Multi-Agent Systems (MAS)

Discrete-Event Simulation

Tatiana Makhalova

Clermont-Ferrand, France
2015 - 2016

Contents

Introduction

A cellular automata is a powerful tool for different kind of simulations. In this project we consider the SIS-like (susceptible-infectious-susceptible) model of epidemics, some details of the implementation and the results of simulations with respect to various parameters, such as the probability of infection and recovery, the density of inhabitants and viruses populations, lifetime of viruses and strategies of inhabitants movements.

In the first section we consider the concept of the model, types of agents, their properties and parameters. Details of the implementation are represented in the second section. In the third section we discuss results of model evolution and the simulations result depending on different parameters.

1 Model description

1.1 Concept

The simple SIS-like model of epidemics is described in the project. There are two type of objects (agents) here: *viruses* and *inhabitants*. Inhabitants move inside an rectangle area situated at a 2-dimensional torus. Each inhabitant moves randomly or by its own intentions within an area. If an inhabitant fall into a surrounding of a virus it begins to ache sick with a probability $P\,(infection)$ and will recover in the feature with probability $P\,(recovery)$. The probability of infection is defined by a virus (each virus can have its own value or common values for all instances of the *Virus* class), while the probability of recovery is be defined for inhabitant (the value may be different or the same for all instances of the *Inhabitant* class). Viruses can arise and vanish, inhabitants can only move and change their states.

It should be noted that proposed model is not the classical SIS model, since the network changes dynamically (due to moving of inhabitants) and the spreading of an epidemic goes from a virus to an inhabitant (whereas in the classical model it spreads from an infected inhabitant to susceptible ones). Thus, it is impossible to get an analytical solution and the equations, derived by Kermack and McKendrick, are not applicable in this case. We can get some statistic and investigate the properties of the model by simulations. Further, we will consider each class of objects, their properties (parameters) and actions.

1.2 Inhabitants

Inhabitants have its own or the same for all inhabitants probability of recovery (*recoveryRate* field). The number of inhabitants N is constant, they cannot die, but they can change their states (susceptible S or infected I - the property *Infected*), at each time t we have $S_t + I_t = N$. Inhabitants go from being susceptible to being infected and to being susceptible again (fig. 1).

We defined two types of movement: random and knowledge-based. The last type of movements takes the best cell within a given neighborhood (determined by *radius*). The best position corresponds to the cell with the smallest density of inhabitants and viruses (weights for density of inhabitants and viruses are used to manage the importance of parameters and therefore to change the strategy of movements). An inhabitant will retake the cell on an iteration, if the more healthy inhabitant wish to take the same cell (in other words, we define inhabitants priority for resolving conflicts during the relocation procedure).

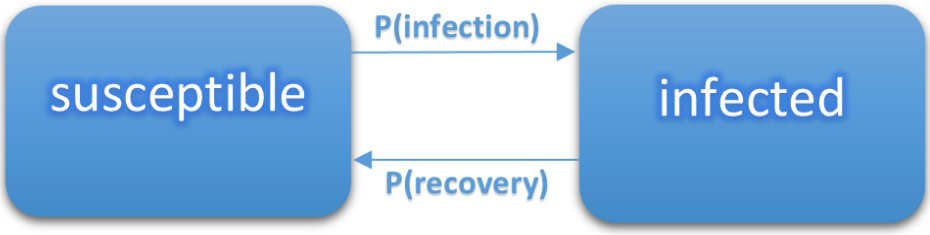


Figure 1: Inhabitants states

1.3 Viruses

Viruses arise by chance in a random position and have been living for a certain period of time (*lifetime* is the number of iterations during which an virus is presenting and spreading an infection). Each virus has its own radius (*radius*) of spreading, infection probability (*infectionRate* is the probability that an inhabitant becomes infected after relocation into an area of influence of a virus), on each iteration the random number of viruses arises (the maximal rate of new viruses is a parameter of simulations *virusesRate*). Viruses can spread an infection in the given neighborhood (fig. 2) with a given probability (the same for all objects or random for each virus).

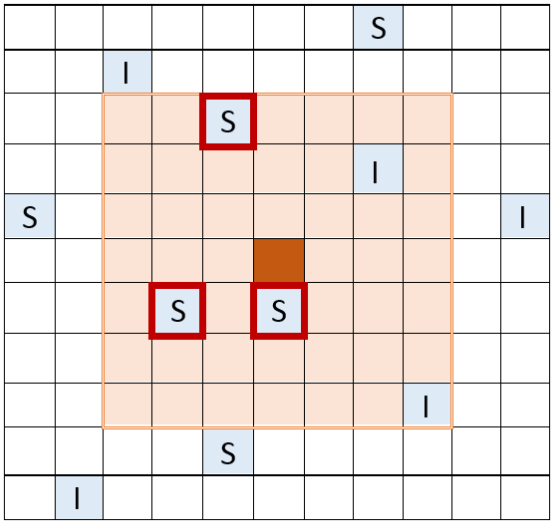


Figure 2: The spreading of an infection: a virus infects susceptible inhabitants in the neighborhood (cells are circled in red) with a certain probabilrty

1.4 An Expert

An expert was introduced for collecting statistic, generating of random numbers and giving some information (for example, a density of objects in a cell). Creating of viruses and inhabitants, supplying information to a view (GUI), collecting of statistic (including data that is given by deleted viruses) fall to an expert.

1.5 Objects: summary

Table 1: The description of viruses and inhabitants

Viruses (objects)	**Inhabitants (cognitive agents)**
Become activated on each iteration (activity - spreading disease)	Analyze the environment (how much viruses and inhabitants live in the given area)
Don't organize: appear in a random cell	Select a cell with the most favorable conditions
Don't communicate with each other	Social intelligence: enter into negotiations in the case of a conflict, an inhabitant may give place to more healthy one
	Don't map out long-term strategy, since the environment changes on each iteration

2 Implementation

2.1 Software Engineering Tools

Visual Studio 2015 is used as a programming environment, a programming language is C#. To make the analysis of simulations result the following libraries have been used: *pandas*, *numpy* and *matplotlib* (Python).

2.1.1 Profiling tools

For the performance analysis Microsoft Visual Studio Profiling Tools was choosen. The table 2 represents available tools for scrutiny of the program. Next, we briefly look at some of them.

Table 2: The profiling methods are available in *VS 2015*

The method	Description
CPU Sampling	Collects a call stack of the functions that are executing on the processor, memory allocation information is also available in this mode
Instrumentation	Collects detailed timing information about each function call including memory allocation
.NET memory allocation	Tracks a lifetime of objects (starts on resource allocation, ends after garbage collecting)
Concurrency	It is used for multi-threading applications

CPU sampling (fig.3) provides information on functions which do the most of work. To analyze the

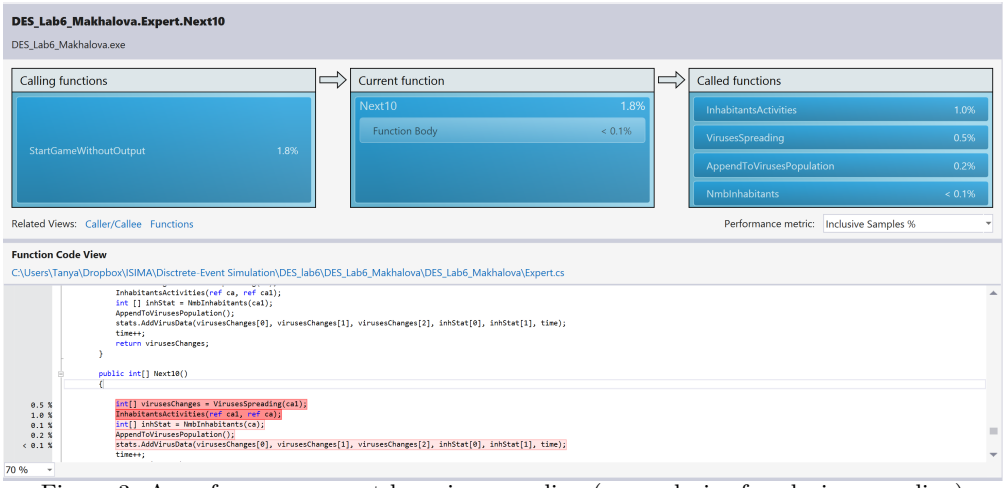


Figure 3: A performance report by using sampling (an analysis of exclusive sampling)

weakness of "lite" functions the inclusive analysis was performed. The comparative analysis the execution time of the target function and child function was made.

Instrumentation (fig.4) provides tools for understanding the impact of input and output operations on application performance, since it gathers detailed timing information about a section of analyzed code. Inclusive sampling provides statistics of the target function and other functions that are called by the target function. Is is used to analyze and optimize such function as spreading of viruses, moving inhabitants and the "heaviest" sub-function of the target one.

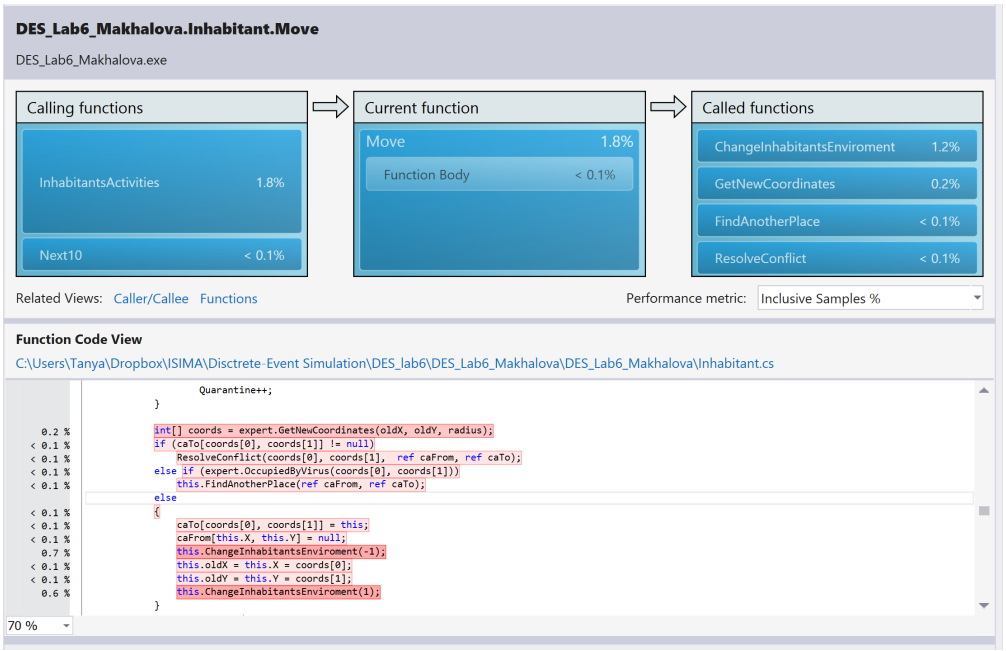


Figure 4: Performance report by using instrumentation

2.1.2 UML tools

The UML diagrams were constructed with Visual Studio 2015 (manual mode). Standard tools of VS allow to create several types of diagrams (fig.5), a diagram of the process of cognitive inhabitant movements will be represented further (fig.9).

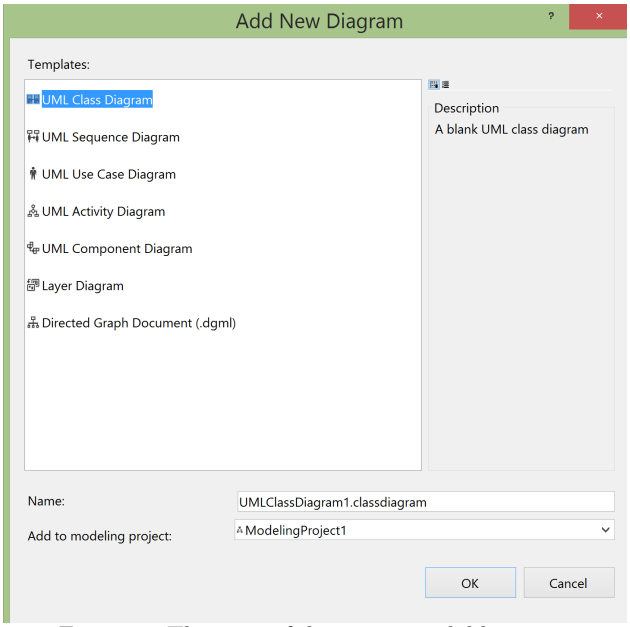


Figure 5: The type of diagrams available to use

2.2 Implementation of classes of agents

The structure of the class diagram is represented on fig.6. Viruses and inhabitants inherit from the same class *MASObject* that contains the following properties: *coordinates of object*, *radius of spreading*, *id*.

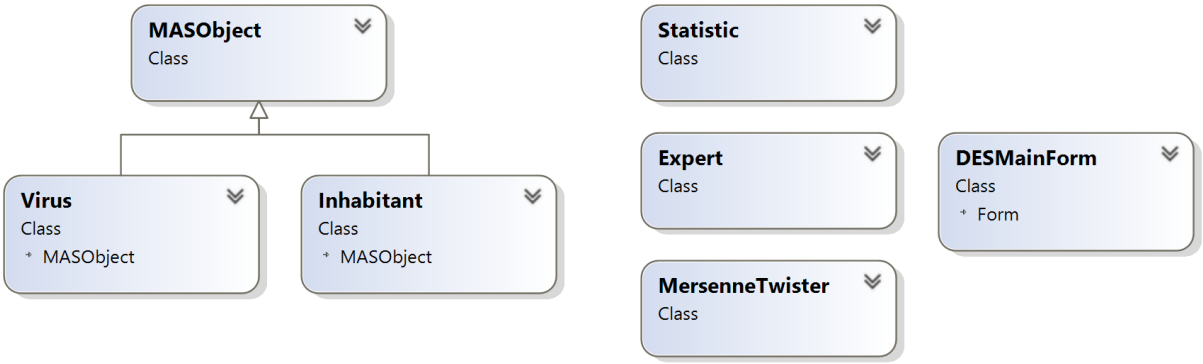


Figure 6: The class diagram of the project

2.2.1 Class Virus

Viruses are created on each iteration by an *Expert*. During a lifetime each virus is collecting the amount of infected inhabitants on each iteration and sends this information to an *Expert* on its (virus's) disappearance. For a virus only one action is defined: spreading of an infection. Since we use a torus-like space we have a special spreading procedure on the borders and in the corners (fig.7, a - the spreading are is divided on several parts), the spreading inside an rectangle is identical (fig.7, b).

```
/*
 * Spreads the disease in the area, that defined by the radius.
 * In the case of boundary positions the disease is spreading on a space torus like folding
 * */
1 reference
public void Diffusion(Inhabitant [,] ca)
{
    if (lifetime == 0) //virus is dying
    {
        ChangeVirusesEnvirement(-1); //change enviroment information
        expert.stats.AddVirusData(this); //append data to a statistics collector
    }
    else
    {
        lifetime--; // decrease the remaining lifetime

        int[] coords = expert.GetBorders(X, Y, radius);
        int strategy = (coords[2] > coords[3] ? 1 : 0) + (coords[0] > coords[1] ? 2 : 0); //select the borders of spreading the decease
                                                                                          // for torus like folding
        switch (strategy)
        {
            case (0): // a virus is located inside the area
                DiffusionInsideArea(coords[0], coords[1], coords[2], coords[3], ca);
                break;
            case (1): //a virus is located beside a vertical border
                DiffusionInsideArea(coords[0], coords[1], 0, coords[3], ca);
                DiffusionInsideArea(coords[0], coords[1], coords[2], expert.Width - 1, ca);
                break;
            case (2):  //a virus is located beside a horisontal border
                DiffusionInsideArea(0, coords[1], coords[2], coords[3], ca);
                DiffusionInsideArea(coords[0], expert.Height - 1, coords[2], coords[3], ca);
                break;
            case (3): //a virus is located beside a corner
                DiffusionInsideArea(0, coords[1], 0, coords[3], ca);
                DiffusionInsideArea(coords[0], expert.Height - 1, 0, coords[3], ca);
                DiffusionInsideArea(0, coords[1], coords[2], expert.Width - 1 - 1, ca);
                DiffusionInsideArea(coords[0], expert.Height - 1, coords[2], expert.Width - 1, ca);
                break;
        }
    }
}
```

(a) The main spreading function

```
/**
 * Spreads the disease in the given area. Noninfected instanses of Inhabitant class become infected with a given probability of infection.
 * The corresponding duration of the disease is assigned.
 * \param[in] coordinates of the rectangular area
 * \param[in] the current area of CA
 */
9 references
void DiffusionInsideArea(int x0, int x1, int y0, int y1, Inhabitant [,] ca)
{
    float p = 0;
    for (int i = x0; i <= x1; i++)
        for (int j = y0; j < y1; j++)
            if (ca[i, j] != null)
                if (!ca[i, j].Infected)
                {
                    p = (float)expert.mt.genrand_real3();
                    if (p < infectionRate)
                    {
                        ca[i, j].Infected = true;
                        ca[i, j].IncreaseNmbOfInfections();
                        CurrentInfected++;
                    }
                }
    totalInfected += CurrentInfected;
}
```

(b) The function of spreading inside a rectangle area

Figure 7: The function of spreading of the infection

2.2.2 Class Inhabitant

An Expert creates the population of inhabitants once, during a simulation process inhabitants can change their state (susceptible and infected) and move within a space. For inhabitants two types of moving are defined: random (fig.8,a) and intelligent (based on knowledge about an environment and social intentions, fig.8, b).

The knowledge-based movements function For intelligent movements of inhabitants (an UML diagram is represented on fig.9) the following options were determined:

- Movement to the best location
- Giving up the place (which was taken on the current iteration) to the more healthy inhabitant in the case of a conflict

```
/**
 * The agent moves to a random position inside the neighborhood and
 * the random recovering process is simulated for infected inhabitant.
 * If some conflicts happen, another functions for resolving that collisions will be called.
 * \param [in] the departure area of CA
 * \param [in] the destination area of CA
 * \param [out] the boolean value - whether the agent recovered during this iteration
 * */
1 reference
public bool Move(ref Inhabitant[,] caFrom, ref  Inhabitant[,] caTo)
{
    bool recovered = false;
    if (Infected) //an recovery process
    {
        double rndRecovery = expert.mt.genrand_RealInInterval(0, 1);
        if (rndRecovery < recoveryRate)
        {
            Infected = false;
            recovered = true;
        }
        else
            Quarantine++; //collect the statistic data
    }

    int[] coords = expert.GetNewCoordinates(oldX, oldY, radius);
    if (caTo[coords[0], coords[1]] != null) //resolve conflict, if the cell was occuped by another inhabitant on this iteration
        ResolveConflict(coords[0], coords[1],  ref caFrom, ref caTo);
    else if (expert.OccupiedByVirus(coords[0], coords[1])) //remove an inhabitant to another cell
        FindAnotherPlace(ref caFrom, ref caTo);
    else //change the current position for the selected successfully coordinates
    {
        caTo[coords[0], coords[1]] = this;
        caFrom[X, Y] = null;
        ChangeInhabitantsEnviroment(-1);
        oldX = X = coords[0];
        oldY = Y = coords[1];
        ChangeInhabitantsEnviroment(1);
    }
    return recovered;
}
```

(a) Random

```
/**
 * The agent moves to the best position inside the neighborhood and
 * the random recovering process is simulated for infected inhabitant.
 * If some conflicts happen, another functions for resolving that collisions will be called.
 * \param [in] the departure area of CA
 * \param [in] the destination area of CA
 * \param [out] the boolean value - whether the agent recovered during this iteration
 * */
1 reference
public bool CognitiveMove(ref Inhabitant[,] caFrom, ref  Inhabitant[,] caTo)
{
    bool recovered = false;
    if (Infected)
    {
        double rndRecovery = expert.mt.genrand_RealInInterval(0, 1);
        if (rndRecovery < recoveryRate)
        {
            Infected = false;
            recovered = true;
        }
        else
            Quarantine++;
    }

    int[] coords = GetBestCellInNeiborhoood(oldX, oldY, radius, caTo); //get the best position in the given neighborhood

    if (caTo[coords[0], coords[1]] != null)
        ResolveConflict(coords[0], coords[1], ref caFrom, ref caTo);
    else
    {
        caTo[coords[0], coords[1]] = this;
        caFrom[X, Y] = null;
        ChangeInhabitantsEnviroment(-1);
        oldX = X = coords[0];
        oldY = Y = coords[1];
        ChangeInhabitantsEnviroment(1);
    }
    return recovered;
}
```

(b) Intelligent

Figure 8: The relocation functions of an inhabitant inside a neighborhood of the radius *radius*

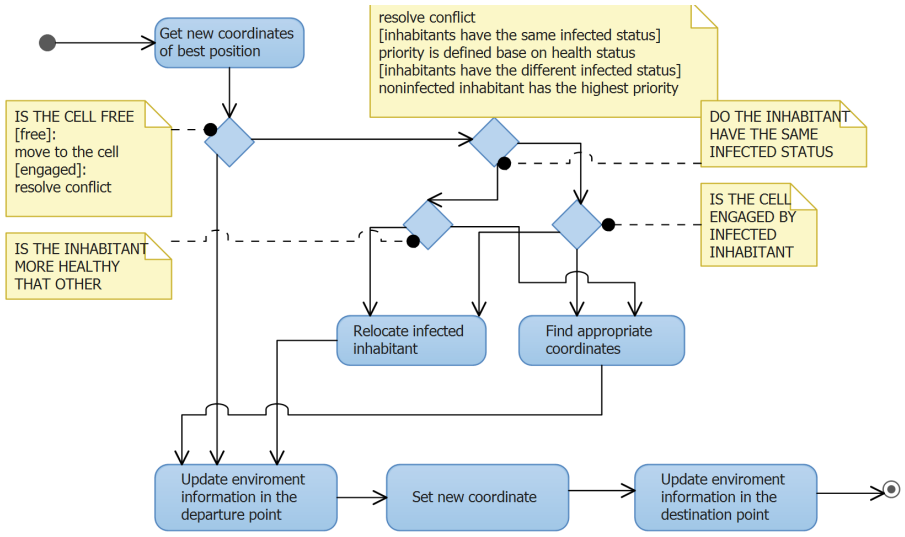


Figure 9: UML - Activity diagram of inhabitants movements

Movement to the best location in the neighborhood is a knowledge-based function. It is based on density of inhabitants and density of viruses. The function *CognitiveMove* (fig.8, b), the child function *GetBestCell* (fig.10) and *ResolveConflict* (fig. 11) make up the knowledge-based movement.

Dislocation Since the density of objects can be high, the expansion of neighborhood was provided in the case of a multitude of unsuccessful attempts of relocation.

2.3 GUI

GUI (fig.13) makes provision for the variation the following parameters:

1. The output space;
2. The size of a cellular automata: *Width* and *Height*;
3. The duration of a simulation: *Time*;

```
/**
 Returns coordinates of the most favourable for living cell in the rectungle area
    \param[in] x0, x1, y0, y1 coordinates of the rectangular area
    \param[in] weightV, weightI - cooficients of the objective function (for estimating cells)
    \param[in] ca area where cells are disposed
*/

9 references
float[] GetBestCell(int x0, int x1, int y0, int y1, float weightV, float weightI, Inhabitant [,] ca)
{
    float bestVal = Int16.MinValue, val = Int16.MinValue;
    int xBest = -1, yBest = -1;
    for (int i = x0; i <= x1; i++)
        for (int j = y0; j <= y1; j++)
        {
            val = - weightI * expert.GetInhabitantDensity(i, j) - weightV * expert.GetVirusesDensity(i, j);
            if ((val > bestVal) && (!expert.OccupiedByVirus(i, j)))
            {
                xBest = i;
                yBest = j;
                bestVal = val;
            }
        }
    return new float[3] { bestVal, xBest, yBest };
}
```

Figure 10: The function for selecting the best position in the neighborhood based on density of inhabitants (*GetInhabitantsDensity*) and density of viruses (*GetVirusesDensity*). Weights *weightI* and *weightV* are used to determine the importance of densities

```
/**
 * Resolves conflict, when two agent try to take the same cell.
 * The agent priority is defined by status of agent (noninfected one has the highest priority),
 * in the case of similar status the highest priority has agent with the best health (healthStatus value).
 * \param [in] coordimates of the cell
 * \param [in] the departure area of CA
 * \param [in] the destination area of CA
 */
2 references
void ResolveConflict(int x1, int y1,  ref Inhabitant [,] caFrom, ref Inhabitant [,] caTo)
{
    if (caTo[x1, y1].Infected != Infected)
    {
        nmbOfConflictsINI++;
        if (caTo[x1, y1].Infected)
        {
            caTo[x1, y1].FindAnotherPlace(ref caTo, ref caTo);
            caTo[x1, y1] = this;
            caFrom[X, Y] = null;
            ChangeInhabitantsEnviroment(-1);
            oldX = X = x1;
            oldY = Y = y1;
            ChangeInhabitantsEnviroment(1);
        }
        else
            FindAnotherPlace(ref caFrom, ref caTo);
    }
    else
    {
        if (caTo[x1, y1].Infected)
            nmbOfConflictsII++;
        else
            nmbOfConflictsININ++;
        if (recoveryRate > caTo[x1, y1].recoveryRate)
        {
            caTo[x1, y1].FindAnotherPlace(ref caTo, ref caTo);
            caTo[x1, y1] = this;
            caFrom[X, Y] = null;
            ChangeInhabitantsEnviroment(-1);
            oldX = X = x1;
            oldY = Y = y1;
            ChangeInhabitantsEnviroment(1);
        }
        else
            FindAnotherPlace(ref caFrom, ref caTo);
    }
}
```

Figure 11: The function for resolving conflicts

4. Checkbox *Cognitive moving* defines the type of inhabitants movements;
5. Checkbox *Intermediate output* is used for tracking states of a cellular automata during a simulation;
6. *The rate of viruses* (defines the maximal share of cells, which will be taken by new viruses on each iteration);
7. The probability of becoming infected after falling into a neighborhood of a virus: *The infection probability* or random for each virus - checkbox *Random infection rate for each virus*;
8. A virus lifetime (in iterations);
9. *The rate of inhabitants* (the share of cellular automata's cells);

```
/**
 * Looks for another position in the radius of movements
 * If this neighborhood is overcrowded (mor that 2*radius attempts to take a cell), the radius is increased.
 */
5 references
void FindAnotherPlace(ref Inhabitant[,] caFrom, ref Inhabitant[,] caTo)
{
    bool notRemoved = true;
    int curRadius = radius;
    int attempt = 0;
    do
    {
        notRemoved = Dislocate(curRadius, ref caFrom, ref caTo);
        if (attempt > 2 * curRadius)
        {
            attempt = 0;
            curRadius+=radius;
            if ((curRadius > caTo.GetLength(0) / 2) || (curRadius > caTo.GetLength(1) / 2))
                curRadius = Math.Min(caTo.GetLength(0), caTo.GetLength(1));
        }
        attempt++;
    } while (notRemoved);
}
```

Figure 12: The function of change a position

10. The probability of becoming recovered on an iteration (being infected): *The recovery probability* or random for each inhabitant - checkbox *Random recovery rate for each virus*;

11. The button *Start* is used to start simulation;

12. The button *Clear the output box* is used for clearing the output space.

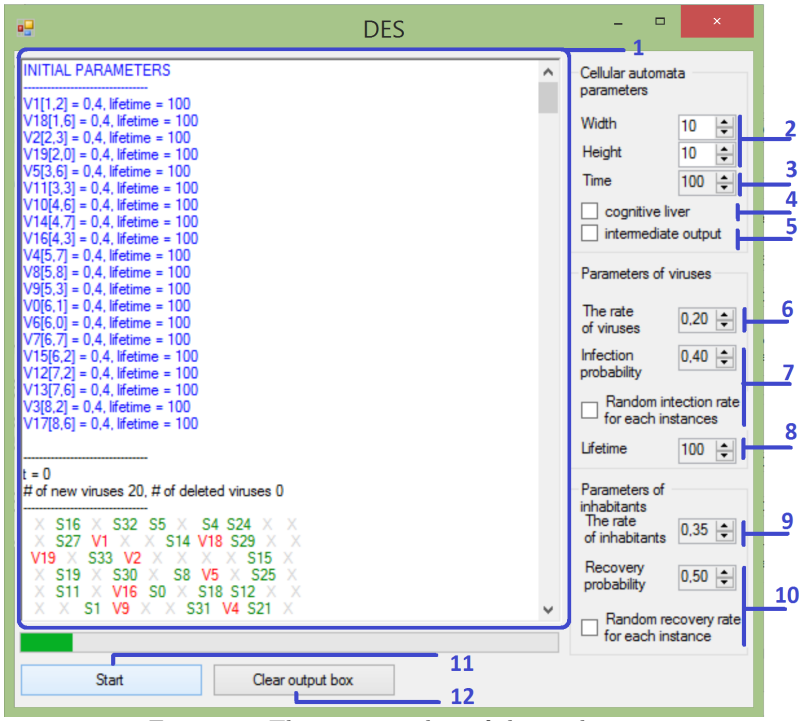


Figure 13: The main window of the application

2.4 Statistics

We collect raw data, which is processed by Python data analysis tools. For inhabitants the following information is collected:

- Duration of disease
- The number of relocation for infected/susceptible inhabitants

- The number of conflicts between infected / infected, susceptible / susceptible and infected / susceptible inhabitants
- The number of viruses (total / average / on each iteration)
- The amount infected and sustained inhabitants (on average / on each iteration / by each virus)

3 Analysis of results

The 10000 simulations for each set of parameters were used to investigate properties of the model. Futher, the following questions will be considered:

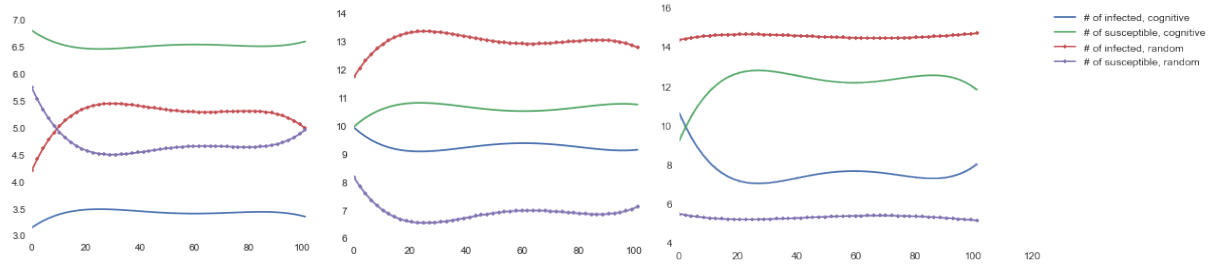
- The model evolution depending on various parameters
- The rate of infected and susceptible inhabitants w.r.t. different parameters of the model
- Time of simulations for two types of relocation strategies
- The number conflicts
- The average duration of the disease for an inhabitant

3.0.1 Ways of the model evolution

There are several ways of the evolution:

- The number of infected and susceptible inhabitants remain the same during the long period of time (a stable state from the beginning)
- The proportion of infected and susceptible inhabitants is unstable and changes significantly from iteration to iteration (an unstable state)
- The number of inhabitants of a certain type decreases/increases for the beginning and becomes stable after several iteration (the saturation process)

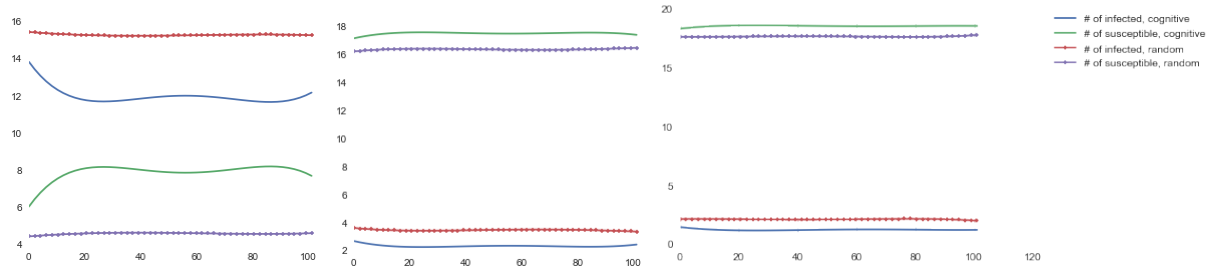
We can see from the fig.14 that the type of inhabitant movement has significant influence on the simulation results. Having the same probability of infection and recovery, but a small variation of density, the model evolution is mostly determined by the way of inhabitants moving. The relatively stable state of an epidemic with the random movements (c) opposes the state of halting of an epidemic during the simulation. The relatively stable "no epidemic" state with cognitive moving (a) opposes the state of halting of epidemic with random moving.



(a) inhabitants rate = 0.1, viruses rate = 0.05 (b) inhabitants rate = 0.2, viruses rate = 0.1 (c) inhabitants rate = 0.2, viruses rate = 0.2

Figure 14: The averaged number of infected and susceptible inhabitants during 100 iteration, probability of recovery = 0.2, probability of infection = 0.2

The evolution of the model is also changed, when the rates of viruses and inhabitants remain the same, but values of infection and recovery probabilities are changed. In this case the type of movement affects the results in the same manner (fig.15).



(a) probability of recovery = 0.2, probability of infection = 0.8 (b) probability of recovery = 0.8, probability of infection = 0.8 (c) probability of recovery = 0.8, probability of infection = 0.2

Figure 15: The averaged number of infected and susceptible inhabitants during 100 iteration, inhabitants rate = 0.2, viruses rate = 0.1, the number of viruses is constant and they are nonmovable

As the rate of occupied cells increases, the number of relocation (conflicts) increases. It happens due to strive of inhabitants to occupy the best location. As the rate of inhabitants and viruses increases, the share of the same best position for several inhabitants increases and hence, the number of conflicts for the same cell (in the case of knowledge-based movements) rises faster, then for random moving strategy.

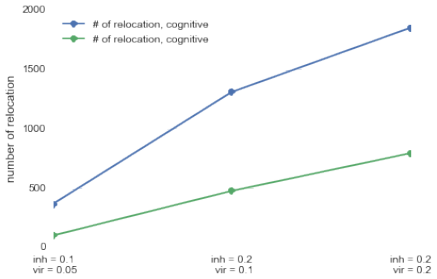


Figure 16: The number of relocations (conflicts) for different strategies of the cell selection

The time of simulations increases as well (as the number of relocations increases). The average time of simulation for random and intellegent cell selection is represented on the fig. 17.

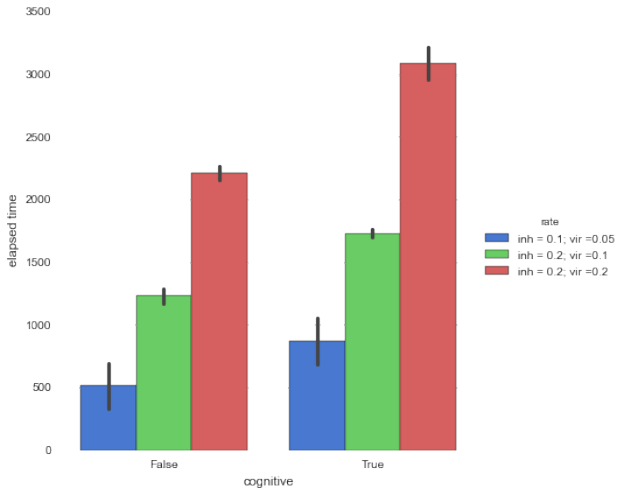
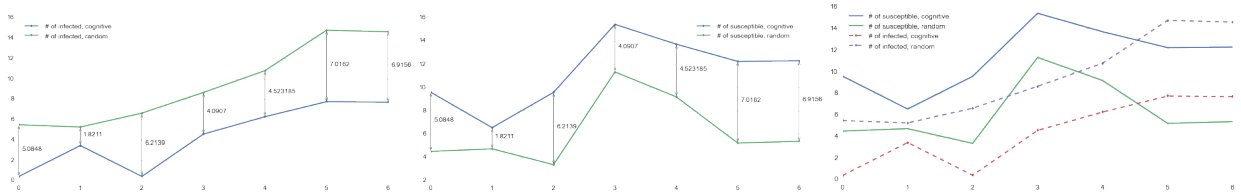


Figure 17: The average time of simulation for different density and two strategies of the cell selection

Since we consider SIS models of epidemic, each inhabitant can be infected and recovers several time.

Long-distance relocation effect As it was mentioned early, an inhabitant that cannot find good cell in its neighborhood removes to out of the neighborhood. As the rate of inhabitants increases such overcrowding



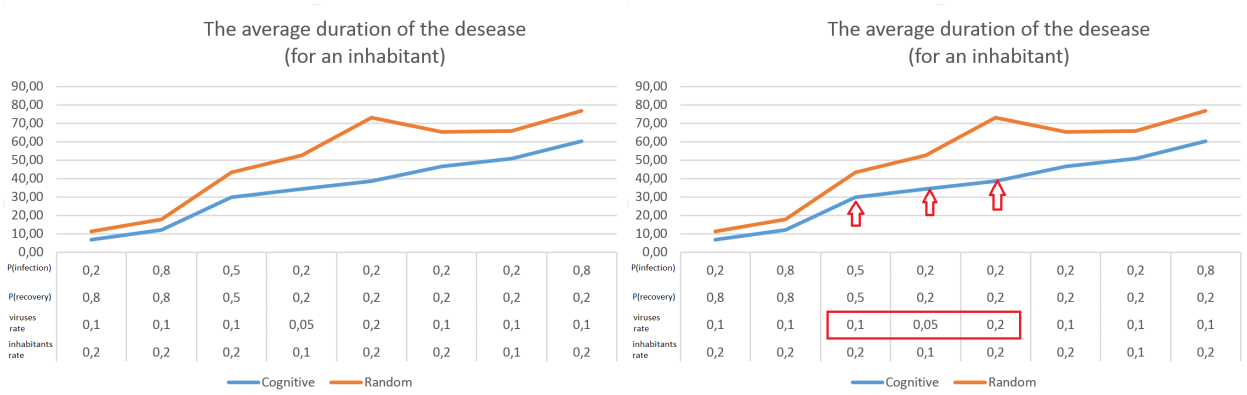
(a) The disparity in infected inhabitants (b) The disparity in susceptible inhabitants (c) The summary disparity in inhabitants states

Figure 18: The difference of the average number of infected and susceptible inhabitants (averaged among 10000 simulation) with respect to different parameters (table 3)

Table 3: The parameters of simulations represented on fig. 18, the rows are correspond to the points on x-axis

Number of a point	inhabitants rate	viruses rate	viruses lifetime	knowledge-based mean	random mean
0	0.10	0.05	1	0.40	5.49
1	0.10	0.05	101	3.44	5.27
2	0.10	0.10	1	0.42	6.63
3	0.20	0.10	1	4.57	8.66
4	0.20	0.10	101	6.28	10.80
5	0.20	0.20	1	7.76	14.77
6	0.20	0.20	101	7.70	14.62

on condition nonmovable viruses may lead to decreasing the average duration of disease of an inhabitant due to relocation takes a place beyond the virus's scope. It means that there is some point of the population density, such that exceeding this value leads to decreasing the duration of disease (the number of infections) because of inhabitants relocations into the area free of viruses (fig.19).



(a) (b) The effect of long-distance movements

Figure 19: The average duration of the disease for an inhabitant

4 Conclusion

In the report the SIS-like model has been considered. Unlike the classical SIS model, accordingly to the introduced model the spreading of a disease is permitted from a virus to an inhabitant, while in the original model a disease spreads from an inhabitant to another one. More than that, our model allows dynamic modification a neighborhood of agents (that is equivalent to changing the graph structure - edges between "an inhabitant node" and "a virus node" in the case of graph representation of agents).

In the report we presented the concepts of agent (inhabitants and viruses) and described the implementation details.

Performed series of simulations with the introduced model were analyzed and summarized in this report:

the evolution of the model, the rate of susceptible and infected inhabitants, the number of conflicts and relocations with respect to different parameters were shown. Some characteristic properties were discovered and represented in the last chapter of the report.

A Python code (fragments)

Polynomial fit

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

def get_approx(vals):
    y = vals.astype(float)
    x = np.array(range(y.shape[0]))
    return np.poly1d(np.polyfit(x, y, 5))
```

Plot of aggregated parameters of the model (the number of infected and susceptible inhabitants) during simulations

```
for key in data_stamp.keys():
    print "key = " ,key
    print different_parameters[key:key+1]
    criterion = logs.Stamp.map(lambda x: x in data_stamp[key])
    df_logs_same_parameters = logs[criterion]
    agg_df = df_logs_same_parameters.groupby(by = "DataType").mean().
        reset_index()
    val = agg_df[agg_df.DataType == "cntInfected"].values.T[2:].T+  agg_df
        [agg_df.DataType == "cntNonInfected"].values.T[2:].T
    plt.plot(agg_df[agg_df.DataType == "cntCurInfected"].T[2:], label = "
        cntCurInfected")
    plt.plot(agg_df[agg_df.DataType == "cntInfected"].T[2:], label = "
        cntInfected")
    plt.plot(agg_df[agg_df.DataType == "cntNonInfected"].T[2:], label = "
        cntNonInfected")
    plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
    plt.show()
```

Average simulation time for different strategies of selecting new cells and densities of viruses and inhabitants (fig.17).

```
agg_df["rate"] = pd.Series(data = l, index=agg_df.index)
g = sns.factorplot(x="cognitive", y=" elapsed time", hue="rate", data=
    agg_df, size=6, kind="bar", palette="muted")
g.despine(left=True)
```

Comparative graphs of inhabitants states with different parameters (corresponds to fig.18)

```
val_cog_inf = list(res[res.DataType == "cntInfected"].nmb_cog)
val_not_cog_inf = list(res[res.DataType == "cntInfected"].nmb_noncog)

val_cog_nonInf = list(res[res.DataType == "cntNonInfected"].nmb_cog)
val_not_cog_nonInf = list(res[res.DataType == "cntNonInfected"].nmb_noncog
    )

fig = plt.figure(1, figsize=(10, 15))
```

```
ax = fig.add_subplot(311, autoscale_on=True)

x_coords = range(len(val_not_cog_inf))
ax.plot(val_cog_inf,'.-', label = "␣#␣of␣infected,␣cognitive")
ax.plot(val_not_cog_inf, '.-', label = "␣#␣of␣infected,␣random")
ax.legend(loc = 0)
for i in xrange(len(val_not_cog_inf)):
    dif = abs(val_cog_inf[i] - val_not_cog_inf[i])
    plt.annotate('', xy=(i, val_cog_inf[i]), xycoords='data',xytext=(i,
       val_not_cog_inf[i]), textcoords='data',
                arrowprops={'arrowstyle': '<->'})
    plt.annotate(
        dif, xy=(i, max(val_cog_inf[i], val_not_cog_inf[i]) - dif/2),
           xycoords='data',
        xytext=(5, 0), textcoords='offset points')


fig = plt.figure(2, figsize=(10, 15))
ax = fig.add_subplot(312, autoscale_on=True)

x_coords = range(len(val_cog_nonInf))
ax.plot(val_cog_nonInf,'.-', label = "␣#␣of␣susceptible,␣cognitive")
ax.plot(val_not_cog_nonInf, '.-',label = "␣#␣of␣susceptible,␣random")
ax.legend(loc = 0)
for i in xrange(len(val_cog_nonInf)):
    dif = abs(val_cog_nonInf[i] - val_not_cog_nonInf[i])
    plt.annotate('', xy=(i, val_cog_nonInf[i]), xycoords='data',xytext=(i,
        val_not_cog_nonInf[i]), textcoords='data',
                arrowprops={'arrowstyle': '<->'})
    plt.annotate(dif, xy=(i, max(val_cog_nonInf[i], val_not_cog_nonInf[i])
        - dif/2), xycoords='data',
    xytext=(5, 0), textcoords='offset points')

fig = plt.figure(3, figsize=(10, 15))
ax = fig.add_subplot(313, autoscale_on=True)
ax.plot(val_cog_nonInf, label = "#␣of␣susceptible,␣cognitive")
ax.plot(val_not_cog_nonInf, label = "#␣of␣susceptible,␣random")
ax.plot(val_cog_inf,'.--',label = "#␣of␣infected,␣cognitive")
ax.plot(val_not_cog_inf, '.--', label = "#␣of␣infected,␣random")
ax.legend(loc = 0)
```

B Source code (fragments)

B.1 Class "Virus"

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

namespace DES_Lab6
{
```

```
/**
 * Viruses are stationary objects, spreading an infection
 * among instances of the Inhabitant class
 *
 * Instances of this class may differ the radius of spreading,
 * the probability of the infection by a contact with noninfected
    inhabitants, duration of life.
 * The parent class is MASObject
 */
class Virus : MASObject
{
    /// <summary>
    /// Probability of the infection
    /// </summary>
    float infectionRate;
    /// <summary>
    /// The number of iteratings during the virus is existing
    /// </summary>
    int lifetime = 0;
    /// <summary>
    /// The total number of infected agents by the virus
    /// </summary>
    int totalInfected = 0;
    /// <summary>
    /// The number of infected agents by the virus on current
       iteration (statistic variable)
    /// </summary>
    public int CurrentInfected { get; set; }

    public float Lifetime
    {
        get { return lifetime; }
    }

    public float Contagiousness
    {
        get { return infectionRate; }
    }

    /**
     * Creates the instance of the Virus class
     * in the random position at unique place for each instance
     * \param[in] expert an instance of the Expert class
     * \param[in] id the unique number of a new-generated virus
     */
    public Virus(Expert expert, int id)
        : base(expert, id)
    {
        this.expert = expert;
        lifetime = expert.VirusesLifetime;
        infectionRate = expert.InfectionRate;
        if (infectionRate == -1)
            infectionRate = base.GetRealNumber(0, (float)0.6);
        bool collision = false;
```

```
        int[] coords;
        do
        {
            coords = expert.GetSomeCoordinates();
            if (this.expert.ca[coords[0], coords[1]] == null)
                collision = expert.DetectVirusCollision(coords[0],
                    coords[1], ref expert.ca);
        } while (collision);
        X = coords[0];
        Y = coords[1];
        ChangeVirusesEnvirement(1);
    }

    /**
     * Spreads the disease in the given area. Noninfected instanses of
         Inhabitant class become infected with a given probability of
        infection.
     * The corresponding duration of the disease is assigned.
     * \param[in] x0,x1,y0,y1 coordinates of a rectangular area
     * \param[in] ca an area of a cellular automata
     */
    void DiffusionInsideArea(int x0, int x1, int y0, int y1,
        Inhabitant [,] ca)
    {
        float p = 0;
        for (int i = x0; i <= x1; i++)
            for (int j = y0; j < y1; j++)
                if (ca[i, j] != null)
                    if (!ca[i, j].Infected)
                    {
                        p = (float)expert.mt.genrand_real3();
                        if (p < infectionRate)
                           {
                                ca[i, j].Infected = true;
                                ca[i, j].IncreaseNmbOfInfections();
                                CurrentInfected++;
                            }
                    }
        totalInfected += CurrentInfected;
    }

    /*
     * Spreads the disease in the area, that defined by the radius.
     * In the case of boundary positions the disease is spreading on a
         space torus like folding
     * \param[in] ca an area of a cellular automata
     * */
    public void Diffusion(Inhabitant [,] ca)
    {
        if (lifetime == 0) //virus is dying
        {
            ChangeVirusesEnvirement(-1); //change enviroment
                information
```

```
            expert.stats.AddVirusData(this); //append data to a
                statistics collector
        }
        else
        {
            lifetime--; // decrease the remaining lifetime

            int[] coords = expert.GetBorders(X, Y, radius);
            int strategy = (coords[2] > coords[3] ? 1 : 0) + (coords
                [0] > coords[1] ? 2 : 0); //select the borders of
                spreading the decease









            switch (strategy)
            {
                case (0): // a virus is located inside the area
                    DiffusionInsideArea(coords[0], coords[1], coords
                        [2], coords[3], ca);
                    break;
                case (1): //a virus is located beside a vertical
                    border
                    DiffusionInsideArea(coords[0], coords[1], 0,
                        coords[3], ca);
                    DiffusionInsideArea(coords[0], coords[1], coords
                        [2], expert.Width - 1, ca);
                    break;
                case (2):  //a virus is located beside a horisontal
                    border
                    DiffusionInsideArea(0, coords[1], coords[2],
                        coords[3], ca);
                    DiffusionInsideArea(coords[0], expert.Height - 1,
                        coords[2], coords[3], ca);
                    break;
                case (3): //a virus is located beside a corner
                    DiffusionInsideArea(0, coords[1], 0, coords[3], ca
                        );
                    DiffusionInsideArea(coords[0], expert.Height - 1,
                        0, coords[3], ca);
                    DiffusionInsideArea(0, coords[1], coords[2],
                        expert.Width - 1 - 1, ca);
                    DiffusionInsideArea(coords[0], expert.Height - 1,
                        coords[2], expert.Width - 1, ca);
                    break;
            }
        }
    }
```

```
/**
* Updates information about viruses in a given area
* Increases rate of viruses, since a new virus appears
* \param[in] x0,x1,y0,y2 coordinates of the rectangular area
* \param[in] sign the sign of changes (increasing or decreasing)
* */
void ChangeVirusesInArea(int x0, int x1, int y0, int y1, short
    sign)
{
    for (int i = x0; i <= x1; i++)
        for (int j = y0; j < y1; j++)
            expert.IncreaseVirsusDensity(i,j,(float)sign/square);
}

/**
 * Updates information about viruses in a given neigborhood
 * Increases rate of viruses, since a new virus appears
 * \param[in] an area of a cellular automata
 * */
void ChangeVirusesEnvirement(short sign)
{
    int[] coords = expert.GetBorders(X, Y, radius);
    int strategy = (coords[2] > coords[3] ? 1 : 0) + (coords[0] >
       coords[1] ? 2 : 0);

    switch (strategy)
    {
        case (0):
            ChangeVirusesInArea(coords[0], coords[1], coords[2],
               coords[3], sign);
            break;
        case (1):
            ChangeVirusesInArea(coords[0], coords[1], 0, coords
               [3], sign);
            ChangeVirusesInArea(coords[0], coords[1], coords[2],
               expert.Width - 1, sign);
            break;
        case (2):
            ChangeVirusesInArea(0, coords[1], coords[2], coords
               [3], sign);
            ChangeVirusesInArea(coords[0], expert.Height - 1,
               coords[2], coords[3], sign);
            break;
        case (3):
            ChangeVirusesInArea(0, coords[1], 0, coords[3], sign);
            ChangeVirusesInArea(coords[0], expert.Height - 1, 0,
               coords[3], sign);
            ChangeVirusesInArea(0, coords[1], coords[2], expert.
               Width - 1 - 1, sign);
            ChangeVirusesInArea(coords[0], expert.Height - 1,
               coords[2], expert.Width - 1, sign);
            break;
    }
```

```
        }

    }
}
```

B.2 Class "Inhabitant"

```
using System;

namespace DES_Lab6
{
    /**
     * \brief Inhabitants are moving agents, susceptible to infection.
     *
     * Inhabitants have a radius of moving (a maximal distance of a
        movement). Each instance also has a specific health state
     * If an instance moves into an occupied cell, a conflict is solved
        taking into account a status of the disease (infected/noninfected)
     **/

    class Inhabitant : MASObject
    {
        /// <summary>
        /// A probability to recover (difune health state)
        /// </summary>
        float recoveryRate =0; //
        /// <summary>
        /// Share of infected inhabitants in the movement area
        /// </summary>
        float infectedRate;
        /// <summary>
        /// The number of conflictes Infected-Infected
        /// </summary>
        int nmbOfConflictsII = 0;
        /// <summary>
        /// The number of conflicts Infected-Noninfected
        /// </summary>
        int nmbOfConflictsINI = 0;
        /// <summary>
        /// The number of conflicts NonInfected-NonInfected
        /// </summary>
        int nmbOfConflictsININ = 0;
        /// <summary>
        /// Last coordinates of the successful movement
        /// </summary>
        int oldX, oldY;
        /// <summary>
        /// The number of infections
        /// </summary>
        int nmbOfInfections = 0;
        /// <summary>
        /// The total number of relocation during the simulation when
           inhabutant is noninfected
        /// </summary>
```

```
int nmbTotalRelocationNonInfected = 0;
/// <summary>
/// Weights for viruses and inhabitants density, which are used
   for selecting the best position in a neighborhood
/// </summary>
float wV = 1;
float wI = (float)0.3;

public int NmbOfConflictsII
{
    get { return nmbOfConflictsII; }
}

public int NmbOfConflictsINI
{
    get { return nmbOfConflictsINI; }
}

public int NmbOfConflictsININ
{
    get { return nmbOfConflictsININ; }
}

public float RecoveryRate
{
    get { return recoveryRate; }
}

/// <summary>
/// Indected status
/// </summary>
public bool Infected { get; set; }

/// <summary>
/// The duration of disease (how long the agent was infected,
   number of iterations)
/// </summary>
public int Quarantine { get; set; }

public void IncreaseNmbOfInfections()
{
    nmbOfInfections++;
}

/// <summary>
/// The total number of relocation during the simulation when
   inhabutant is infected
/// </summary>
int nmbTotalRelocationInfected = 0;
public int NmbTotalRelocationInfected
{
    get { return nmbTotalRelocationInfected; }
}
```

```
public int NmbTotalRelocationNonInfected
{
    get { return nmbTotalRelocationNonInfected; }
}

/**
 * Creates the instance of the Instant class
 * in the random position in unique place for each instance (
    without collisions with other instances of Inhabitant and
    Virus classes)
 * \param[in] expert an instance of an Expert class
 * \param[in] id an id of a new-generated instance
 */
public Inhabitant(Expert expert, int id)
    : base(expert, id)
{
    this.expert = expert;
    recoveryRate = expert.RecoveryRate;
    if (recoveryRate == -1)
        recoveryRate = (float)expert.mt.genrand_RealInInterval(0,
           1);
    int[] coords;
    bool collision = false;
    do
    {
        coords = expert.GetSomeCoordinates();
        collision = expert.DetectInhabitantCollision(coords[0],
           coords[1]);
    } while (collision);

    nmbOfInfections = 0;
    oldX = X = coords[0];
    oldY = Y = coords[1];


    Infected = false;
    SetInfectedRate(expert.ca);
    ChangeInhabitantsEnviroment(1);
}

/*
 * Calculates the number of the infected agents in the given
    rectangular area.
 * \param[in] x0,x1,y0,y1 of a rectangular area
 * \param[in] ca a cellular automata area
 * \param[out] infectedRate the number of infected inhabitants in
    a given rectangle area
 * */
int SetInfectedInsideArea(int x0, int x1, int y0, int y1,
   Inhabitant [,] ca)
{
    int population = 0;
    int infectedRate = 0;
    for (int i = x0; i <= x1; i++)
```

```
            for (int j = y0; j <= y1; j++)
                if (ca[i, j] != null)
                {
                    population++;
                    if (ca[i, j].Infected)
                        infectedRate++;
                }
        return infectedRate;
    }

    /**
     * Calculates the rate of the infected agents in a new
        heighborhood of destination cell
     * In the case of boundary positions the calculation is spreading
        on a space torus like folding
     *\param[in] ca ca a cellular automata area
     */
    void SetInfectedRate(Inhabitant [,] ca)
    {
        int[] coords = expert.GetBorders(X, Y, this.radius);
        int strategy = (coords[2] > coords[3] ? 1 : 0) + (coords[0] >
            coords[1] ? 2 : 0);
        infectedRate = 0;
        switch (strategy)
        {
            case (0):
                infectedRate = SetInfectedInsideArea(coords[0], coords
                    [1], coords[2], coords[3], ca);
                break;
            case (1):
                infectedRate = SetInfectedInsideArea(coords[0], coords
                    [1], 0, coords[3], ca) +
                    SetInfectedInsideArea(coords[0], coords[1], coords
                        [2], expert.Width - 1, ca);
                break;
            case (2):
                infectedRate = SetInfectedInsideArea(0, coords[1],
                    coords[2], coords[3], ca) +
                    SetInfectedInsideArea(coords[0], expert.Height -
                        1, coords[2], coords[3], ca);
                break;
            case (3):
                infectedRate = SetInfectedInsideArea(0, coords[1], 0,
                    coords[3], ca) +
                    SetInfectedInsideArea(coords[0], expert.Height -
                        1, 0, coords[3], ca) +
                    SetInfectedInsideArea(0, coords[1], coords[2],
                        expert.Width - 1, ca) +
                    SetInfectedInsideArea(coords[0], expert.Height -
                        1, coords[2], expert.Width - 1, ca);
                break;
        }
        infectedRate -= ((ca[X, Y] != null) && (ca[X, Y].Infected)) ?
            1 : 0;
```

```
        infectedRate /= (square - 1);
    }

    /**
     * Resolves conflict, when two agent try to take the same cell.
     * An agent priority is defined by status of agents (noninfected
        one has the highest priority),
     * in the case of similar statuses agent with the best health (
        healthStatus value) has the highest priority
     * \param[in] x1,y1 coordimates of a cell
     * \param[in] caFrom,caTo departure and destination aread of a
        cellular automata
     */
    void ResolveConflict(int x1, int y1,  ref Inhabitant [,] caFrom,
       ref Inhabitant [,] caTo)
    {
        if (caTo[x1, y1].Infected != Infected)
        {
            nmbOfConflictsINI++;
            if (caTo[x1, y1].Infected)
            {
                caTo[x1, y1].FindAnotherPlace(ref caTo, ref caTo);
                caTo[x1, y1] = this;
                caFrom[X, Y] = null;
                ChangeInhabitantsEnviroment(-1);
                oldX = X = x1;
                oldY = Y = y1;
                ChangeInhabitantsEnviroment(1);
            }
            else
                FindAnotherPlace(ref caFrom, ref caTo);
        }
        else
        {
            if (caTo[x1, y1].Infected)
                nmbOfConflictsII++;
            else
                nmbOfConflictsININ++;
            if (recoveryRate > caTo[x1, y1].recoveryRate)
            {
                caTo[x1, y1].FindAnotherPlace(ref caTo, ref caTo);
                caTo[x1, y1] = this;
                caFrom[X, Y] = null;
                ChangeInhabitantsEnviroment(-1);
                oldX = X = x1;
                oldY = Y = y1;
                ChangeInhabitantsEnviroment(1);
            }
            else
                FindAnotherPlace(ref caFrom, ref caTo);
        }
    }

    /**
```

```
 * Looks for another position in a radius of movements
 * If a current neighborhood is overcrowded (more that 2*radius
    attempts to take a cell), a radius is increased.
 * \param[in] caFrom,caTo departure and destination aread of a
    cellular automata
 */
void FindAnotherPlace(ref Inhabitant[,] caFrom, ref Inhabitant[,]
   caTo)
{
    bool notRemoved = true;
    int curRadius = radius;
    int attempt = 0;
    do
    {
        notRemoved = Dislocate(curRadius, ref caFrom, ref caTo);
        if (attempt > 2 * curRadius)
        {
            attempt = 0;
            curRadius+=radius;
            if ((curRadius > caTo.GetLength(0) / 2) || (curRadius
               > caTo.GetLength(1) / 2))
                curRadius = Math.Min(caTo.GetLength(0), caTo.
                   GetLength(1));
        }
        attempt++;
    } while (notRemoved);
}

/**
 * Try to shift to random position in the neighborhood, defined by
    a given radius
 * \param [in] curRadius radius of the neighborhood, where
    movements are possible
 * \param[in] caFrom,caTo departure and destination aread of a
    cellular automata
 * \param[out] bool whether the dislocation was successful
 */
bool Dislocate(int curRadius, ref Inhabitant[,] caFrom, ref
   Inhabitant[,] caTo)
{
    int[] coords = expert.GetNewCoordinates(oldX, oldY, curRadius)
       ;
    if ((caTo[coords[0], coords[1]] == null) && (!expert.
       OccupiedByVirus(coords[0], coords[1])))
    {
        caTo[coords[0], coords[1]] = this;
        caFrom[X, Y] = null;
        ChangeInhabitantsEnviroment(-1);
        oldX = X = coords[0];
        oldY = Y = coords[1];
        ChangeInhabitantsEnviroment(1);
        if (Infected)
            nmbTotalRelocationInfected++;
        else
```

```
                nmbTotalRelocationNonInfected++;
            return false;
        }
        else
        {
            if (Infected)
                nmbTotalRelocationInfected++;
            else
                nmbTotalRelocationNonInfected++;
            return true;
        }
    }

    /**
     * Moving the agent to a random position inside the neighborhood.
        If some conflicts happen, another functions for resolving of
        collisions will be called
     * \param[in] caFrom,caTo departure and destination aread of a
        cellular automata
     * \param[out] bool whether the inhabitant became susceptible
     * */
    public bool Move(ref Inhabitant[,] caFrom, ref  Inhabitant[,] caTo
        )
    {
        bool recovered = false;
        if (Infected)
        {
            double rndRecovery = expert.mt.genrand_RealInInterval(0,
                1);
            if (rndRecovery < recoveryRate)
            {
                Infected = false;
                recovered = true;
            }
            else
                Quarantine++;
        }

        int[] coords = expert.GetNewCoordinates(oldX, oldY, radius);
        if (caTo[coords[0], coords[1]] != null)
            ResolveConflict(coords[0], coords[1],  ref caFrom, ref
                caTo);
        else if (expert.OccupiedByVirus(coords[0], coords[1]))
            this.FindAnotherPlace(ref caFrom, ref caTo);
        else
        {
            caTo[coords[0], coords[1]] = this;
            caFrom[X, Y] = null;
            ChangeInhabitantsEnviroment(-1);
            oldX = X = coords[0];
            oldY = Y = coords[1];
            ChangeInhabitantsEnviroment(1);
        }
        return recovered;
```

```
}

/**
 * Moving the agent to the best position inside the neighborhood.
    If some conflicts happen, another functions for resolving of
    collisions will be called
 * \param[in] caFrom,caTo departure and destination aread of a
    cellular automata
 * \param[out] bool whether the inhabitant became susceptible
 * */
public bool CognitiveMove(ref Inhabitant[,] caFrom, ref
   Inhabitant[,] caTo)
{
    bool recovered = false;
    if (Infected)
    {
        double rndRecovery = expert.mt.genrand_RealInInterval(0,
           1);
        if (rndRecovery < recoveryRate)
        {
            Infected = false;
            recovered = true;
        }
        else
            Quarantine++;
    }
    int[] coords = GetBestCellInNeiborhoood(oldX, oldY, radius,
       caTo);

    if (caTo[coords[0], coords[1]] != null)
        ResolveConflict(coords[0], coords[1], ref caFrom, ref caTo
           );
    else
    {
        caTo[coords[0], coords[1]] = this;
        caFrom[X, Y] = null;
        ChangeInhabitantsEnviroment(-1);
        oldX = X = coords[0];
        oldY = Y = coords[1];
        ChangeInhabitantsEnviroment(1);
    }
    return recovered;
}

/**
 * Returns coordinates of the most favourable for living cell in a
     neigborhood
 * \param[in] x, y center of the neigborhood
 * \param[in] radius radius of the neigborhood
 * \param[in] ca area where cells are disposed
 * \param[out] retCoords coordinates of a new cell
*/
public int[] GetBestCellInNeiborhoood(int x, int y, int radius,
   Inhabitant [,] ca)
```

```
{
    int[] coords = expert.GetBorders(X, Y, radius);
    int[] retCoords = new int[2] { 0, 0 };
    int strategy = (coords[2] > coords[3] ? 1 : 0) + (coords[0] >
       coords[1] ? 2 : 0);
    infectedRate = 0;
    float[] out1 = new float[3] { 0, 0, 0 };
    float[] out2 = new float[3] { 0, 0, 0 };

    switch (strategy)
    {
        case (0):
            out1 = GetBestCell(coords[0], coords[1], coords[2],
               coords[3], wV, wI, ca);
            retCoords[0] = (int)out1[1];
            retCoords[1] = (int)out1[2];
            break;
        case (1):
            out1 = GetBestCell(coords[0], coords[1], 0, coords[3],
                wV, wI, ca);
            out2 = GetBestCell(coords[0], coords[1], coords[2],
               expert.Width - 1, wV, wI, ca);
            if (out1[0] > out2[0])
            {
                retCoords[0] = (int)out1[1];
                retCoords[1] = (int)out1[2];
            }
            else
            {
                retCoords[0] = (int)out2[1];
                retCoords[1] = (int)out2[2];
            }
            break;
        case (2):
            out1 = GetBestCell(0, coords[1], coords[2], coords[3],
                wV, wI, ca);
            out2 = GetBestCell(coords[0], expert.Height - 1,
               coords[2], coords[3], wV, wI, ca);
            if (out1[0] > out2[0])
            {
                retCoords[0] = (int)out1[1];
                retCoords[1] = (int)out1[2];
            }
            else
            {
                retCoords[0] = (int)out2[1];
                retCoords[1] = (int)out2[2];
            }
            break;
        case (3):
            float[] out3 = new float[3] { 0, 0, 0 };
            out1 = GetBestCell(0, coords[1], 0, coords[3], wV, wI,
                ca);
```

```
                out2 = GetBestCell(coords[0], expert.Height - 1, 0,
                    coords[3], wV, wI, ca);
                if (out1[0] > out2[0])
                {
                    out3[0] = (int)out1[0];
                    out3[1] = (int)out1[1];
                    out3[2] = (int)out1[2];
                }
                else
                {
                    out3[0] = (int)out2[0];
                    out3[1] = (int)out2[1];
                    out3[2] = (int)out2[2];
                }
                out1 = GetBestCell(0, coords[1], coords[2], expert.
                    Width - 1, wV, wI, ca);
                out2 = GetBestCell(coords[0], expert.Height - 1,
                    coords[2], expert.Width - 1, wV, wI, ca);
                if (out1[0] > out2[0])
                {
                    if (out1[0] > out3[0])
                    {
                        retCoords[0] = (int)out1[1];
                        retCoords[1] = (int)out1[2];
                    }
                    else
                    {
                        retCoords[0] = (int)out3[1];
                        retCoords[1] = (int)out3[2];
                    }
                }
                else
                {
                    if (out2[0] > out3[0])
                    {
                        retCoords[0] = (int)out2[1];
                        retCoords[1] = (int)out2[2];
                    }
                    else
                    {
                        retCoords[0] = (int)out3[1];
                        retCoords[1] = (int)out3[2];
                    }
                }
                break;
            }
            return retCoords;
        }

        /**
        * Returns coordinates of the most favourable for living cell in a
            rectungle area
        * \param[in] x0, x1, y0, y1 coordinates of the rectangular area
```

```
* \param[in] weightV, weightI - cooficients of the objective
    function (for estimating cells)
* \param[in] ca area of a cellular automata
* \param[out] array an 1x3-dim array that contains an assessment
    of the best position and its coordinates
**/
float[] GetBestCell(int x0, int x1, int y0, int y1, float weightV,
     float weightI, Inhabitant [,] ca)
{
     float bestVal = Int16.MinValue, val = Int16.MinValue;
     int xBest = -1, yBest = -1;
     for (int i = x0; i <= x1; i++)
         for (int j = y0; j <= y1; j++)
         {
              val = - weightI * expert.GetInhabitantDensity(i, j) -
                 weightV * expert.GetVirusesDensity(i, j);
              if ((val > bestVal) && (!expert.OccupiedByVirus(i, j))
                 )
              {
                      xBest = i;
                      yBest = j;
                      bestVal = val;
              }
         }
     return new float[3] { bestVal, xBest, yBest };
}

/**
* Change the enviroment information about density in a rectangular
     area
* The corresponding duration of the disease is assigned.
* \param[in] x0,x1,y0,y1coordinates of the rectangular area
* \param[in] sign 1/-1 for departure/destination points
    accordingly
*/
void InhabitantsInsideArea(int x0, int x1, int y0, int y1, short
    sign)
{
     for (int i = x0; i <= x1; i++)
         for (int j = y0; j < y1; j++)
              expert.ChangeInhabitantDensity(i, j, (float)sign /
                 square);
}

/**
 * Change the enviroment information about density in the
     neigborhood of departure and destination poing
 * In the case of boundary positions the disease is spreading on a
      space torus like folding
 * \param[in] sign 1/-1 for departure/destination points
     accordingly
 * */
public void ChangeInhabitantsEnviroment(short sign)
{
```

```
            int[] coords = expert.GetBorders(X, Y, radius);
            int strategy = (coords[2] > coords[3] ? 1 : 0) + (coords[0] >
                coords[1] ? 2 : 0);

            switch (strategy)
            {
                case (0):
                    InhabitantsInsideArea(coords[0], coords[1], coords[2],
                        coords[3], sign);
                    break;
                case (1):
                    InhabitantsInsideArea(coords[0], coords[1], 0, coords
                        [3], sign);
                    InhabitantsInsideArea(coords[0], coords[1], coords[2],
                        expert.Width - 1, sign);
                    break;
                case (2):
                    InhabitantsInsideArea(0, coords[1], coords[2], coords
                        [3], sign);
                    InhabitantsInsideArea(coords[0], expert.Height - 1,
                        coords[2], coords[3], sign);
                    break;
                case (3):
                    InhabitantsInsideArea(0, coords[1], 0, coords[3], sign
                        );
                    InhabitantsInsideArea(coords[0], expert.Height - 1, 0,
                        coords[3], sign);
                    InhabitantsInsideArea(0, coords[1], coords[2], expert.
                        Width - 1 - 1, sign);
                    InhabitantsInsideArea(coords[0], expert.Height - 1,
                        coords[2], expert.Width - 1, sign);
                    break;
            }
        }
    }

}
```