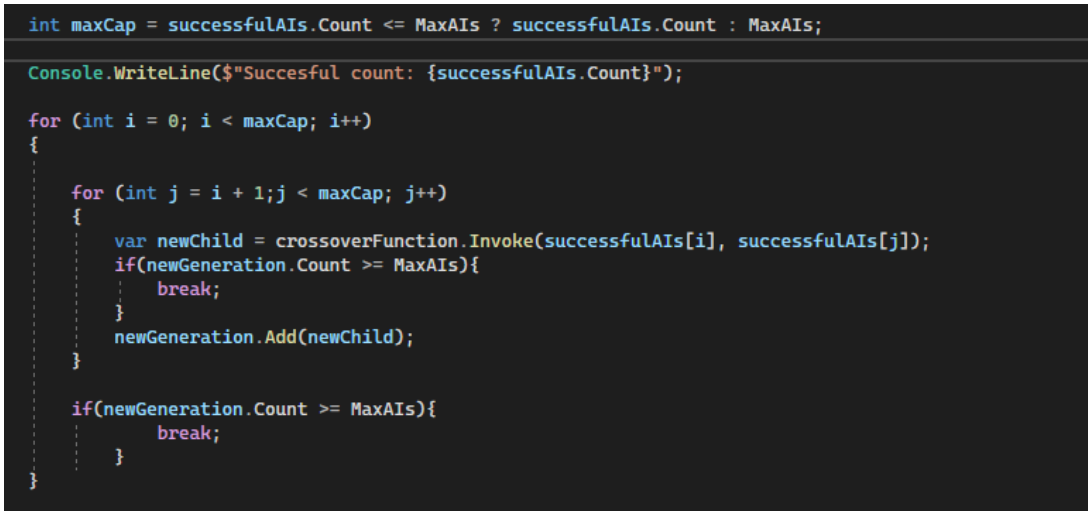
```
//Roulette
int elitCount = successfulAIs.Count;
for(int i = 0; i< MaxAIs - elitCount; i++)
{
    var number = random.NextDouble() *100;
    int counter = 0;
    var element = chanceDictionary[counter];
    double total = element.Item1;

    while (total < number)
    {
        element = chanceDictionary[counter++];
        total += element.Item1;
    }
    if (!successfulAIs.Contains(element.Item2))
    {
        successfulAIs.Add(element.Item2);
    }

}
```

Before crossover if there are not enough unique entities selected, because of the roulette one entity can be selected multiple times but only added once the successful entity list is filled with normal entities starting from the top (with the best).



```
//Fill out with normals
for (int i = 0; successfulAIs.Count < Math.Ceiling(Math.Sqrt(MaxAIs))+3; i++)
{
    if (!successfulAIs.Contains(VirtualMachines[i]))
    {
        var vm = VirtualMachines[i];
        successfulAIs.Add(vm);

    }

}
```

After the successful entities are selected comes the crossover.



```
int maxCap = successfulAIs.Count <= MaxAIs ? successfulAIs.Count : MaxAIs;

Console.WriteLine($"Succesful count: {successfulAIs.Count}");

for (int i = 0; i < maxCap; i++)
{

    for (int j = i + 1;j < maxCap; j++)
    {
        var newChild = crossoverFunction.Invoke(successfulAIs[i], successfulAIs[j]);
        if(newGeneration.Count >= MaxAIs){
            break;
        }
        newGeneration.Add(newChild);
    }

    if(newGeneration.Count >= MaxAIs){
            break;
        }
}
```

In the crossover each entity is crossed over with each other starting from the best. First the first entity is crossed over with the other entities, then the second entity is crossed over with the other entities and so on, until the new generation list is filled or all entities have been crossed over. Crossing over the entities is the responsibility of the crossover function. After crossover the old generation is replaced with the new one.

Tournament selection

In tournament selection virtual machines are selected randomly at first then order from the highest fitness score to the lowest and crossed over. 3 elites are always transferred to the new generation, which are only mutated and not crossed over.

Algorithm

```
private void DoTournamentEvolution()
        {
            var aisInTournament = new List<VirtualMachine>();
            var random = new Random();
            var requiredAiCount = Math.Max(MaxAIs / 3, Math.Sqrt(MaxAIs)+3);
            Console.WriteLine($"Required ai cound: {requiredAiCount}");

            for(int i = 0; aisInTournament.Count < requiredAiCount; i++)
            {
                var ai = VirtualMachines[random.Next(VirtualMachines.Count)];
                if (!aisInTournament.Contains(ai))
                    aisInTournament.Add(ai);
            }

            aisInTournament.Sort();
            aisInTournament.Reverse();

            var maxForNewChildren = Math.Ceiling(Math.Sqrt(MaxAIs)) + 5;

            var newGeneration = new List<VirtualMachine>();

            //Elits for only mutation
            for (int i = 0, counter = 0; i < 3; i++)
            {
                if (!newGeneration.Contains(VirtualMachines[counter]))
                {
                    var vm = VirtualMachines[counter++];
                    vm.Memory = Mutate(vm.Memory);
                    newGeneration.Add(vm);

                }
                else
                    i--;
            }

            for (int i = 0; i< maxForNewChildren; i++)
            {
                for (int j = i+1; j < maxForNewChildren; j++)
                {
                    var newChild = crossoverFunction.Invoke(aisInTournament[i],
aisInTournament[j]);
```

```
                        newGeneration.Add(newChild);

                        if (newGeneration.Count >= MaxAIs)
                            break;
                    }

                    if (newGeneration.Count >= MaxAIs)
                        break;
                }

                VirtualMachines = newGeneration;

            }
```

Crossover style

Crossover style determines how two parents produce a child. There are two types of crossover methods implemented. One is GENE_CROSSOVER and the other is CROSSOVER based on splitting the genome. Crossover determines which part of the genome will be inherited from which parent.

Gene crossover

Gene crossover takes the fitness of each parent calculates the relative success to the other parent. The genes are inherited individually and by chance which is calculated at the beginning of the function. If one parent is more 30% more successful, than it will have 30% more chance to pass its genes to the child.

At the end of the crossover the new child’s genes are also mutated.

Algorithm

```
private VirtualMachine GeneCrossover(VirtualMachine parent1, VirtualMachine parent2)
        {
            var child = new VirtualMachine(ControlBounds);
            var crossoverMemory = new List<Memory>();
            var totalFitness = parent1.Fitness + parent2.Fitness;
            var random = new Random();

            for (int i = 0; i < parent1.Memory.Count; i++)
            {
                if(random.NextDouble() < (parent1.Fitness / totalFitness))
                {
                    crossoverMemory.Add(parent1.Memory[i].DeepClone());
                }
                else
                {
                    crossoverMemory.Add(parent2.Memory[i].DeepClone());
                }

            }
            child.Memory = crossoverMemory.DeepClone();
            child.Memory = Mutate(child.Memory);

            return child;
        }
```

Crossover by split

Crossover by split is similar to the first one, it also calculates the first parents relative fitness compared to the second parent, but here the relative success rate determines how much genes are inherited in series.

Algorithm

```
private VirtualMachine Crossover(VirtualMachine parent1, VirtualMachine parent2)
        {
            var child = new VirtualMachine(ControlBounds);
            var crossoverMemory = new List<Memory>();
            var totalFitness = parent1.Fitness + parent2.Fitness;

            int crossoverPoint = (int)Math.Floor(parent1.Memory.Count *
(parent1.Fitness / totalFitness));

            for(int i = 0; i < parent1.Memory.Count; i++)
            {
                if(i >= crossoverPoint)
                    crossoverMemory.Add(parent2.Memory[i].DeepClone());
                else
                    crossoverMemory.Add(parent1.Memory[i].DeepClone());

            }
            child.Memory = crossoverMemory;
            child.Memory = Mutate(child.Memory);

            return child;
        }
```

First generation

The first generation is generated randomly. First each virtual machine is created then its 64 memory cells are generated and filled by random values and assigned the appropriate addresses from 0-63.

Fitness function

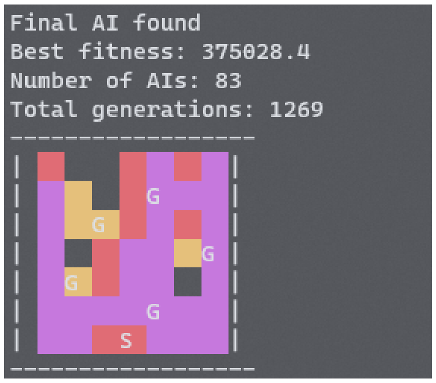
Fitness score is calculated on each entity at the beginning of each evolution. The fitness function takes in account the distance from the not collected treasures, the collected golds and the steps taken. The collected golds are rewarded exponentially the more golds it has a far better score it has. Steps are also penalized exponentially but only when the score is already high enough. The distances from the treasures are only taken into count if there are still treasure left to be found.

Algorithm

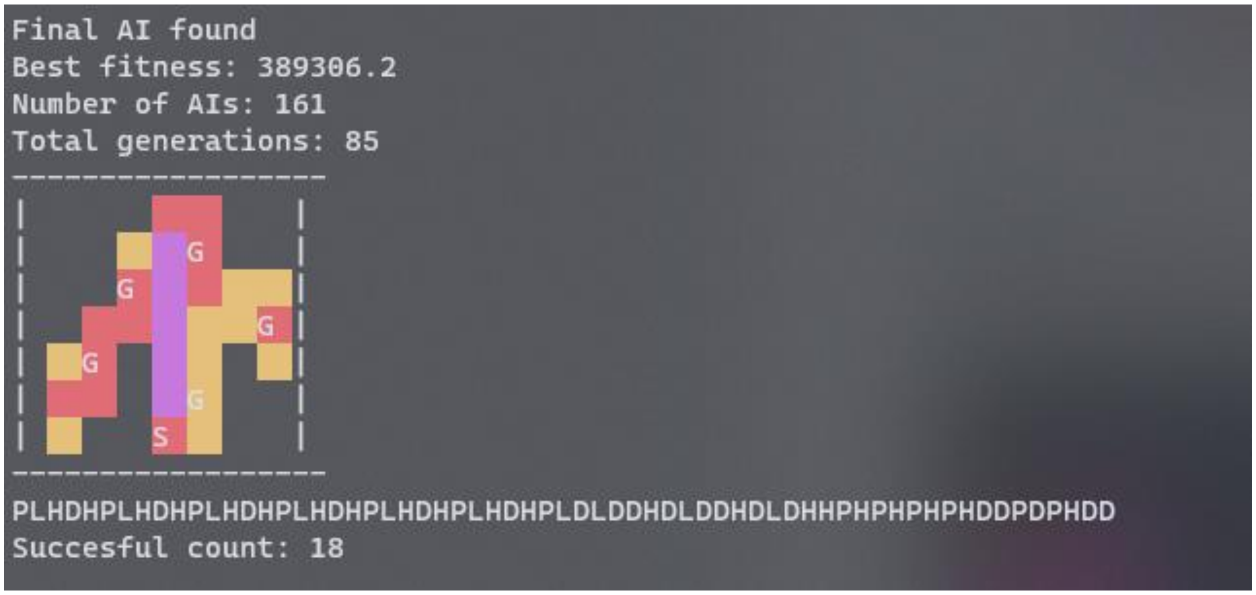
```
private double FitnessFunction(VirtualMachine machine) {

            double score = 0.01;
            if (machine.Steps == null)
                return 0.01;

            var pathData = Map.PerfromsSteps(machine.Steps);
            var position = pathData.Item1;
            var goldsCollected = pathData.Item2;
            bool wereNotCollectedGolds = false;

            score = Math.Pow(goldWorth.Invoke(),goldsCollected.Count);
            var stepPenalty = Math.Pow(1.20, machine.Steps.Count / 5);

            if (score - stepPenalty > 0.1)
                score = score - stepPenalty;
            else
                score = 0.1;

            foreach (var goldPosition in Map.GoldPositions)
            {
                if (goldsCollected.ContainsByHash(goldPosition))
                    continue;
                if(!SolutionFound)
                score += Math.Pow(1.4,goldWorth.Invoke() /
(Map.GetDistance(goldPosition, position)+1));
                wereNotCollectedGolds = true;
            }

            if( goldsCollected.Count == Map.GoldPositions.Count)
            {
                SolutionFound = true;
            }
            return score;

        }
```

Output

Output of the program are steps to the correct solution and a map displaying the map and the path the algorithm took with colors. Yellow – means algorithm crossed its only once, Red – means it was crossed twice, Purple – it was crossed multiple times.



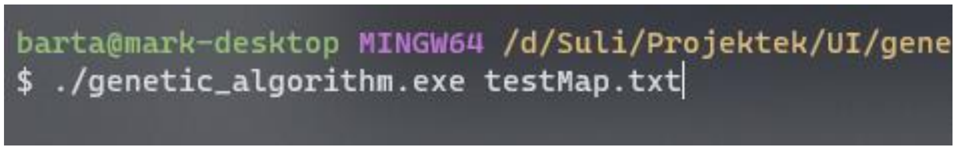
```
Final AI found
Best fitness: 375028.4
Number of AIs: 83
Total generations: 1269
-----------------
```



```
Final AI found
Best fitness: 389306.2
Number of AIs: 161
Total generations: 85
------------------
|              |
|        G     |
|    G         |
|            G |
|  G           |
|        G     |
|      S       |
------------------
PLHDHPLHDHPLHDHPLHDHPLHDHPLHDHPLDLDDHDLDDHDLDHHPHPHPHPHDDPDPHDD
Succesful count: 18
```

Input

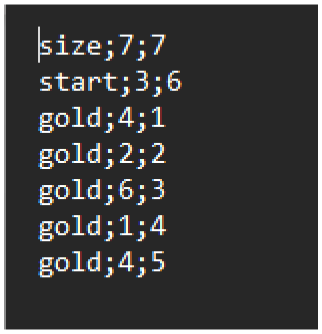
The program can be started with and input argument where the argument hast to be a path to a map file.



```
barta@mark-desktop MINGW64 /d/Suli/Projektek/UI/gene
$ ./genetic_algorithm.exe testMap.txt
```

The structure of the map

The first line has to be the size of the map, after that comes the start position and all the other lines are positions of the treasures. All the values have to be separated by semicolons and every property has to be on a new line. The last two properties signify the position the first being X (width in case of size) and second is the position on the Y axis.



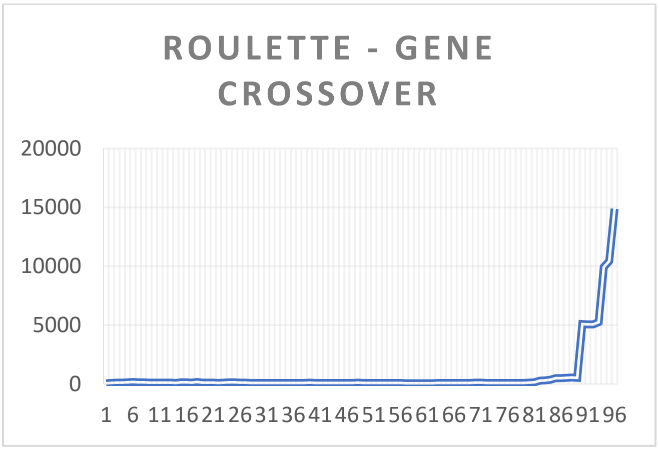
```
size;7;7
start;3;6
gold;4;1
gold;2;2
gold;6;3
gold;1;4
gold;4;5
```

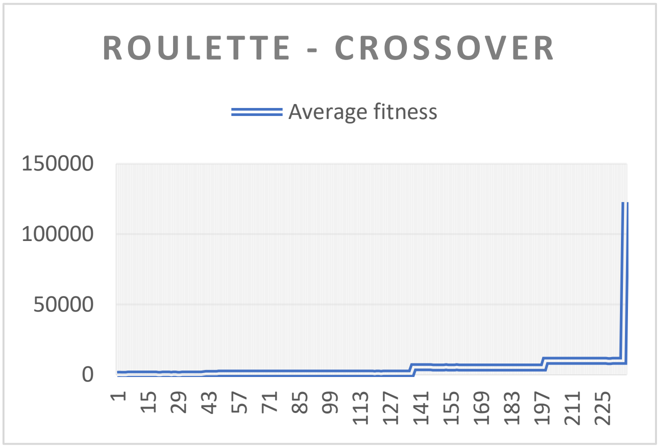
Testing

For testing the algorithm, I exported the data for 4 different runs each with a different configuration. The data contains the average score and the number of generations elapsed.

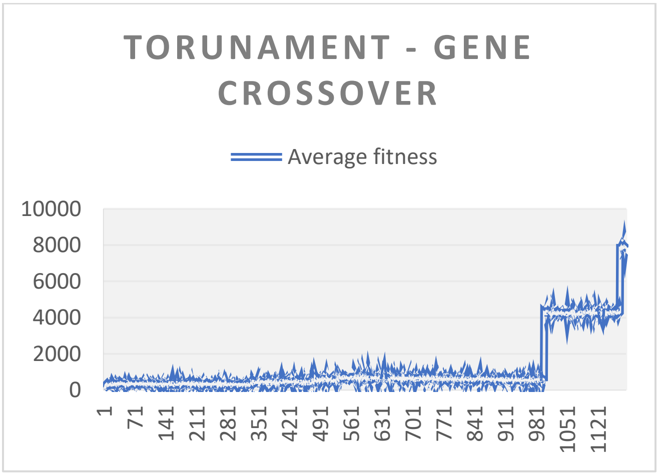
The first two graph shows the roulette selection with the two different crossover methods.

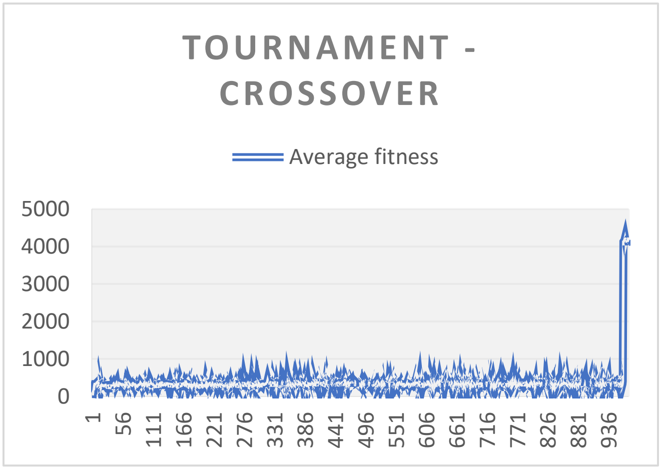
Roulette selection (individual samples)





Tournament selection (individual samples)





The test results each represent and individual run of the program on the test map. The title of the chart indicates their selection and crossover configuration. Each had 1% chance of mutation and initial generation of 100 virtual machines. Due to these are only single runs not averaged total of multiple runs the data can have some discrepansies, but none the less its representative enough to be able to see the differences between four test cases.

Conclusion

As you can see by the results that with the interval-based gene crossover there was a sudden jump of progress while in the case of individual gene crossover there was more of a slower progression. The interval-based crossover usually was slower to solve the problem, most probably due to big differences between two parents and one inheriting forward almost all their genes.

Ideas for improvement

Fitness function

The fitness function could be more fine-tuned and improved for example with capping the step penalty limit or in case of larger maps reducing the penalization. Also, the calculation of reward for each treasure also can be reduced or made not exponential but still large to motivate the virtual machines.

Roulette selection

There are some cases in roulette selection when due to random and the same VMs already existing in the successful entities list there is now always enough parents to repopulate a complete generation and, in some cases, the next generation count will be less than the previous.