Санкт-Петербургский государственный университет информационных технологий,
механики и оптики
Факультет информационных технологий и программирования
Кафедра «Компьютерные Технологии»

Отчёт по лабораторной работе «Построение управляющих автоматов с помощью генетических алгоритмов»

Сергей Мельников

16 октября 2010 г.

Оглавление

1 Введение

В лабораторной работе требуется реализовать генетический алгоритм построения автомата, решающего задачу об «Умном муравье-3». Для этого реализуется плагин для виртуальной лаборатории «GlOpt» на языке C#, который реализует конкретный тип генетического алгоритма и конкретное представление особей-автоматов.

2 Постановка задачи

Действие задачи об «Умном муравье-3» происходит на поверхности тора размером $32 \cdot 32$ клетки. В некоторых клетках находится еда. Муравей видит восемь клеток вокруг себя:

Изначально он находится в фиксированной стартовой клетке. За ход муравей может выполнить следующие действия:

- повернуть налево;
- повернуть направо;
- сделать шаг вперед и, если в новой клетке есть еда, съесть ее.

Муравей может сделать не более 200 ходов. Задача заключается в создании конечного автомата, управляющего муравьем так, чтобы тот за минимальное число ходов съел как можно больше яблок.

3 Алгоритм

3.1 Представление особи

Особью генетического алгоритма для решения данной задачи является конечный автомат Мили – детерминированный конечный автомат, каждому переходу которого сопоставлено некоторое действие (в данной задаче это либо шаг вперед, либо поворот вправо или влево). Входным алфавитом этого автомата являются значения восьми логических переменных, обозначающих наличие или отсутствие еды в каждой из видимых муравьем клеток. Автомат хранится как полная таблица переходов из каждого состояния при всех возможных воздействиях. В каждой ячейке таблицы записана информация о переходе из соответствующего ячейке состояния при соответствующем ей воздействии: номер состояния, в которое осуществляется переход, и действие, совершаемое при этом муравьем. Число состояний автомата является настраиваемым параметром алгоритма. Все генерируемые алгоритмом автоматы имеют одинаковое число состояний, не изменяющееся во время выполнения алгоритма.

3.2 Тип генетического алгоритма

При выполнении лабораторной работы использован островной генетический алгоритм. Островной генетический алгоритм отличается от обычного тем, что популяция особей разбита на несколько подпопуляций – «островов». Подпопуляции развиваются независимо друг от друга, однако периодически происходят миграции – обмены между островами некоторыми наиболее развитыми особями.

Часто при реализации данного алгоритма развитие подпопуляций распараллеливается. В работе распараллеливание не реализовано ввиду особенностей структуры лаборатории, а именно из-за того,

что класс, реализующий алгоритм, должен содержать метод, описывающий очередную итерацию алгоритма, то есть требуется последовательное выполнение шагов алгоритма.

На каждом шаге алгоритма выполняются следующие операции:

- формирование нового поколения путем скрещивания и мутаций особей из предыдущего поколения;
- мутация некоторых особей нового поколения;
- миграция;
- сортировка популяции по убыванию значений функции приспособленности особей.

Параметрами алгоритма являются:

- число островов;
- число особей на острове;
- период миграций;
- число мигрирующих особей;
- вероятность мутации;
- функция зависимости вероятности выбора, от ранга.

3.3 Генетические операции

Формирование начального поколения. Все острова заполняются автоматами, таблицы переходов которых заполнены случайными значениями.

Метод отбора. В данной работе реализован ранговый метод отбора. Очередное поколение генерируется в результате скрещивания и мутаций особей предыдущего поколения. При этом вероятность взять особь из предыдущего поколения пропорциональна некоторой функции её позиции в предыдущем поколении. Так же было исследовано поведение алгоритма, когда вероятность выбора особи зависит от её функции приспособленности (метод рулетки) и элитизма, когда несколько лучших особей переходят в новое поколение в любом случае.

Скрещивание (кроссовер). При кроссовере двух особей образуются две дочерние особи. В каждую ячейку таблицы первой дочерней особи равновероятно записывается значение соответствующей (по номеру состояния и воздействию) ячейки одного из предков, в ячейку второй — соответствующее значение другого. Аналогично, номером начального состояния первой дочерней особи равновероятно назначается номер начального состояния одного из предков, номером начального состояния второй — номер начального состояния другого предка.

Мутация. Каждая особь может мутировать. Мутация особей заключается в изменении выходного состояния и действия на одном из переходов автомата на случайные и в изменении начального состояния на произвольно определенное.

Миграция. Раз в некоторое фиксированное число поколений с каждого острова происходит миграция. Некоторое число особей с двух островов меняются друг с другом.

Оценка приспособленности. Функцией приспособленности особей в данной задаче является разность числа яблок, съеденных муравьем, управляемым автоматом-особью, и отношения числа сделанных им шагов к максимально возможному числу шагов (а именно в данном случае к двумстам). Наиболее приспособленной считается особь с максимальным значением функции приспособленности.

4 Результаты работы

В результате исследований было выяснено:

- Применение метода рулетки, вместо рангового отбора, даёт прирост скорости работы алгоритма в $1,2$–$1,5$ раза.
- Применение элитизма не даёт заметного прироста скорости работы алгоритма.
- Одним из неплохих параметров миграций, является проведение миграции один раз в 100 поколений, при этом мигрируют $0,1$ особей.
- Число островов не влияет на скорость работы алгоритма, поэтому было выбрано число островов равное 10.
- Большой размер популяции на острове замедляет работу алгоритма, однако при маленьком размере острова (5–10) решение не находится вовсе, поэтому был выбран размер острова равный 100.

В приложении на рис. 1, 2 приведены примеры выращивания автомата. Лучшая особь является автоматом с одним состоянием. Управляемый им муравей съедает все 89 яблок за 146 шагов. На основе экспериментов можно сделать вывод, применение островного алгоритма в задаче не является необходимым. Такие же результаты получаются и при применении классического подхода (одного острова).

Литература

[1] Инструкция по работе с виртуальной лабораторией GlOpt.
http://is.ifmo.ru/genalg/labs_2010-2011/GlOpt_problems.doc
http://is.ifmo.ru/genalg/labs_2010-2011/GlOpt_instruction.doc

[2] Яминов Б. Генетические алгоритмы.
http://rain.ifmo.ru/cat/view.php/theory/unsorted/genetic-2005

[3] А. А. Давыдов, Д. О. Соколов, Ф. Н. Царев. Применение генетических алгоритмов для построения автоматов мура и систем взаимодействующих автоматов мили на примере задачи об «умном муравье».
http://is.ifmo.ru/works/_2009_08_12_davydov.pdf

[4] И. Л. Каширина. Введение в эволюционное моделирование.
http://window.edu.ru/window_catalog/pdf2txt?p_id=29587

[5] Ф. Н. Царев, А. А. Шалыто. Разработка технологии генетического программирования для генерации автоматов управления системами со сложным поведением.
http://is.ifmo.ru/present/_genetic-itmo.ppt

Приложение

1 Графики

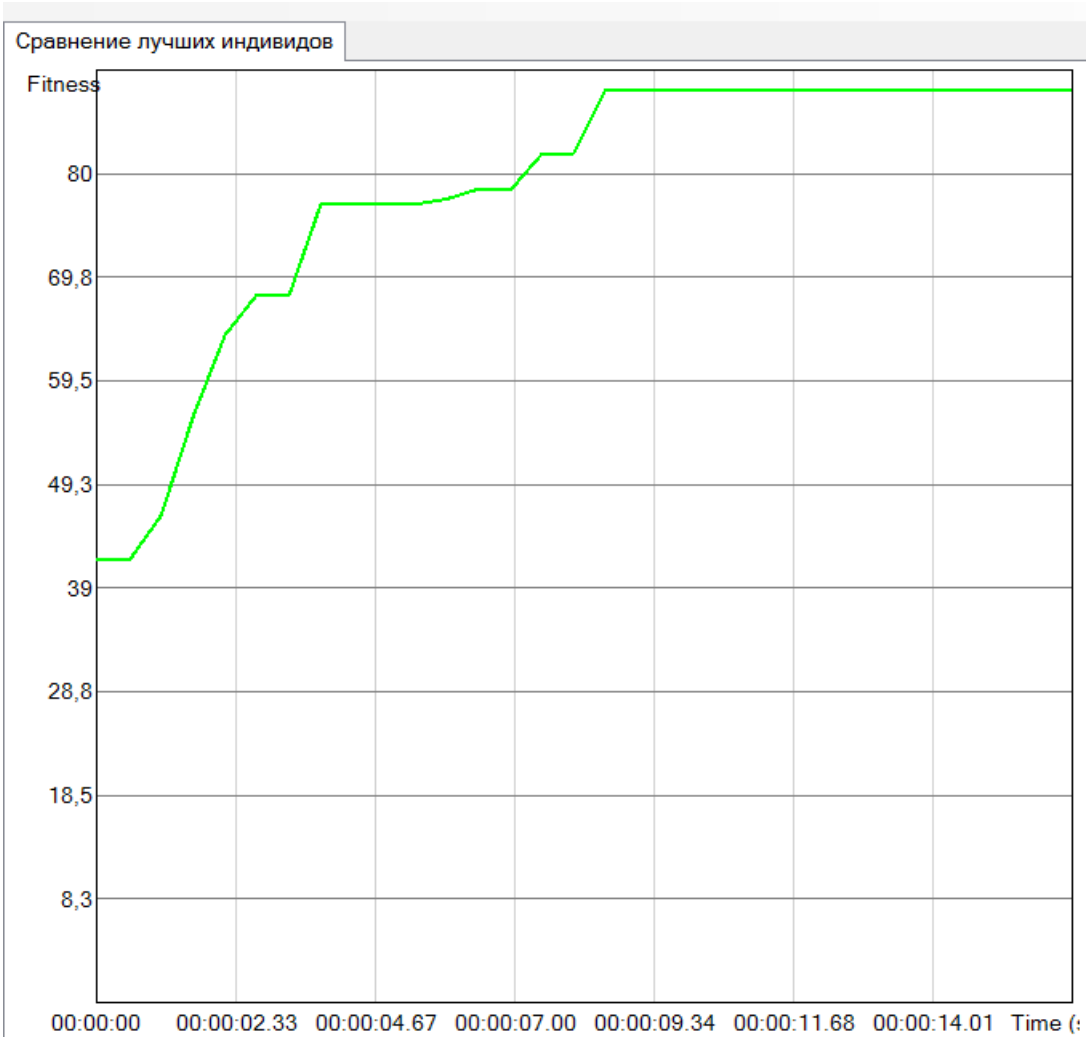

Рис. 1. График выращивания автомата с выбранными параметрами при использовании рангового отбора

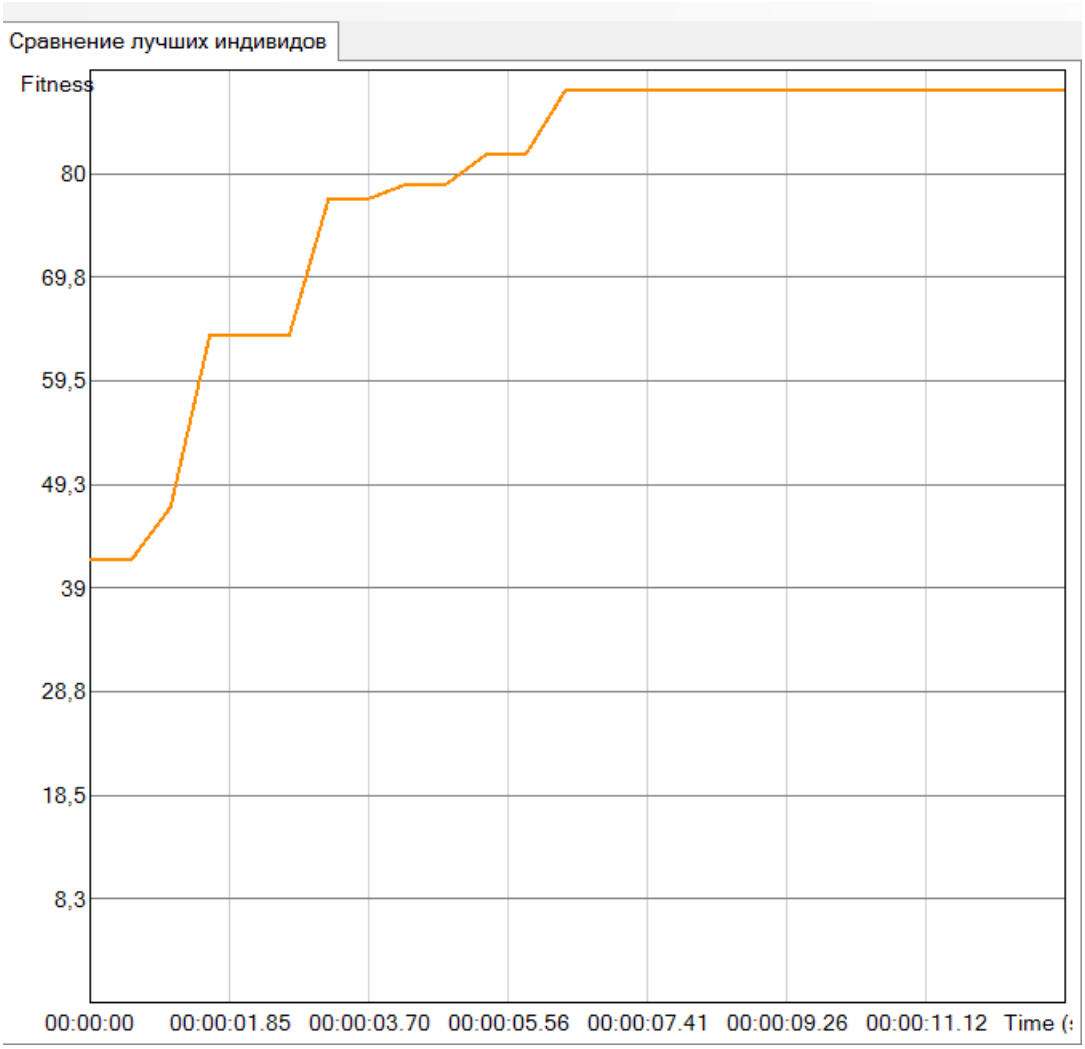

Рис. 1. График выращивания автомата с выбранными параметрами при использовании метода рулетки

2 Исходные коды

2.1 Automation.cs

```
using System;
using System.Collections;
using ArtificialAnt3.ProblemInfo;
using NewBrain;

namespace ArtificialAnt3.IndividualInfo
{
    public class Automation : Individual
    {
        public int StateNumberLength { get; private set; }
        public int TransitionInfoLength { get; private set; }
        public int StateInfoLength { get; private set; }
        public Transition[,] transitions;
```

```
        public Automation(int initialState, int statesCount, Random random)
        {
            InitialState = initialState;
            StatesCount = statesCount;
            transitions = new Transition[statesCount,256];
            for (int i = 0; i < statesCount; i++)
            {
                for (int j = 0; j < 256; j++)
                {
                    transitions[i, j] =
                    new Transition(random.Next(statesCount
                    ), (SimpleAntProblem.AntActions)random.Next(3));
                }
            }

        }
        public Automation(Automation a)
        {
            InitialState = a.InitialState;
            StatesCount = a.StatesCount;
            transitions = (Transition[,])a.transitions.Clone();
        }

        public int InitialState { get; set; }
        public int StatesCount { get; set; }


        public class Transition
        {
            public Transition(int endState, AntActions action)
            {
                EndState = endState;
                Action = action;
            }

            public int EndState { get; private set; }

            public SimpleAntProblem.AntActions Action { get; protected set; }

            public override string ToString()
            {
            }
        }


        public Transition GetTransition(int stateNumber, int state)
        {
            return transitions[stateNumber,state];
        }

    }
}
```

2.2 MyGeneticOptimizationAlgorithm.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using ArtificialAnt3.IndividualInfo;
using ArtificialAnt3.ProblemInfo;
using Genetic.Operators;
using NewBrain;
using NewBrain.Attributes;
using NewBrain.Plugins;

namespace Genetic
{
    [IndividualType(typeof(Automation))]
    public abstract class SearchOperatorOnAutomation3 : SearchOperator
    {
        public SearchOperatorOnAutomation3()
        {
            StatesCount = 1;
            MutationProbability = 0.1;
        }

        [Configurable]
        public int StatesCount { get; set; }
        [Configurable]
        public double MutationProbability { get; set; }

        public override Individual Create(Random random)
        {
            Automation automation = new Automation(0, StatesCount, random);
            return automation;
        }

        protected SimpleAntProblem.AntActions GetRandomAction
(Random random)
        {
            SimpleAntProblem.AntActions action = SimpleAntProblem.AntActions.Left;
            switch (random.Next(3))
            {
                case 0:
                    action = SimpleAntProblem.AntActions.Left;
                    break;
                case 1:
                    action = SimpleAntProblem.AntActions.Move;
                    break;
                case 2:
                    action = SimpleAntProblem.AntActions.Right;
                    break;
            }
```

```
            return action;
        }
    }

    [IndividualType(typeof(Automation))]
    public class GeneticSearchOperatorOnAutomation3 : SearchOperatorOnAutomation3, IGeneticSearch
    {
        public override string Title
        {
        }

        public override string Description
        {
            get { return ; }
        }


        public Individual Mutate(Individual oldIndividual, Random random)
        {
            Automation aa = oldIndividual as Automation;
            if (aa == null)
                return oldIndividual;
            Automation res = new Automation(aa);
            if (random.NextDouble() < MutationProbability)
            {
                res.InitialState = random.Next(aa.StatesCount);
            }


            for (int state = 0; state < aa.StatesCount; state++)
            {
                for (int c = 0; c < 256; c++)
                {
                    if (random.NextDouble() < MutationProbability)
                    {
                        res.transitions[state, c] =
                            new ArtificialAnt3.IndividualInfo.Automation.Transition
                                                (random.Next(aa.StatesCount),
                                                (SimpleAntProblem.AntActions)
                                                random.Next(3));
                    }
                }
            }

            return res;
        }

        public Individual[] Crossover(Individual[] individuals, Random random)
        {
            for (int i = 0; i < individuals.Length; i++)
            {
                individuals[i] = Mutate(individuals[i], random);
```

```
        }

        return individuals;
    }

}
[SearchOperatorType(typeof(IGeneticSearchOperator))]
public class MyGeneticOptimizationAlgorithm : Algorithm
{
    public int IslandCount = 10;
    public int IslandSize = 100;
    public int MigrationCount = 50;
    public int MigrationPeriod = 100;

    protected Random Random { get; private set; }
    protected override OptimizationDirection OptimizationDirection
    {
        get { return OptimizationDirection.Maximize; }
    }

    public override string Title
    {
        get { return My island algorithm; }
    }

    public override string Description
    {
        get { return ; }
    }

    protected IGeneticSearchOperator GeneticSearchOperator
    {
        get { return SearchOperator as IGeneticSearchOperator; }
    }

    protected override List<Individual> CurrentIndividuals { get; set; }
    protected override Individual BestIndividual { get; set; }
    private List<List<Individual>> Islands;
    private int generation;
    protected override void Initialize()
    {
        generation = 0;
        Random = new Random();
        CurrentIndividuals = new List<Individual>(5);
        Islands = new List<List<Individual>>();
        for (int i = 0; i < IslandCount; i++)
        {
            List<Individual> l = new List<Individual>();
            for (int j = 0; j < IslandSize; j++)
            {
                l.Add(GeneticSearchOperator.Create(Random));
            }
```

```
                l.Sort();
                l.Reverse();
                Islands.Add(l);
            }

        }


        protected static double F(double x)
        {
            return x;
        }

        static int randomWeightIndexOf(double[] a, double sum, Random rand)
        {
            double r = rand.NextDouble() * sum;
            int j = 0;
            while (r > a[j])
            {
                r -= a[j];
                j++;
            }
            return j;
        }
        protected List<Individual> WorkIsland(List<Individual> a, Random random)
        {
            a.Sort();
            a.Reverse();
            double[] b = new double[a.Count];
            double sum = 0;
            for (int i = 0; i < a.Count; i++)
            {
                //b[i] = (a.Count- i+0.0)*(a.Count- i+0.0)*(a.Count- i+0.0);
                b[i] = F(a[i].Fitness);
                sum += b[i];
            }
            List<Individual> newGen = new List<Individual>();
            for (int i = 0; i < 10;i++ )
            {
                newGen.Add(a[i]);
            }
            while (newGen.Count < a.Count)
            {
                if (random.Next(5) == 0)
                {
                    newGen.Add(GeneticSearchOperator.Mutate(a[randomWeightIndexOf(b, sum,
random)], random));
                }
                else
                {
                    Individual x = a[randomWeightIndexOf(b, sum, random)];
                    Individual y = a[randomWeightIndexOf(b, sum, random)];
```

```
                foreach (var s in GeneticSearchOperator.Crossover(
                new Individual[] { x, y }, random))
                {
                    newGen.Add(s);
                }
            }
        }
        while (newGen.Count > a.Count)
        {
            newGen.RemoveAt(newGen.Count - 1);
        }
        a.Sort();
        a.Reverse();
        return newGen;
    }

    protected override void NextIteration()
    {
        for (int i = 0; i < IslandCount; i++)
        {
            Islands[i] = WorkIsland(Islands[i], Random);
        }
        generation++;
        Console.WriteLine(generation);
        if (generation % MigrationPeriod == 0)
        {
            for (int i = 0; i < MigrationCount; i++)
            {
                int x = Random.Next(IslandCount);
                int xp = Random.Next(IslandSize);
                int y = Random.Next(IslandCount);
                int yp = Random.Next(IslandSize);
                Individual ind = Islands[x][xp];
                Islands[x][xp] = Islands[y][yp];
                Islands[y][yp] = ind;
            }
        }
        CurrentIndividuals = new List<Individual>();
        for (int i = 0; i < IslandCount; i++)
        {
            for (int j = 0; j < Islands[i].Count; j++)
            {
                CurrentIndividuals.Add(Islands[i][j]);
            }
        }
        CurrentIndividuals.Sort();
        CurrentIndividuals.Reverse();
        BestIndividual = CurrentIndividuals[0];
    }

    protected override bool Terminated()
    {
```

```
                return false;
            }
        }
    }
```