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

Фильченко Н.В.

Отчет по лабораторной работе «Применение генетических алгоритмов для построения автоматов в задаче об умном муравье-3»

Вариант №12

Санкт-Петербург
2011

Содержание

1. Введение

В данной лабораторной работе сравнивается эффективность работы генетического алгоритма при задании автомата полными и сокращенными таблицами. Для решения этой задачи написана программа на C++ и несколько скриптов.

1.1. Постановка задачи

Необходимо исследовать, при использовании какого способа задания автомата (полными или сокращенными таблицами) найдется лучший автомат Мура, решающий задачу об «умном муравье-3». Для этого необходимо по очереди на различных наборах случайных полей запускать генетический алгоритм с различными способами задания автомата.

1.2. Задача об «умном муравье-3»

Даны случайное поле размера 32x32, вероятность нахождения еды в каждой клетке которого одинакова (параметр задачи), и муравей, который видит восемь клеток перед собой (рис. 1). Муравей может совершить следующие действия: пойти вперед (если в клетке есть еда, он ее съедает), повернуть налево, повернуть направо, ничего не делать. Требуется построить автомат Мура, который съест максимальное количество еды (в процентах от количества еды на поле) за 200 шагов.

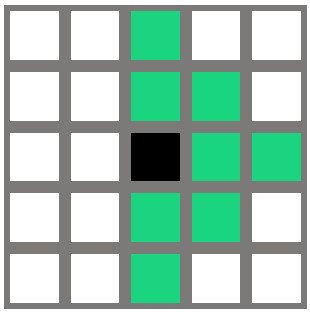

Рис. 1: Видимые клетки

2. Реализация

Программа состоит из небольшой программы на C++, реализующей непосредственно построение автомата с помощью генетического алгоритма, и набора скриптов для обработки результатов.

2.1. Структура программы

- automaton
  - moore
    - full_table – реализация автомата Мура, заданного полными таблицами
    - reduced_table – реализация автомата Мура, заданного сокращеннымии таблицами
  - input – интерфейс входа автомата
  - output – интерфейс выхода автомата
  - representation – интерфейс автомата
- problem
- ant3
  - model – модель окружения, используемая при решении задачи об умном муравье-3
- model – интерфейс модели окружения
- settings
  - settings_manager – менеджер настроек
- algorithm – реализация генетического алгоритма с ранговым отбором
- generation – поколение генетического алгоритма
- main – основной исполняемый файл, запускает обе реализации автомата мура на случайно сгенерированном наборе полей

2.2. Описание алгоритма

1. Создание первого поколения из случайно сгенерированных особей;
2. генерация следующего поколения:
   2.1. вычисление функции приспособленности;
   2.2. элитизм (заданная доля особей переходит в следующее поколение без изменений);
   2.3. ранговый отбор;
   2.4. мутация;
3. пока количество поколений меньше заданного, переходим ко второму пункту.

2.3. Описание составных частей алгоритма

2.3.1. Отбор

Список особей отсортирован по убыванию значения функции приспособленности. Проходя от начала к концу с вероятностью p/N, где p – настраиваемая константа, N – размер поколения, особь проходит на следующий этап, в противном случае переходим к следующей особи. Если достигнут конец списка, а особь не выбрана – отбор продолжается с начала.

2.3.2. Скрещивание

Полные таблицы

1. Состояния автоматов нумеруются в порядке обхода DFS;
2. для каждой пары соответсвующих состояний выбираются одна или две (в зависимости от настроек) позиции в таблице переходов;
3. особи обмениваются участками таблицы между выбранными позициями;
4. с вероятностью 50% происходит обмен выходными воздействиями состояний.

2.3.2.1. Сокращенные таблицы

1. Состояния автоматов нумеруются в порядке обхода DFS;
2. для каждой пары соответсвующих состояний выбираются одна или две (в зависимости от настроек) позиции в таблице переходов;
3. особи обмениваются участками таблицы между выбранными позициями;
4. несовпадающие номера значимых входов случайным образом распередляются между потомками;
5. с вероятностью 50% происходит обмен выходными воздействиями состояний.

2.3.3. Мутация

Полные таблицы

1. Для каждого состояния с заданной вероятностью изменяется выходное воздействие на случайное;
2. для каждого состояния с заданной вероятностью изменяется переход по каждому набору входов на случайный.

Сокращенные таблицы

1. Для каждого состояния с заданной вероятностью изменяется выходное воздействие на случайное;
2. для каждого состояния с заданной вероятностью изменяется номер значимого входа на случайный;
3. для каждого состояния с заданной вероятностью изменяется переход по каждому набору входов на случайный.

2.3.4. Функция приспособленности

Значение функции вычисляется как среднее по случайно сгенерированному перед запуском алгоритма набору полей.

Для каждого поля $f = \frac{100 \cdot E}{K} + \frac{1}{S}$, где E – количество съеденной еды, K – количество еды на поле, S – шаг, на котором съедена последняя единица еды.

3. Результаты

Было произведено три серии из 10 запусков: одна при 300 поколениях (рис. 3) и две с 1000 поколений (рис. 4–5). При использовании полных таблиц алгоритм лучше сходится к ответу в большинстве экспериментов, но требует гораздо больших вычислительных мощностей. В нескольких запусках алгоритм при использовании сокращенных таблиц показал сильно отличающиеся результаты (рис. 2), это произошло при большой мутации, к которой сокращенные таблицы более чувствительны, чем полные. Остальные запуски сошлись к приблизительно одинаковому значению функции приспособленности – 35% (рис. 3–5).

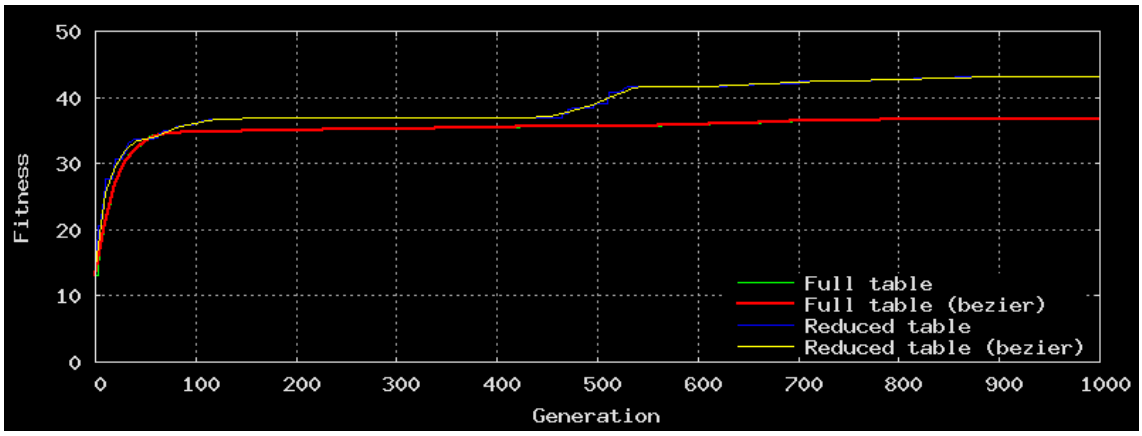

Рис. 2: Резское увеличение значеня функции приспособленности после большой мутации сокращенных таблиц (10 запуск третьей серии)

4. Протоколы запусков

4.1. Серия 1

4.2. Настройки

Вероятность еды в клетке	0.050
Ширина поля	32
Высота поля	32
Количество полей	50
Количесвто шагов	200
Количество состояний (полные таблицы)	15
Одноточечный кроссовер (полные таблицы)	0
Вероятность мутации состояния (полные таблицы)	0.149
Количество мутирующий состояний (полные таблицы)	0
Количесвто состояний (сокращенные таблицы)	15
Одноточечный кроссовер (сокращенные таблицы)	1
Вероятность мутации состояния (сокращенные таблицы)	0.149
Количество мутирующий состояний (сокращенные таблицы)	0
Количество значимых параметров (сокращенные таблицы)	3
Размер поколения	100
Количество поколений	300
Вероятность мутации	0.070
Количество поколений между большими мутациями	70
Вероятность большой мутации	0.699

4.2.1. Усреднение

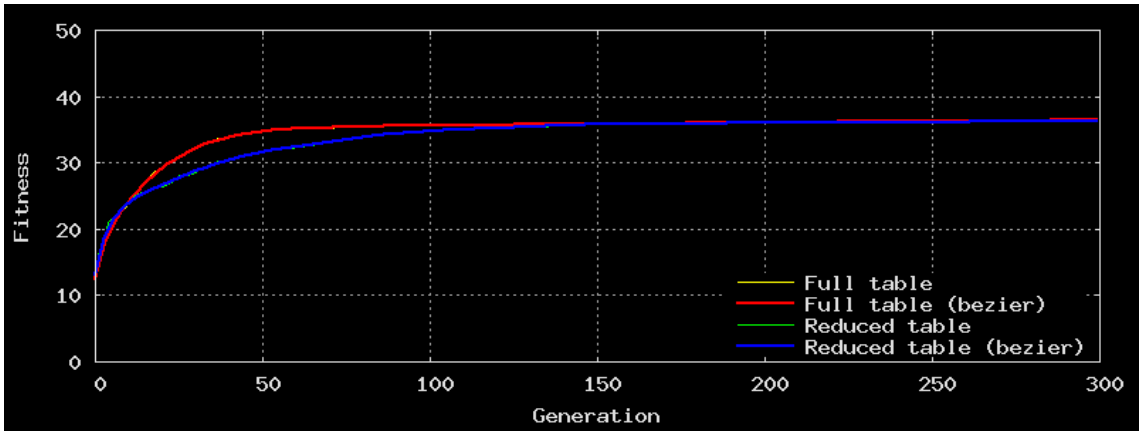

Рис. 3: Усреднение по серии запусков

4.3. Серия 2

4.4. Настройки

Вероятность еды в клетке	0.050
Ширина поля	32
Высота поля	32
Количество полей	100
Количесвто шагов	200
Количество состояний (полные таблицы)	20
Одноточечный кроссовер (полные таблицы)	0
Вероятность мутации состояния (полные таблицы)	0.149
Количество мутирующий состояний (полные таблицы)	0
Количесвто состояний (сокращенные таблицы)	20
Одноточечный кроссовер (сокращенные таблицы)	1
Вероятность мутации состояния (сокращенные таблицы)	0.149
Количество мутирующий состояний (сокращенные таблицы)	0
Количество значимых параметров (сокращенные таблицы)	3
Размер поколения	100
Количество поколений	1000
Вероятность мутации	0.070
Количество поколений между большими мутациями	70
Вероятность большой мутации	0.699

4.4.1. Усреднение

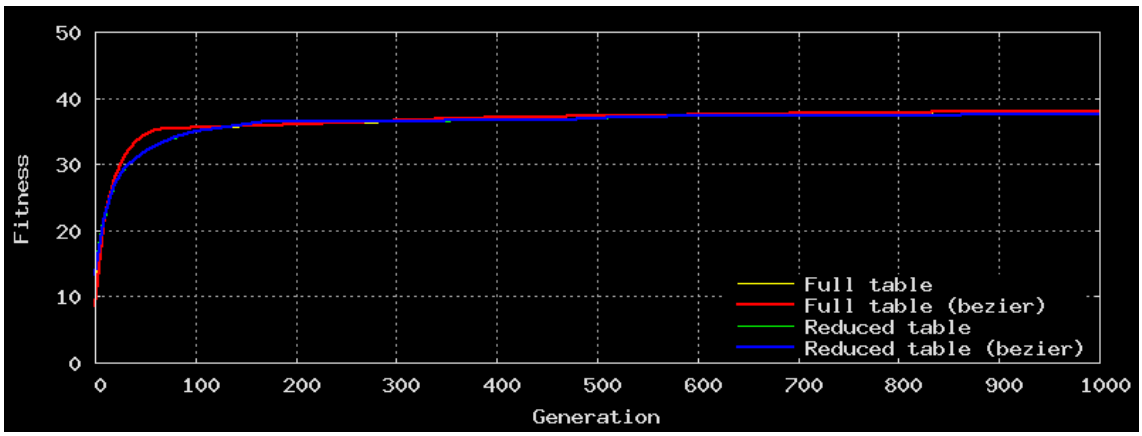

Рис. 4: Усреднение по серии запусков

4.5. Серия 3

4.6. Настройки

Вероятность еды в клетке	0.050
Ширина поля	32
Высота поля	32
Количество полей	100
Количесвто шагов	200
Количество состояний (полные таблицы)	20
Одноточечный кроссовер (полные таблицы)	0
Вероятность мутации состояния (полные таблицы)	0.149
Количество мутирующий состояний (полные таблицы)	0
Количесвто состояний (сокращенные таблицы)	20
Одноточечный кроссовер (сокращенные таблицы)	1
Вероятность мутации состояния (сокращенные таблицы)	0.149
Количество мутирующий состояний (сокращенные таблицы)	0
Количество значимых параметров (сокращенные таблицы)	3
Размер поколения	100
Количество поколений	1000
Вероятность мутации	0.070
Количество поколений между большими мутациями	70
Вероятность большой мутации	0.699

4.6.1. Усреднение

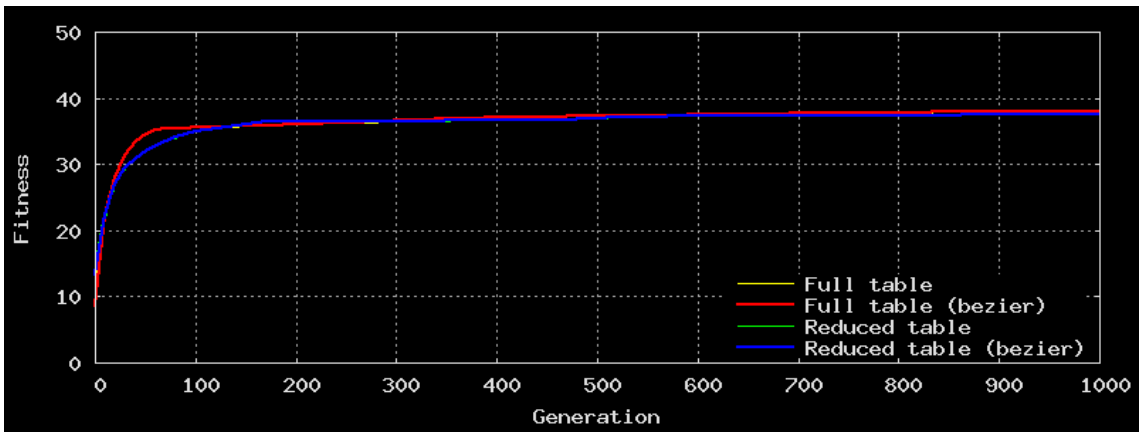

Рис. 5: Усреднение по серии запусков

5. Источники

1. Исходный код программы, скрипты и результаты запусков https://bitbucket.org/finomen/genethic
2. Пакет gnuplot http://www.gnuplot.info

6. Исходный код

6.0.1.1. main

main.cpp

```
#include "global.h"
#include "algorithm.h"
#include "problem/ant3/model.h"
#include "automaton/moore/full_table.h"
#include "automaton/moore/reduced_table.h"
#include <boost/functional/factory.hpp>
#include <boost/make_shared.hpp>
#include <boost/bind.hpp>
#include <boost/algorithm/string.hpp>
#include <iostream>
#include <fstream>
#include "settings/settings_manager.h"
#include <vector>
boost::shared_ptr<automaton::representation> ft()
{
    return boost::make_shared<automaton::moore::full_table>();
}

boost::shared_ptr<automaton::representation> rt()
{
    return boost::make_shared<automaton::moore::reduced_table>();
}

int main(int argc, const char * argv[])
{
    for (size_t i = 1; i < argc; ++i)
    {
        typedef std::vector<std::string> split_vector_type;
        split_vector_type SplitVec;
        std::string s(argv[i]);
        boost::algorithm::split(SplitVec, s, boost::algorithm::is_any_of(" ="), boost::↩
            algorithm::token_compress_on );
        settings::settings_manager::instance().override(SplitVec[0], SplitVec.back());
    }

    std::cout << settings::settings_manager::instance().data() << std::endl;
    std::fstream out("settings.txt", std::ios::out);
    out << settings::settings_manager::instance().data() << std::endl;
    out.close();
    boost::shared_ptr<problem::ant3::model> m = boost::make_shared<problem::ant3::model>();
    algorithm f(boost::bind(&ft), m);
    f.run("full_table");
    algorithm r(boost::bind(&rt), m);
    r.run("reduced_table");
    return 0;
}
```

6.0.1.2. generation

generation.h

```
/*
 * generation.h
 *
 *  Created on: Oct 14, 2011
 *      Author: Filchenko Nikolay
```

```
 */

#ifndef GENERATION_H_
#define GENERATION_H_

#include <boost/shared_ptr.hpp>
#include <boost/function.hpp>
#include "individual.h"
#include <vector>
#include "problem/model.h"

class generation
{
public:
    generation(boost::function<boost::shared_ptr<individual>()> const & factory);
    boost::shared_ptr<generation> next(boost::shared_ptr<problem::model> const & model, bool bmut = 0) const;
    double best_fitness() const;
    double avg_fitness() const;
    double worst_fitness() const;

    boost::shared_ptr<individual> best_individual();

private:
    generation(std::vector<boost::shared_ptr<individual> > const &);
    mutable std::vector<boost::shared_ptr<individual> > data;
};

#endif /* GENERATION_H_ */
```

generation.cpp

```
#include "generation.h"
#include <algorithm>
#include <boost/make_shared.hpp>
#include "settings/settings_manager.h"

#include <boost/thread.hpp>

SM_ADD_PARAM(size_t, gsize, 100);
SM_ADD_PARAM(size_t, threads, 8);
SM_ADD_PARAM(double, elite, 0.02);
SM_ADD_PARAM(double, mutation_p, 0.07);
SM_ADD_PARAM(double, mutation_bp, 0.7);

#define N_THREADS 8

bool less(boost::shared_ptr<individual> const & o1, boost::shared_ptr<individual> const & o2)
{
    return (*o1) < (*o2);
}

generation::generation(std::vector<boost::shared_ptr<individual> > const & d) : data(d)
{
}

generation::generation(boost::function<boost::shared_ptr<individual>()> const & factory)
{
    size_t gs = SM_PARAM(size_t, gsize);
    for (size_t i = 0; i < gs; ++i)
    {
        data.push_back(factory());
    }
}

void executor(std::vector<boost::shared_ptr<individual> > const & data, boost::shared_ptr<problem::model> const & model, size_t ipos)
{
        size_t gs = SM_PARAM(size_t, gsize);
        for (size_t i = ipos; i < gs; i += N_THREADS)
        {
            data[i]->fitness = model->fitness(data[i]);
        }
}

boost::shared_ptr<generation> generation::next(boost::shared_ptr<problem::model> const & model, bool bp) const
```

```
{
    model->generation();
    size_t gs = SM_PARAM(size_t, gsize);
    size_t threads = SM_PARAM(size_t, threads);

    std::vector<boost::shared_ptr<boost::thread> > workers;

    for (size_t i = 0; i < threads; ++i)
    {
        workers.push_back(boost::make_shared<boost::thread>(boost::bind(executor, data, model, ↩
            i)));
    }

    for (size_t i = 0; i < threads; ++i)
    {
        workers[i]->join();
    }

    std::sort(data.begin(), data.end(), less);

    size_t el = SM_PARAM(double, elite) * gs;
    double pm = SM_PARAM(double, mutation_p);
    double bpm = SM_PARAM(double, mutation_bp);

    std::vector<boost::shared_ptr<individual> > nd;

    for (size_t i = 0; i < el; ++i)
    {
        nd.push_back(data[i]);
    }

    while (nd.size() < gs)
    {
        size_t id1 = 0;
        size_t id2 = 0;

        while (!random(0.03))
            id1 = (id1 + 1) % data.size();

        while (!random(0.03))
            id2 = (id2 + 1) % data.size();

        assert(id1 < data.size());
        assert(id2 < data.size());

        std::pair<boost::shared_ptr<individual>,boost::shared_ptr<individual> > ni =
            data[id1]->corssover(data[id2]);
        if (random(bp ? bpm : pm))
        {
            ni.first = ni.first->mutate();
        }
        if (random(bp ? bpm : pm))
        {
            ni.second = ni.second->mutate();
        }

        nd.push_back(ni.first);
        nd.push_back(ni.second);
    }

    if (nd.size() > gs)
        nd.pop_back();

    return boost::shared_ptr<generation>(new generation(nd));
}


double generation::best_fitness() const
{
    return data[0]->fitness;
}

double generation::worst_fitness() const
{
    return data.back()->fitness;
}

double generation::avg_fitness() const
{
```

```
122     size_t gs = SM_PARAM(size_t, gsize);
123     double r = 0;
124     for (size_t i = 0; i < gs; ++i)
125         r += data[i]->fitness;
126     return r / gs;
127 }
128
129 boost::shared_ptr<individual> generation::best_individual()
130 {
131     return data[0];
132 }
```

6.0.1.3. algorithm

algorithm.h

```
1  #ifndef _H_ALGORITHM_
2  #define _H_ALGORITHM_
3
4  #include "global.h"
5  #include "problem/model.h"
6  #include "individual.h"
7  #include "generation.h"
8
9  class algorithm
10 {
11 public:
12     algorithm(boost::function<boost::shared_ptr<individual>()> const & individual_factory,
13             boost::shared_ptr<problem::model> const & mod);
14     void run(const char * filename);
15
16 private:
17     boost::function<boost::shared_ptr<individual>()> factory_;
18     boost::shared_ptr<problem::model> _model;
19 };
20
21 #endif
```

algorithm.cpp

```
1  #include "algorithm.h"
2  #include <iostream>
3  #include <boost/make_shared.hpp>
4  #include "settings/settings_manager.h"
5  #include <fstream>
6  #include "automaton/representation.h"
7  SM_ADD_PARAM(size_t, gens, 200);
8  SM_ADD_PARAM(size_t, gbmut, 70);
9
10 algorithm::algorithm(boost::function<boost::shared_ptr<individual>()> const & ↩
       individual_factory,
11             boost::shared_ptr<problem::model> const & mod) : _model(mod), factory_(↩
                   individual_factory)
12 {
13 }
14
15 void algorithm::run(const char * filename)
16 {
17     boost::shared_ptr<generation> g(boost::make_shared<generation>(factory_));
18     boost::shared_ptr<generation> ng;
19     size_t gs = SM_PARAM(size_t, gens);
20     size_t gbm = SM_PARAM(size_t, gbmut);
21     std::fstream out((std::string(filename) + ".log").c_str(), std::ios::out);
22
23     for (size_t i = 0; i < gs; ++i)
24     {
25         ng = g->next(_model, (i % gbm == 0));
26         std::cout << g->best_fitness() << " : " << g->avg_fitness() << std::endl;
27         out << g->best_fitness() << " " << g->avg_fitness() << " " << g->worst_fitness() <<  "\↩
               n";
28         g = ng;
29     }
30
```

```
    _model->fitness(g->best_individual(), true, filename);

    boost::dynamic_pointer_cast<automaton::representation>(g->best_individual())->draw((std::↩
        string(filename) + "_best_result.log").c_str());

    out << std::flush;
    out.close();
}
```

6.0.1.4. individual

individual.h

```
/*
 * individual.h
 *
 *  Created on: Oct 14, 2011
 *      Author: Filchenko Nikolay
 */

#ifndef INDIVIDUAL_H_
#define INDIVIDUAL_H_

#include <boost/shared_ptr.hpp>

class individual
{
public:
    virtual ~individual() {};
    virtual boost::shared_ptr<individual> mutate() const = 0;
    virtual std::pair<boost::shared_ptr<individual>, boost::shared_ptr<individual> > corssover(↩
        boost::shared_ptr<individual> const & o) const = 0;

    mutable double fitness;
    bool operator <(individual const & i)
    {
        return fitness > i.fitness;
    }
};

#endif /* INDIVIDUAL_H_ */
```

6.0.1.5. global

global.h

```
/*
 * global.h
 *
 *  Created on: Oct 14, 2011
 *      Author: Filchenko Nikolay
 */

#ifndef GLOBAL_H_
#define GLOBAL_H_

#include <cmath>
#include <math.h>
#include <time.h>
#include <cstdlib>

inline void rinit()
{
    static bool init = false;
    if (!init)
    {
        srand(time(NULL));
        init = true;
    }
}

inline bool random(double p)
```

```
{
    rinit();
    return (static_cast<double>(rand()) / RAND_MAX) <= p;
}

inline size_t random(int max)
{
    return (rand() % max);
}

inline size_t random(size_t max)
{
    return (rand() % max);
}

#endif /* GLOBAL_H_ */
```

6.1. automaton

6.1.0.6. input

input.h

```
/*
 * input.h
 *
 *  Created on: Oct 14, 2011
 *      Author: Filchenko Nikolay
 */

#ifndef INPUT_H_
#define INPUT_H_

#include <cstring>

namespace automaton {

class input
{
public:
    virtual bool x(size_t i) const = 0;
};

}

#endif /* INPUT_H_ */
```

6.1.0.7. output

output.h

```
/*
 * output.h
 *
 *  Created on: Oct 14, 2011
 *      Author: Filchenko Nikolay
 */

#ifndef OUTPUT_H_
#define OUTPUT_H_

namespace automaton {

#include <cstring>

class output
{
public:
    virtual void z(size_t) const = 0;
};

```

```
}

#endif /* OUTPUT_H_ */
```

6.1.0.8. representation

representation.h

```
/*
 * representation.h
 *
 *  Created on: Oct 14, 2011
 *      Author: Filchenko Nikolay
 */

#ifndef REPRESENTATION_H_
#define REPRESENTATION_H_

#include "automaton/input.h"
#include "automaton/output.h"
#include "individual.h"


namespace automaton {

class representation : public individual
{
public:
    virtual size_t get_states_count() const = 0;
    virtual size_t input_event(input const & i, output & o, size_t state) const = 0;
    virtual ~representation() {}
    virtual void draw(const char * filename) = 0;
};

}

#endif /* REPRESENTATION_H_ */
```

6.1.1. moore

6.1.1.1. full_table

full_table.h

```
/*
 * full_table.h
 *
 *  Created on: Oct 14, 2011
 *      Author: Filchenko Nikolay
 */

#ifndef FULL_TABLE_H_
#define FULL_TABLE_H_

#include "automaton/representation.h"
#include <vector>

namespace automaton {

namespace moore {

class full_table : public representation
{
public:
    full_table();
    virtual boost::shared_ptr<individual> mutate() const;
    virtual std::pair<boost::shared_ptr<individual>, boost::shared_ptr<individual> > corssover(↩
        boost::shared_ptr<individual> const & o) const;
    virtual size_t get_states_count() const;
    virtual size_t input_event(input const & i, output & o, size_t state) const;
    virtual void draw(const char * filename);
private:
```

```
28      full_table(const full_table * o);
29      full_table(std::vector<std::pair<size_t, std::vector<size_t> > > const & a_);
30      std::vector<std::pair<size_t, std::vector<size_t> > > a;
31  };
32
33  }
34
35  }
36
37  #endif /* FULL_TABLE_H_ */
```

full_table.cpp

```
1   /*
2    * full_table.cpp
3    *
4    *  Created on: Oct 14, 2011
5    *      Author: Filchenko Nikolay
6    */
7
8   #include "automaton/moore/full_table.h"
9
10  #include "settings/settings_manager.h"
11
12  #include "global.h"
13  #include <iostream>
14  #include <fstream>
15  #include <stack>
16
17  SM_ADD_PARAM(size_t, automaton_size, 20)
18  SM_ADD_PARAM(size_t, x_size, 8)
19  SM_ADD_PARAM(size_t, z_count, 3)
20  SM_ADD_PARAM(bool, one_point_crossover, false)
21  SM_ADD_PARAM(bool, mutate_count, 0)
22  SM_ADD_PARAM(double, mutate_p, 0.15)
23
24
25  namespace automaton {
26  namespace moore {
27
28  full_table::full_table()
29  {
30      size_t sz = SM_PARAM(size_t, automaton_size);
31      size_t xs = SM_PARAM(size_t, x_size);
32      size_t zc = SM_PARAM(size_t, z_count);
33
34      for (size_t i = 0; i < sz; ++i)
35      {
36          a.push_back(std::pair<size_t, std::vector<size_t> >(std::make_pair(0, std::vector<↩
                size_t>(1 << xs))));
37
38          a[i].first = random(zc);
39
40          for (size_t j = 0; j < (1 << xs); ++j)
41          {
42              a[i].second[j] = random(sz);
43          }
44
45      }
46  }
47
48  boost::shared_ptr<individual> full_table::mutate() const
49  {
50      return boost::shared_ptr<full_table>(new full_table(this));
51  }
52
53  full_table::full_table(const full_table * o)
54  {
55      a = o->a;
56      size_t mc = SM_PARAM(size_t, mutate_count);
57      double mp = SM_PARAM(double, mutate_p);
58      size_t sz = SM_PARAM(size_t, automaton_size);
59      size_t xs = SM_PARAM(size_t, x_size);
60      size_t zc = SM_PARAM(size_t, z_count);
61
62      if (mc)
63      {
```

```
64          std::vector<bool> mut(sz, false);
65          size_t c = 0;
66
67          while (c < mc)
68          {
69              size_t mr = random(sz);
70
71              if (!mut[mr])
72              {
73                  mut[mr] = true;
74                  ++c;
75              }
76          }
77
78          for (size_t j = 0; j < sz; ++j)
79          {
80              if (mut[j])
81              {
82                  if (random(0.5))
83                  {
84                      a[j].first = random(zc);
85                  }
86                  else
87                  {
88                      for (size_t i = 0; i < (1 << xs); ++i)
89                      {
90                          if (random(mp))
91                          {
92                              a[j].second[i] = random(sz);
93                          }
94                      }
95                  }
96              }
97          }
98      }
99      else
100     {
101         for (size_t j = 0; j < sz; ++j)
102         {
103             if (random(mp))
104             {
105                 if (random(0.5))
106                 {
107                     a[j].first = random(zc);
108                 }
109                 else
110                 {
111                     for (size_t i = 0; i < (1 << xs); ++i)
112                     {
113                         if (random(0.5))
114                         {
115                             a[j].second[i] = random(sz);
116                         }
117                     }
118                 }
119             }
120         }
121     }
122 }
123
124 full_table::full_table(std::vector<std::pair<size_t, std::vector<size_t> > > const & a_) :
125     a(a_)
126 {
127 }
128
129 std::pair<boost::shared_ptr<individual>, boost::shared_ptr<individual> > full_table::corssover(↩
    boost::shared_ptr<individual> const & o) const
130 {
131     boost::shared_ptr<full_table> another = boost::dynamic_pointer_cast<full_table>(o);
132
133     std::vector<std::pair<size_t, std::vector<size_t> > > a1, a2;
134     size_t sz = SM_PARAM(size_t, automaton_size);
135
136     std::vector<size_t> an1(0), an2(0), rn1(sz), rn2(sz);
137
138     std::stack<size_t, std::vector<size_t> > st;
139     std::vector<bool> mark(sz, false);
140
141     for (size_t i = 0; i < sz; ++i)
```

```
142     {
143         if (mark[i])
144             continue;
145         st.push(i);
146         mark[i] = true;
147         while (!st.empty())
148         {
149             size_t u = st.top();
150             st.pop();
151             rn1[u] = an1.size();
152             an1.push_back(u);
153 
154             for (size_t i = 0; i < a[u].second.size(); ++i)
155             {
156                 size_t v = a[u].second[i];
157                 if (!mark[v])
158                 {
159                     st.push(v);
160                     mark[v] = true;
161                 }
162             }
163         }
164     }
165     mark.assign(sz, false);
166     for (size_t i = 0; i < sz; ++i)
167     {
168         if (mark[i])
169             continue;
170         st.push(i);
171         mark[i] = true;
172         while (!st.empty())
173         {
174             size_t u = st.top();
175             st.pop();
176             rn2[u] = an2.size();
177             an2.push_back(u);
178 
179             for (size_t i = 0; i < another->a[u].second.size(); ++i)
180             {
181                 size_t v = another->a[u].second[i];
182                 if (!mark[v])
183                 {
184                     st.push(v);
185                     mark[v] = true;
186                 }
187             }
188         }
189     }
190     bool opc = SM_PARAM(bool, one_point_crossover);
191     size_t xs = SM_PARAM(size_t, x_size);
192 
193     for (size_t i = 0; i < sz; ++i)
194     {
195         size_t fp = opc ? 0 : random(1 << xs);
196         size_t sp = random(1 << xs);
197 
198         std::vector<size_t> c1, c2;
199 
200         if (fp > sp)
201             std::swap(fp, sp);
202 
203         for (size_t j = 0; j < fp; ++j)
204         {
205             c1.push_back(rn1[a[an1[i]].second[j]]);
206             c2.push_back(rn2[another->a[an2[i]].second[j]]);
207         }
208 
209         for (size_t j = fp; j < sp; ++j)
210         {
211             c2.push_back(rn1[a[an1[i]].second[j]]);
212             c1.push_back(rn2[another->a[an2[i]].second[j]]);
213         }
214 
215         for (size_t j = sp; j < (1 << xs); ++j)
216         {
217             c1.push_back(rn1[a[an1[i]].second[j]]);
218             c2.push_back(rn2[another->a[an2[i]].second[j]]);
219         }
220 
```

```
221         size_t s1 = a[an1[i]].first, s2 = another->a[an2[i]].first;
222 
223         if (random(0.5))
224             std::swap(s1, s2);
225 
226         a1.push_back(std::make_pair(s1, c1));
227         a2.push_back(std::make_pair(s2, c2));
228     }
229 
230     return std::make_pair(boost::shared_ptr<individual>(new full_table(a1)),
231             boost::shared_ptr<individual>(new full_table(a2)));
232 }
233 
234 size_t full_table::get_states_count() const
235 {
236     return SM_PARAM(size_t, automaton_size);
237 }
238 
239 size_t full_table::input_event(input const & i, output & o, size_t state) const
240 {
241     size_t x = 0;
242     size_t xs = SM_PARAM(size_t, x_size);
243 
244     for (size_t j = 0; j < xs; ++j)
245     {
246         if (i.x(j))
247             x = x | (1 << j);
248     }
249 
250     size_t ns = a[state].second[x];
251     o.z(a[ns].first);
252 
253     return ns;
254 }
255 
256 void full_table::draw(const char * filename)
257 {/*
258   digraph G {
259     1 -> 2 [label="qq"]
260     1 -> 3
261     1 -> 1
262     2 -> 3
263     3 -> 2
264 }*/
265     std::fstream out(filename, std::ios::out);
266 
267     size_t sz = SM_PARAM(size_t, automaton_size);
268     size_t xs = SM_PARAM(size_t, x_size);
269     out << sz << " 0\n";
270     for (size_t i = 0; i < sz; ++i)
271     {
272         out << a[i].first << " ";
273     }
274 
275     out << "\n";
276 
277     for (size_t i = 0; i < sz; ++i)
278     {
279         for (size_t j = 0; j < (1 << xs); ++j)
280         {
281             out << i << " " << j << " " << a[i].second[j] << "\n";
282         }
283     }
284 
285     out << std::flush;
286     out.close();
287 }
288 
289 }
290 }
```

6.1.1.2. reduced_table

reduced_table.h

```
1 /*
```

```
 * reduced_table.h
 *
 *  Created on: Oct 14, 2011
 *      Author: Filchenko Nikolay
 */

#ifndef REDUCED_TABLE_H_
#define REDUCED_TABLE_H_

#include "automaton/representation.h"
#include <vector>

namespace automaton {

namespace moore {

class reduced_table : public representation
{
public:
    reduced_table();
    virtual boost::shared_ptr<individual> mutate() const;
    virtual std::pair<boost::shared_ptr<individual>, boost::shared_ptr<individual> > corssover(
        boost::shared_ptr<individual> const & o) const;
    virtual size_t get_states_count() const;
    virtual size_t input_event(input const & i, output & o, size_t state) const;
    virtual void draw(const char * filename);
private:
    reduced_table(const reduced_table * o);
    reduced_table(std::vector<std::pair<size_t, std::pair<std::vector<size_t>, size_t> > >
        const & a_);
    std::vector<std::pair<size_t, std::pair<std::vector<size_t>, size_t> > > a;
};

}

}

#endif /* REDUCED_TABLE_H_ */
```

reduced_table.cpp

```
/*
 * reduced_table.cpp
 *
 *  Created on: Oct 14, 2011
 *      Author: Filchenko Nikolay
 */

#include "automaton/moore/reduced_table.h"

#include "settings/settings_manager.h"
#include <iostream>
#include <fstream>
#include "global.h"

#include <stack>

SM_ADD_PARAM(size_t, rt_automaton_size, 20);
SM_ADD_PARAM(size_t, rt_x_size, 8);
SM_ADD_PARAM(size_t, rt_r_x_size, 3);
SM_ADD_PARAM(size_t, rt_z_count, 3);
SM_ADD_PARAM(bool, rt_one_point_crossover, true);
SM_ADD_PARAM(bool, rt_mutate_count, 0);
SM_ADD_PARAM(double, rt_mutate_p, 0.15);



namespace automaton {
namespace moore {

reduced_table::reduced_table()
{
    size_t sz = SM_PARAM(size_t, rt_automaton_size);
    size_t xs = SM_PARAM(size_t, rt_x_size);
    size_t rxs = SM_PARAM(size_t, rt_r_x_size);
    size_t zc = SM_PARAM(size_t, rt_z_count);

```

```
37      for (size_t i = 0; i < sz; ++i)
38      {
39          a.push_back(std::pair<size_t, std::pair<std::vector<size_t>, size_t > >(std::make_pair↩
                (0, std::make_pair(std::vector<size_t>(1 << xs), 0))));
40
41          a[i].first = random(zc);
42
43          a[i].second.second = 0;
44
45          for (size_t j = 0; j < rxs; ++j)
46          {
47              size_t p = 0;
48              while ( (1 << (p = random(xs))) & a[i].second.second );
49              a[i].second.second = a[i].second.second | (1 << p);
50
51          }
52
53          for (size_t j = 0; j < (1 << xs); ++j)
54          {
55              a[i].second.first[j] = random(sz);
56          }
57
58      }
59  }
60
61  boost::shared_ptr<individual> reduced_table::mutate() const
62  {
63      return boost::shared_ptr<reduced_table>(new reduced_table(this));
64  }
65
66  reduced_table::reduced_table(const reduced_table * o)
67  {
68      a = o->a;
69      size_t mc = SM_PARAM(size_t, rt_mutate_count);
70      double mp = SM_PARAM(double, rt_mutate_p);
71      size_t sz = SM_PARAM(size_t, rt_automaton_size);
72      size_t xs = SM_PARAM(size_t, rt_x_size);
73      size_t rxs = SM_PARAM(size_t, rt_r_x_size);
74      size_t zc = SM_PARAM(size_t, rt_z_count);
75
76      if (mc)
77      {
78          std::vector<bool> mut(sz, false);
79          size_t c = 0;
80
81          while (c < mc)
82          {
83              size_t mr = random(sz);
84
85              if (!mut[mr])
86              {
87                  mut[mr] = true;
88                  ++c;
89              }
90          }
91
92          for (size_t j = 0; j < sz; ++j)
93          {
94              if (random(0.5))
95              {
96                  size_t f = 0;
97                  while (a[j].second.second & (1 << (f = random(xs))));
98                  a[j].second.second = a[j].second.second | (1 << f);
99
100                 while ((a[j].second.second & (1 << (f = random(xs)))) == 0);
101                 a[j].second.second = a[j].second.second ^ (1 << f);
102             }
103             if (mut[j])
104             {
105                 if (random(0.5))
106                 {
107                     a[j].first = random(zc);
108                 }
109                 else
110                 {
111                     for (size_t i = 0; i < (1 << rxs); ++i)
112                     {
113                         if (random(mp))
114                         {
```

```
                            a[j].second.first[i] = random(sz);
                        }
                    }
                }
            }
        }
    }
    else
    {
        for (size_t j = 0; j < sz; ++j)
        {
            if (random(0.5))
            {
                size_t f = 0;
                while (a[j].second.second & (1 << (f = random(xs))));
                a[j].second.second = a[j].second.second | (1 << f);

                while ((a[j].second.second & (1 << (f = random(xs)))) == 0);
                a[j].second.second = a[j].second.second ^ (1 << f);
            }

            if (random(mp))
            {
                if (random(0.5))
                {
                    a[j].first = random(zc);
                }
                else
                {
                    for (size_t i = 0; i < (1 << rxs); ++i)
                    {
                        if (random(0.5))
                        {
                            a[j].second.first[i] = random(sz);
                        }
                    }
                }
            }
        }
    }
}

reduced_table::reduced_table(std::vector<std::pair<size_t, std::pair<std::vector<size_t>, ↩
    size_t> > > const & a_) :
    a(a_)
{
}

std::pair<boost::shared_ptr<individual>, boost::shared_ptr<individual> > reduced_table::↩
    corssover(boost::shared_ptr<individual> const & o) const
{
    boost::shared_ptr<reduced_table> another = boost::dynamic_pointer_cast<reduced_table>(o);

    std::vector<std::pair<size_t, std::pair<std::vector<size_t>, size_t> > > a1, a2;
    size_t sz = SM_PARAM(size_t, rt_automaton_size);

    std::vector<size_t> an1(0), an2(0), rn1(sz), rn2(sz);

    std::stack<size_t, std::vector<size_t> > st;
    std::vector<bool> mark(sz, false);

    for (size_t i = 0; i < sz; ++i)
    {
        if (mark[i])
            continue;
        st.push(i);
        mark[i] = true;
        while (!st.empty())
        {
            size_t u = st.top();
            st.pop();
            rn1[u] = an1.size();
            an1.push_back(u);

            for (size_t i = 0; i < a[u].second.first.size(); ++i)
            {
                size_t v = a[u].second.first[i];
                if (!mark[v])
                {
```

```
                    st.push(v);
                    mark[v] = true;
                }
            }
        }
    }
    mark.assign(sz, false);
    for (size_t i = 0; i < sz; ++i)
    {
        if (mark[i])
            continue;
        st.push(i);
        mark[i] = true;
        while (!st.empty())
        {
            size_t u = st.top();
            st.pop();
            rn2[u] = an2.size();
            an2.push_back(u);

            for (size_t i = 0; i < another->a[u].second.first.size(); ++i)
            {
                size_t v = another->a[u].second.first[i];
                if (!mark[v])
                {
                    st.push(v);
                    mark[v] = true;
                }
            }
        }
    }
    bool opc = SM_PARAM(bool, rt_one_point_crossover);
    size_t xs = SM_PARAM(size_t, rt_x_size);
    size_t rxs = SM_PARAM(size_t, rt_r_x_size);

    for (size_t i = 0; i < sz; ++i)
    {
        size_t fp = opc ? 0 : random(1 << rxs);
        size_t sp = random(1 << rxs);

        std::vector<size_t> c1, c2;

        if (fp > sp)
            std::swap(fp, sp);

        for (size_t j = 0; j < fp; ++j)
        {
            c1.push_back(rn1[a[an1[i]].second.first[j]]);
            c2.push_back(rn2[another->a[an2[i]].second.first[j]]);
        }

        for (size_t j = fp; j < sp; ++j)
        {
            c2.push_back(rn1[a[an1[i]].second.first[j]]);
            c1.push_back(rn2[another->a[an2[i]].second.first[j]]);
        }

        for (size_t j = sp; j < (1 << rxs); ++j)
        {
            c1.push_back(rn1[a[an1[i]].second.first[j]]);
            c2.push_back(rn2[another->a[an2[i]].second.first[j]]);
        }

        size_t s1 = a[an1[i]].first, s2 = another->a[an2[i]].first;

        if (random(0.5))
            std::swap(s1, s2);
        size_t mask1 = a[an1[i]].second.second;
        size_t mask2 = another->a[an2[i]].second.second;

        size_t cross = mask1 ^ mask2;

        mask1 = mask1 & (~cross);
        mask2 = mask1 & (~cross);

        while (cross != 0)
        {
            size_t mi;
            while ((cross & (1 << (mi = random(xs)))) == 0);
```

```
271             mask1 = mask1 | (1 << mi);
272             cross = cross ^ (1 << mi);
273             while ((cross & (1 << (mi = random(xs)))) == 0);
274             mask2 = mask2 | (1 << mi);
275             cross = cross ^ (1 << mi);
276         }
277
278         assert(s1 < SM_PARAM(size_t, rt_automaton_size));
279         assert(s2 < SM_PARAM(size_t, rt_automaton_size));
280
281         a1.push_back(std::make_pair(s1, std::make_pair(c1, mask1))); //FIXME
282         a2.push_back(std::make_pair(s2, std::make_pair(c2, mask2))); //FIXME
283     }
284
285     return std::make_pair(boost::shared_ptr<individual>(new reduced_table(a1)),
286             boost::shared_ptr<individual>(new reduced_table(a2)));
287 }
288
289 size_t reduced_table::get_states_count() const
290 {
291     return SM_PARAM(size_t, rt_automaton_size);
292 }
293
294 size_t reduced_table::input_event(input const & i, output & o, size_t state) const
295 {
296     size_t x = 0;
297     size_t xs = SM_PARAM(size_t, rt_x_size);
298     size_t rxs = SM_PARAM(size_t, rt_r_x_size);
299
300     size_t ep = 0;
301
302     for (size_t j = 0; j < xs; ++j)
303     {
304         if (a[state].second.second & (1 << j))
305         {
306             if (i.x(j))
307             {
308                 x = x | (1 << ep);
309             }
310
311             ++ep;
312         }
313     }
314
315     size_t ns = a[state].second.first[x];
316     o.z(a[ns].first);
317
318     return ns;
319 }
320
321 void reduced_table::draw(const char * filename)
322 {/*
323   digraph G {
324     1 -> 2 [label="qq"]
325     1 -> 3
326     1 -> 1
327     2 -> 3
328     3 -> 2
329 }*/
330     std::fstream out(filename, std::ios::out);
331     out << "digraph G {\n";
332     size_t sz = SM_PARAM(size_t, rt_automaton_size);
333     size_t xs = SM_PARAM(size_t, rt_r_x_size);
334
335     for (size_t i = 0; i < sz; ++i)
336     {
337         for (size_t j = 0; j < (1 << xs); ++j)
338         {
339             out << i << " -> " << a[i].second.first[j] << ";\n";
340         }
341     }
342
343     out << "}" << std::flush;
344     out.close();
345 }
346
347 }
348 }
```

6.2. problem

6.2.0.3. model

model.h

```
#ifndef _H_MODEL_
#define _H_MODEL_

#include "global.h"
#include "individual.h"

namespace problem {

class model {
public:
    virtual double fitness(boost::shared_ptr<individual> const & ind, bool print = false, const char * filename = 0) = 0;
    virtual void generation() = 0;
};

}

#endif
```

6.2.1. ant3

6.2.1.1. model

model.h

```
#ifndef ANT3_MODEL_H
#define ANT3_MODEL_H

#include "problem/model.h"
#include "individual.h"
#include "automaton/representation.h"
#include <set>
#include <vector>

namespace problem {
namespace ant3 {

class model : public problem::model {
public:
    model();
    virtual double fitness(boost::shared_ptr<individual> const & ind, bool print = false, const char * fname = 0);
    virtual void generation();
private:
    void reset();
    std::vector<std::pair<std::set<std::pair<size_t, size_t> >, size_t> > fields;
};

}
}

#endif //ANT3_MODEL_H
```

model.cpp

```
#include "problem/ant3/model.h"
#include "settings/settings_manager.h"

#include "automaton/input.h"
#include "automaton/output.h"
#include "automaton/representation.h"
#include <fstream>
#include <boost/lexical_cast.hpp>


```

```
SM_ADD_PARAM(bool, recreate_fields, false);
SM_ADD_PARAM(size_t, fields_count, 20);
SM_ADD_PARAM(size_t, steps, 200);
SM_ADD_PARAM(size_t, field_w, 32);
SM_ADD_PARAM(size_t, field_h, 32);
SM_ADD_PARAM(double, apple_p, 0.05);

namespace problem {
namespace ant3 {

model::model()
{
    reset();
}

class automaton_input : public automaton::input{
private:
    char inp;
public:
    automaton_input(char c) : inp(c) {};
    virtual bool x(size_t i) const
    {
        return inp & (1 << i);
    }
};

class automaton_output : public automaton::output{
public:
    mutable size_t act;
public:
    automaton_output() : act(0) {};
    virtual void z(size_t i) const
    {
        act = i;
    }
};

double model::fitness(boost::shared_ptr<individual> const & ind, bool print, const char * ↩
    filename)
{
    size_t fc = SM_PARAM(size_t, fields_count);
    size_t fw = SM_PARAM(size_t, field_w);
    size_t fh = SM_PARAM(size_t, field_h);
    size_t sc = SM_PARAM(size_t, steps);

    boost::shared_ptr<automaton::representation> aut = boost::dynamic_pointer_cast<automaton::↩
        representation>(ind);

    double fit = 0;

    std::fstream hist;

    if (print)
    {
        hist.open((std::string(filename) + "_fitness.log").c_str(), std::ios::out);
    }

    for (size_t f = 0; f < fc; ++f)
    {
        double cf = 0;
        size_t x = 0;
        size_t y = 0;
        size_t d = 0;
        size_t cstate = 0;
        std::set<std::pair<size_t, size_t> > cfield = fields[f].first;
        size_t ls = 0;
        if (print)
            hist << "field" << f << "\n";
        for (size_t step = 0; step < sc; ++step)
        {
            if (print)
                hist << x << " " << y << "\n";
            size_t xa[8], ya[8];
            switch (d) {
                case 0:
                    xa[0] = xa[1] = xa[5] = xa[7] = x;
                    ya[0] = y - 2;
                    ya[1] = y - 1;
                    ya[5] = y + 1;
```

```
                    ya[7] = y + 2;
                    xa[2] = xa[3] = xa[6] = x + 1;
                    ya[2] = y - 1;
                    ya[3] = y;
                    ya[6] = y + 1;
                    xa[4] = x + 2;
                    ya[4] = y;
                    break;
                case 1:
                    ya[0] = ya[1] = ya[5] = ya[7] = y;
                    xa[0] = x - 2;
                    xa[1] = x - 1;
                    xa[5] = x + 1;
                    xa[7] = x + 2;
                    ya[2] = ya[3] = ya[6] = y + 1;
                    xa[2] = x - 1;
                    xa[3] = x;
                    xa[6] = x + 1;
                    ya[4] = y + 2;
                    xa[4] = x;
                    break;
                case 2:
                    xa[0] = xa[1] = xa[5] = xa[7] = x;
                    ya[0] = y + 2;
                    ya[1] = y + 1;
                    ya[5] = y - 1;
                    ya[7] = y - 2;
                    xa[2] = xa[3] = xa[6] = x - 1;
                    ya[2] = y + 1;
                    ya[3] = y;
                    ya[6] = y - 1;
                    xa[4] = x - 2;
                    ya[4] = y;
                    break;
                case 3:
                    ya[0] = ya[1] = ya[5] = ya[7] = y;
                    xa[0] = x + 2;
                    xa[1] = x + 1;
                    xa[5] = x - 1;
                    xa[7] = x - 2;
                    ya[2] = ya[3] = ya[6] = y - 1;
                    xa[2] = x + 1;
                    xa[3] = x;
                    xa[6] = x - 1;
                    ya[4] = y - 2;
                    xa[4] = x;
                    break;
            };

            char inp = 0;

            for (size_t k = 0; k < 8; ++k)
            {
                xa[k] = (xa[k] + fw) % fw;
                ya[k] = (ya[k] + fh) % fh;
                if (cfield.find(std::make_pair(xa[k], ya[k])) != cfield.end())
                {
                    inp = inp | (1 << k);
                }
            }

            automaton_input in(inp);
            automaton_output out;
            cstate = aut->input_event(dynamic_cast<automaton::input&>(in),
                dynamic_cast<automaton::output&>(out),
                cstate);
            switch(out.act)
            {
                case 0:
                    switch (d)
                    {
                        case 0:
                            x++;
                            break;
                        case 1:
                            y++;
                            break;
                        case 2:
                            x--;
```

```
                            break;
                        case 3:
                            y--;
                            break;
                    };
                    break;
                case 1:
                    d = (d + 1) % 4;
                    break;
                case 2:
                    d = (d + 3) % 4;
                    break;
            };

            x = (x + fw) % fw;
            y = (y + fh) % fh;

            if (cfield.find(std::make_pair(x, y)) != cfield.end())
            {
                cf += 1;
                ++ls;
                cfield.erase(std::make_pair(x, y));
            }

        }
        cf *= 100;//fw * fh;
        cf /= fields[f].second;
        if (ls > 0)
        {
            cf += 1.0 / ls;
        }
        fit += cf;
    }

    if (print)
    {
        hist << std::flush;
        hist.close();
    }

    return fit / fc;
}

void model::generation()
{
    if (SM_PARAM(bool, recreate_fields))
        reset();
}

void model::reset()
{
    size_t fc = SM_PARAM(size_t, fields_count);
    size_t fw = SM_PARAM(size_t, field_w);
    size_t fh = SM_PARAM(size_t, field_h);

    fields.clear();
    double ap = SM_PARAM(double, apple_p);

    for (size_t f = 0; f < fc; ++f)
    {
        fields.push_back(std::make_pair(std::set<std::pair<size_t, size_t> >(), 0));
        while(fields.back().second == 0)
        {
            for (size_t i = 0; i < fw; ++i)
            {
                for (size_t j = 0; j < fh; ++j)
                {
                    if (random(ap))
                    {
                        fields.back().first.insert(std::make_pair(i, j));
                        ++fields.back().second;
                    }
                }
            }
        }

        std::fstream field(("field_" + boost::lexical_cast<std::string>(f)).c_str(), std::ios::↩
            out);
        field << fw << " " << fh << "\n";
```

```
245         for (size_t i = 0; i < fw; ++i)
246         {
247             for (size_t j = 0; j < fh; ++j)
248             {
249                 field << ((fields.back().first.find(std::make_pair(i, j)) != fields.back().↩
                        first.end()) ? "@" : ".");
250             }
251             field << "\n";
252         }
253         field << std::flush;
254         field.close();
255 
256     }
257 }
258 
259 }
260 }
```

6.3. settings

6.3.0.2. settings_manager

settings_manager.cpp

```
1  /*
2   * settings_manager.h
3   *
4   *  Created on: Oct 14, 2011
5   *      Author: Filchenko Nikolay
6   */
7 
8  #ifndef SETTINGS_MANAGER_H_
9  #define SETTINGS_MANAGER_H_
10 
11 #include <boost/noncopyable.hpp>
12 
13 #include <string>
14 #include <map>
15 #include <boost/lexical_cast.hpp>
16 
17 namespace settings {
18 
19 class settings_manager : private boost::noncopyable
20 {
21 public:
22     template<typename T>
23     T read_value(std::string const & name)
24     {
25         return boost::lexical_cast<T>(set[name]);
26     }
27 
28     template<typename T>
29     void add_value(std::string const & name, T const & def)
30     {
31         set[name] = boost::lexical_cast<std::string>(def);
32     }
33 
34     void override(std::string const & name, std::string const & value)
35     {
36         set[name] = value;
37     }
38 
39     std::string data()
40     {
41         std::string res;
42         for (std::map<std::string, std::string>::iterator it = set.begin(); it != set.end(); ++↩
                it)
43         {
44             res += it->first + " = " + it->second + "\n";
45         }
46 
47         return res;
48     }
49 
50     static settings_manager & instance()
```

```
51      {
52          static settings_manager sm;
53          return sm;
54      }
55
56  private:
57      std::map<std::string, std::string> set;
58  };
59
60  namespace detail {
61  template<typename T>
62  struct __param_declaration
63  {
64      __param_declaration(std::string const & name, T const & def)
65      {
66          settings_manager::instance().add_value<T>(name, def);
67      }
68  };
69  }
70
71  }
72
73  #define SM_ADD_PARAM(type, name, def) \
74      settings::detail::__param_declaration<type> sm_param_ ## name(#name, def);
75
76  #define SM_PARAM(type, name)\
77      (settings::settings_manager::instance().read_value<type>(#name))
78
79  /*
80  #define SM_ADD_PARAM(type, name, def) type name = def;
81  #define SM_PARAM(type, name) name
82  */
83  #endif /* SETTINGS_MANAGER_H_ */
```