The Creation of Simulation with an Algorithm Optimisation in Java for the Teaching Process

Roman Horváth – Jana Fialová
Department of Mathematics and Computer Science,
Faculty of Education, Trnava University in Trnava, Trnava, Slovakia
roman.horvath@truni.sk, jana.fialova@truni.sk

***Abstract*—At the Faculty of Education, we decided to use the current situation related to the spread of the COVID-19 virus. In one of the subjects whose syllabus corresponds to this topic, we included the creation of a simulation of the spread of the disease. While creating the sample example we describe in this article, we found that some original Java algorithms used in our programming framework are not as efficient as they could be. So, we focused on optimising these algorithms (a pair of related algorithms in the field of analytical geometry). Since it was an activity performed as a part of the creation of this simulation, we decided to publish it in this paper as well.**

I. Introduction

The subject of *modelling and simulation of systems* (MSS) is taught in the study programme *computer science in combination with another subject* (CS) at the Faculty of Education of Trnava University in Trnava. The current situation gave us the idea of including a simulation of the spread of viruses in this subject. Nevertheless, the simulation had to be created first.

Often the SIR or SIRS model is used [1, 2, 3]. This model divides the individuals into three groups or states S – susceptible, I – infected, and R – removed. In short, the susceptible subjects are those who were not infected, and the removed might be those who were healed (or died). The last S in SIRS model means that removed subjects may return to the susceptible state and be infected again.

The simulation was ready to run after setting and implementing a series of rules of how to transit between the states. Mostly the solution was to define a set of values (constants or variables) that express the ranges of probabilities for a change to occur. The significant advantage (due to its good visual representation) is to implement the subjects (persons) as moving particles and take the probability value ranges into account according to the distance of particles.

II. The Simulation Setup

The most common is to use the SIR model. There are countless ways on how to implement this model, e.g. [4, 5]. Our implementation was divided into few phases. The first phase included adopting the model to our needs. As we show, we set the rules slightly different from the original model. It is more like the SEIR model [6], but it distinguishes even more states: susceptible (clean), exposed (infected, but not sick), sick (infected1), quarantined (infected2), healed (removed1), and deceased (removed2). The exposed ones cannot be quarantined because they do not know yet that they spread the virus. They will become sick after incubation time. The sick and quarantined may heal or die; the sick ones, though, become quarantined after some delay needed to "lock them down."

Each particle (person) will be represented by dot with a specific colour that expresses its state. The circle around the dot indicates that the person is isolated in the quarantine and cannot move or affect other persons. Later, as the simulation improves, we will implement the option for the persons to stay home, which will also be indicated by a circle around a particular dot. In parallel with the movement of persons on the screen, we will draw a graph on the screen placed underneath the dots and expressing ratio changes in time of the simulated persons' states. More details are available (in Slovak) at [7].

In the preparation phase, we considered the use of an algorithm computing the distance of point from a line segment (wall) that is tightly connected to the computationally simpler version algorithm – computing the distance of a point from a line. The framework used in our work [8, 9] (that had been developed and actively used in many ways for about ten years; see for example: [10, 11, 12]) contains several algorithms in the field of analytical geometry: the nearest point on the circle, the nearest point on the line, the nearest point on the line segment, the intersection of the lines, the intersection of the line segments, the intersections of the circles, the distance of the line from the circle, and so on. Most of them are our implementation, in fact, with one exception: the calculation of the distance of a point from a line and a line segment – just the algorithm we wanted to use in this work. It gave us a feeling of inconsistency, so before using it, we decided to perform a few tests and use them to decide whether to continue to use "the foreign implementation" in the programming framework, or not.

III. Optimising the Point to Line Distance Algorithm

The first tests concerned the algorithm for calculating the distance of a point from a line. Because we do not have direct access to the source code of the current Java implementation (build 1.8.0_241-b07), we selected implementations from freely available online sources [13, 14, 15]. The implementation [13] was the closest to our way of implementation and was suitable for optimisation. We decided to optimise it and after a few tests use it in our framework. The optimisation process is described here using Java/Python-like pseudocode. The pseudocode comes directly from Java code, but to save room, the blocks are distinguished by indentation (like in Python), declarations are made by the first occurrence of the variables (all variables

978-0-7381-2367-7/20/$31.00 ©2020 IEEE

are of the `double` type) and unnecessary characters and keywords (like the `double` type in declarations; the "lonely" semicolons – at ends of the lines, but not between commands at the same line; or the parentheses in the `if` statement) were removed. To spare more space and improve the readability, the lines of code are numbered (so the identical lines do not have to be repeated, and changed rows can be referenced).

As mentioned above the algorithm for calculating the distance between a point and a line was taken from a source [13]. The original algorithm follows. It receives the input values in six parameters – coordinates of two points placed on a line (x_1, y_1 and x_2, y_2) and the coordinates of a point (p_x, p_y) to which we want to know the distance. The return value is the square of the distance between the line and the third point (this is why the original Java method is named `ptLineDistSq`).

```
    // The first line calculates the denominator pd2 for the parameter
    // u (at line 05, below; to get nearest point on the line – x, y):
01: pd2 = (x1 – x2) * (x1 – x2) + (y1 – y2) * (y1 – y2)
02: if pd2 == 0          // That means that points are coincident.
        // Set the nearest point coordinates (x, y) to any original
        // x and y:
03:     x = x1; y = y2
04: else
        // Calculate parameter u and then coordinates x, y:
05:     u = ((px – x1) * (x2 – x1) + (py – y1) * (y2 – y1)) / pd2
06:     x = x1 + u * (x2 – x1)
07:     y = y1 + u * (y2 – y1)
    // Calculate the squared distance of x, y from px, py:
08: return (x – px) * (x – px) + (y – py) * (y – py)
```

Optimisation step 1

Line 01: Since there are multiplications of the same terms, we may reverse the subtraction terms, and the result remains the same:

```
01: pd2 = (x2 – x1) * (x2 – x1) + (y2 – y1) * (y2 – y1)
```

Line 03: There is no reason for use y_2 since when p_{d2} (the distance of points on the line) is zero, y_1 is identical to y_2 so that we will change it to y_1, it will help us further:

```
03:     x = x1; y = y1
```

Optimisation step 2

We will precalculate the subtractions of line coordinates (used, e.g. at line 01) and replace them wherever they occur. This step shifts all line numbers, but we will not repeat them for now. The replacements are at lines 02, 06, 07, and 08 (according to new numbering):

```
01: x2 –= x1; y2 –= y1      // (This line is new and the first
02: pd2 = x2 * x2 + y2 * y2 // replacement is next to it.)
    ...
06:     u = ((px – x1) * x2 + (py – y1) * y2) / pd2
07:     x = x1 + u * x2
08:     y = y1 + u * y2
```

Optimisation step 3

We will shift the whole system of three points to zero by eliminating the values of x_1 and y_1 within. Since the subtraction of coordinates in x_2 and y_2 remains the same (we already did the subtraction in the previous optimisation step), the change applies only to p_x, p_y and x, y. To keep track, we repeat all lines with new numbering:

```
01: x2 –= x1; y2 –= y1
02: px –= x1; py –= y1      // This eliminates x1, y1 from px, py.
03: pd2 = x2 * x2 + y2 * y2
04: if pd2 == 0
05:     x = 0; y = 0// This eliminates x1, y1 from x, y when
                    // pd2 == 0.
06: else
07:     u = (px * x2 + py * y2) / pd2
        // And this in the other case:
08:     x = u * x2
09:     y = u * y2
    // Here are redundant (repeating) subtractions of px and py.
    // We can eliminate them, too – in the next step.
10: return (x – px) * (x – px) + (y – py) * (y – py)
```

Optimisation step 4

We will simplify the line 10. If we subtract the p_x and p_y values on lines 05, 08, and 09, then we do not have (we must not) to do it on line 10:

```
    ...
05:     x = –px; y = –py
    ...
08:     x = u * x2 – px
09:     y = u * y2 – py
10: return x * x + y * y
```

Optimisation step 5

If we "extract" the subtraction of p_x and p_y from both `if-else` branches, we will get rid of one of the branches altogether:

```
01: x2 –= x1; y2 –= y1
02: px –= x1; py –= y1
03: pd2 = x2 * x2 + y2 * y2
04: x = –px; y = –py
05: if pd2 != 0
06:     u = (px * x2 + py * y2) / pd2
07:     x += u * x2
08:     y += u * y2
09: return x * x + y * y
```

After the optimisation, we performed a series of tests. We measured the time of two million runs of each algorithm (with precisely the same pre-generated random data sets) and that task we repeated 50 times. From the gained data we calculated the median ($\tilde{x}$). The gained data is in table 1. The meanings of the columns and rows are as follows:

{Aln} – the Java build 1.8.0_241-b07 implementation.

{Bln} – the algorithm taken from [13].

978-0-7381-2367-7/20/$31.00 ©2020 IEEE

{Cln} – the optimised version of the **{Bln}** algorithm.

{Dln} – the algorithm taken from [14].

{Eln} – the algorithm taken from [15].

$\tilde{x}$ – the median of the measured time values in milliseconds.

NaNs – number of `NaN` values produced in two million runs of the algorithm (receiving random parameters).

Table 1. The data sets for the algorithm of point to line distance (squared) – the first batch.

	{Aln}	{Bln}	{Cln}	{Dln}	{Eln}
run 1					
$\tilde{x}$	14031.3	0.8	0.2	12256.3	0.8
NaNs	5	0	0	5	5
run 2					
$\tilde{x}$	14282.6	0.8	0.4	12486.9	0.8
NaNs	4	0	0	4	4
run 3					
$\tilde{x}$	14318.9	0.85	0.2	12514.55	0.9
NaNs	5	0	0	5	5
run 4					
$\tilde{x}$	14138.8	0.8	0.2	12166.95	0.7
NaNs	4	0	0	4	4
run 5					
$\tilde{x}$	14045.3	0.8	0.5	12170.2	0.7
NaNs	4	0	0	4	4

Further processing follows below the second algorithm optimisation steps. We did not see any room for optimisation of the algorithm [14]. Notice that three algorithms [13, 14, 15] produced the NaNs. This fact only should be sufficient to reject the algorithms; nevertheless, we tested them and compared the data sets to show whether there is a significant difference in time.

IV. Optimising the Point to Line Segment Distance Algorithm

The second algorithm (calculating the distance of a point from a line segment) is in principle the same as the previous one, except that here we need to ensure that the point lies precisely on the line segment (not on the line as a whole). The original algorithm (taken from source [13]) looks like this:

```
01: pd2 = (x1 − x2) * (x1 − x2) + (y1 − y2) * (y1 − y2)
02: if pd2 == 0          // The points are coincident.
03:         x = x1; y = y2
04: else
05:         u = ((px − x1) * (x2 − x1) + (py − y1) * (y2 − y1)) / pd2
06:         if u < 0     // The point lies outside the first end
                         // of the segment.
07:             x = x1; y = y1
08:         elseif u > 1.0   // The point lies outside the other
                             // end of the segment.
09:             x = x2; y = y2
10:         else
11:             x = x1 + u * (x2 − x1)
12:             y = y1 + u * (y2 − y1)
13: return (x − px) * (x − px) + (y − py) * (y − py)
```

Optimisation step 1

We set up a temporary substitution first – until we prove that we can safely use the same subtractions ($x_2 -= x_1$; $y_2 -= y_1$) as in the point to line distance algorithm:

$\Delta x = x_2 - x_1$

$\Delta y = y_2 - y_1$

Like in the previous case, we may rewrite the subtractions $x_1 - x_2$, $y_1 - y_2$ at line 01 the other way around: $x_2 - x_1$, $y_2 - y_1$. Thanks to that we use the substitution at lines 01, 05, 11, and 12:

```
01: pd2 = Δx * Δx + Δy * Δy
    ...
05:         u = ((px − x1) * Δx + (py − y1) * Δy) / pd2
    ...
11:             x = x1 + u * Δx
12:             y = y1 + u * Δy
```

Optimisation step 2

In this step we shift the system to zero, that means that we choose one point (x_1, y_1) that we subtract from all points in the system – including itself, which makes it the zero point. Take a look at these lines of code:

```
x2 = x2 − x1          // (Will be rewritten as: x2 −= x1)
y2 = y2 − y1          // (Will be rewritten as: y2 −= y1)
```

Now we recognise that Δx and Δy are modified (shifted) x_2 and y_2 so that we can dismiss the Δx and Δy substitutions. All changes are transferred here:

```
01: x2 −= x1; y2 −= y1
02: pd2 = x2 * x2 + y2 * y2
```

Since the x_1 and y_2 became zeroes, we will ignore them on the lines below.

```
03: // x1 = x1 − x1  means that  x1 = 0
04: // y1 = y1 − y1  means that  y1 = 0
05: px = px − x1 // (Will be rewritten as: px −= x1)
06: py = py − y1 // (Will be rewritten as: py −= y1)
07: if pd2 == 0
```

See line 03 in the original algorithm. There y_2 has the same value as y_1 so that we can write:

```
08:         x = 0; y = 0
09: else
10:         u = ((px − 0) * x2 + (py − 0) * y2) / pd2
11:         if u < 0
12:             x = 0; y = 0
13:         elseif u > 1.0
14:             x = x2; y = y2
15:         else
16:             x = 0 + u * x2
17:             y = 0 + u * y2
18: return (x − px) * (x − px) + (y − py) * (y − py)
```

In the next step, we will also remove the duplicate subtractions of p_x and p_y at line 18 by similar way as in the point to line distance algorithm – we subtract them from x and y in each branch above the line. (And we will optimise it further.)

978-0-7381-2367-7/20/$31.00 ©2020 IEEE

Optimisation step 3

To keep track, we repeat all lines of the code. The changes outlined above relate to the content of rows: 03, 05, 07, 09, 11, 12, 14, 15, and 16.

```
01: x2 −= x1; y2 −= y1
02: pd2 = x2 * x2 + y2 * y2
03: px −= x1; py −= y1
04: if pd2 == 0
05:         x = −px; y = −py
06: else
07:         u = (px * x2 + py * y2) / pd2
08:         if u < 0
09:                 x = −px; y = −py
10:         elseif u > 1.0
11:                 x = x2 − px
12:                 y = y2 − py
13:         else
14:                 x = u * x2 − px
15:                 y = u * y2 − py
16: return x * x + y * y
```

Optimisation step 4

In this step, we will extract the p_x and p_y subtractions from all branches and place them in front of them. This modification changes each assign operation inside the branches from the simple assignment (=) to the enhanced addition assignment (+=).

The extracted p_x and p_y will be at line 04. We must subtract them from x and y so that the line will look like this:

```
04: x = −px; y = −py
```

After changes at line 05 from previous optimisation step the line would look like this:

```
x += 0; y += 0
```

So, the whole branch became needless.

Changes on the previous version of line 09 would modify it into this form:

```
x += 0; y += 0
```

So, this sub-branch became needless, too. Next change is at lines 11 and 12 of the previous version. They will be merged into this form:

```
x += x2; y += y2
```

Last change is at lines 14 and 15 of the last version, which will afterwards look like this:

```
x += u * x2
y += u * y2
```

The final version of the optimised algorithm follows:

```
01: x2 −= x1; y2 −= y1
02: pd2 = x2 * x2 + y2 * y2
03: px −= x1; py −= y1
04: x = −px; y = −py
05: if pd2 != 0          // (This is the original else branch, so the
                         // condition is negated.)
06:         u = (px * x2 + py * y2) / pd2
07:         if u > 1.0
08:                 x += x2; y += y2
09:         elseif u >= 0.0   // (The original if u < 0 branch had disappeared,
                              // so here must be another condition.)
10:                 x += u * x2
11:                 y += u * y2
12: return x * x + y * y
```

Like before we measured 50 time values of two million runs of each algorithm. The medians are in table 2. The meanings of all symbols in the table are as follows:

{Asg} – the Java build 1.8.0_241-b07 implementation.

{Bsg} – the algorithm taken from [13].

{Csg} – the optimised version of the **{Bsg}** algorithm.

{Dsg} – the algorithm taken from [14].

{Esg} – the algorithm taken from [15].

$\tilde{x}$ – the median of the measured time values in milliseconds.

No algorithm produced NaNs in this batch.

Table 2. The data sets for the algorithm of point to line segment distance (squared) – the second batch.

run		{Asg}	{Bsg}	{Csg}	{Dsg}	{Esg}
1	$\tilde{x}$	30101.5	0.95	0.2	29755.55	28147.4
2	$\tilde{x}$	29215.2	0.8	0.3	28669	27864.6
3	$\tilde{x}$	30492.75	0.9	0.2	29911.9	28061.75
4	$\tilde{x}$	22175.65	0.6	0.1	22073.35	21607.45
5	$\tilde{x}$	22240.6	0.6	0.1	21911.85	21689.5

Next, we used statistical methods to compare the data sets whose medians differ in magnitudes of order. The goal was to show that there is a significant difference in time.

V. Testing the Time Performance of the Algorithms

To approve (or disprove) the normality of the data, we used the Shapiro–Wilk test [16]. Since the p-value < α for most sets, we reject the H_0 for them. That means that the data was not normally distributed. In other words, the difference between the data sets and the normal distribution was big enough to be statistically significant – using the Shapiro–Wilk test. We rejected the normal distribution of the data for the most data sets; therefore, we used the Mann–Whitney U nonparametric test in further analysis.

We used the calculated median values (in tables 1 and 2) to eliminate the nonsensical comparisons. Mann–Whitney U (nonparametric) test is meaningful only for such comparisons of data sets whose difference of medians is small (comparable within the order of magnitude). As we can see in the tables (1 and 2), the comparison is meaningful only for these pairs of data sets: **{Bln}** – **{Cln}**; **{Eln}** – **{Cln}**; and **{Bsg}** – **{Csg}**.

Table 3. The results of Mann–Whitney U nonparametric test for the meaningful comparisons of data sets (as described in the text).

#	{Bln} – {Cln}	{Eln} – {Cln}	{Bsg} – {Csg}
1	$u = 6.630$	$u = 5.141$	$u = 8.154$
2	$u = 7.551$	$u = 7.574$	$u = 7.991$
3	$u = 7.553$	$u = 7.571$	$u = 7.488$
4	$u = 7.524$	$u = 7.517$	$u = 7.849$
5	$u = 7.718$	$u = 7.731$	$u = 7.540$

978-0-7381-2367-7/20/$31.00 ©2020 IEEE

Table 3 shows the values of u that should be ≤ 1.96 for comparable (insignificantly different) data. Since $u > 1.96$ for all comparison, we reject the H_0 for all of them. That means that there is a significant difference in time between the data sets in all comparisons shown in table 3.

In the second batch (the point to line segment distance) only the **{Bsg}** – **{Csg}** data sets were compared (since the medians of times in all other cases were several magnitudes of order bigger). The result simply confirms that the optimisation was meaningful.

Based on the results we have obtained, we will use the **{Cln}** and **{Csg}** algorithms in our framework. The full Java code of both algorithms is available at [17] – see private methods `ptLineDistSq` and `ptSegDistSq` around lines 25,610 and 25,660 (the line numbers might not be exact, since the framework might be updated over time). We will also report this matter to Oracle, as we have had a positive experience with the error/bug/problem reporting in the past.

The optimisation processes and their description might be good example usable in the educational process. The topic of algorithm optimisation is not practically included in the study programmes at our faculty (practically, because according to the plans it should be taught, but mostly this is not possible for time reasons), but we can recommend this paper to the students for self-study and also maybe other colleagues may use this example in their educational process.

VI. The Simulation Implementation

The first phase was simple and straightforward – it implemented a person's initialisation (according to the ranges of attributes determined by the constants), produced the desired number of "people" and started the simulation.

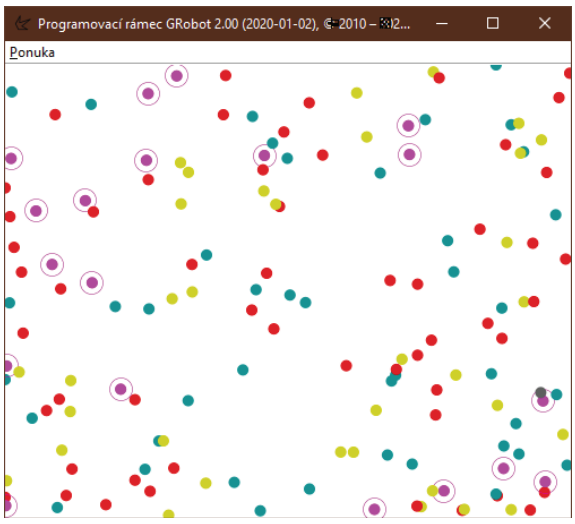

Figure 1. Moving particles on the canvas in the first implementation phase of the simulation.

The main action-driving elements of the simulation are the methods dealing with changes of a person's condition (the `activity` reaction in the `Person` class) and ensuring the contact of persons including the transmission of the disease (infection; the `contact` method in the `Person` class which is called in the `tick` method of the `Main` class). After finishing this phase, the dots start to move. Their colours indicate their states. (See figure 1.)

The simulation can be restarted by clicking the mouse.

In the second phase, we added the possibility of monitoring and drawing statistics (that is: storing the number of people who are in a particular state at a particular moment in time and drawing the collected data in the form of a stacked area graph [18]).

The horizontal axis of the graph (x) is time, and the vertical axis (y) reflect the persons' state ratio. The graph rolls to the left forever after reaching the right corner of the screen. Currently, there is no option to roll back the data rolled out to the left (so the data is lost but given the purpose of the simulation we consider this as no problem).

After several runs of the simulation, we find that shortly after the stabilisation of the system, the data becomes monotonous and uninteresting. (The visual side of the application did not change further. See figure 2.)

In the third phase, we added one important feature – we allowed the person to stay at home, that means that we limited the mobility of people (and thus the spread of the disease).

This phase also included the transformation of the `status` attribute from an integer to an enumerated data type. It was not crucial for the simulation; it was just didactical mean to emphasise maintaining good habits in correct data representation and to show the effort needed to fix the bad initial design of the program structure. (In this case, the effort is not high, but we will emphasise to the students that the later a flaw is detected and the worse the design error is, the harder and more expensive is the fix.)

In the source code provided to the students, the time spent at home was set zero. Students should test the following result: If the values of `home1` and `home2` remained zero, the results of the simulation would be precisely the same as in the previous phase. Then they should change the time, rerun the simulation and compare the results.

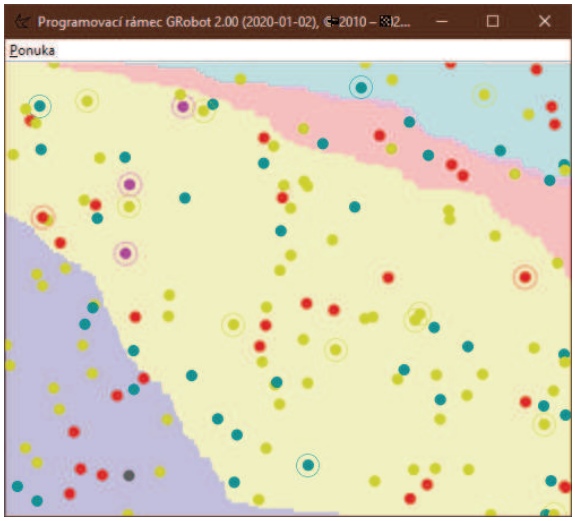

Figure 2. Moving particles over the stacked area graph in the fourth implementation phase of the simulation.

978-0-7381-2367-7/20/$31.00 ©2020 IEEE

In the last phase, we worked on interactive adjusting the simulation parameters and automatic save/load of the configuration (so that we do not have to restart the application all the time).

Graphs on figure 3 to figure 5 show the various situations that our illustrative but straightforward simulation has shown. We can compare situations where people have no movement restrictions on situations where these measures are gradually introduced. If we look at the graph showing high restriction of movement (figure 5) as we witnessed in spring '20 in Slovakia, we will see that the grey area is completely missing (this is the area of the "deceased particles"). It was for us a somewhat surprising result, given that nothing that should significantly help to get there was explicitly included in the simulation – no additional increase in the resilience of persons based on their staying at home, or anything like that. It emanated from the simulation naturally.

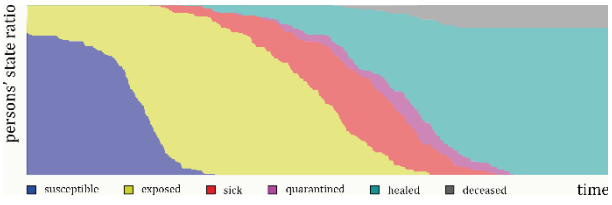

Figure 3. Graph showing changes in persons' status ratios over time when setting up no time of staying at home.

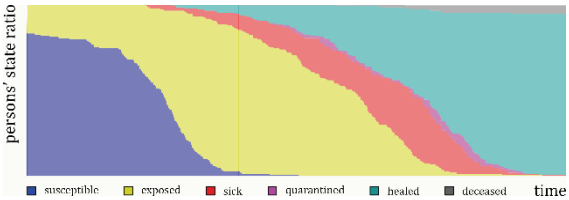

Figure 4. Graph showing changes in persons' status ratios over time when setting up a short time of staying at home.

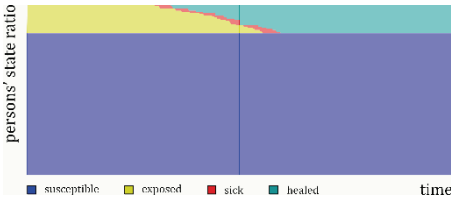

Figure 5. Graph showing changes in persons' status ratios over time when setting up a long time of staying at home.

Although this simulation is primarily intended to serve as an example in class (the simulation parameters are not perfectly tuned to COVID-19), it shows certain behaviour – we think that this shows off the effectiveness of this regulation quite well (if people follow it). We expected some small number of deceased even when the persons stay at home, but definitely not zero.

Some open issues (as homework for the students):

- the capacity of medical facilities (number of beds);
- dynamic scalability of the simulation – total number of persons and beds, configurable board size – that is, the size of the area on which persons would move (as this can affect the overall result too), and similar;
- alternatively, other possibly important aspects, such as the possibility of dividing the persons into some communities, the possibility of defining centres where the people can (or must) meet (like shops, medical facilities, and similar).

VII. Conclusion

In conclusion, we can state that testing the behaviour and functionality of the didactic simulation as well as the series of tests verifying the time efficiency of the optimised algorithms described in this paper proved the success of our efforts. We included the optimised algorithms in our programming framework, and the didactic simulation will become a part of the teaching content of the MSS subject that is part of the CS teacher's study programme at our faculty.

Acknowledgement

The work has been supported by the Cultural and Educational Grant Agency of the Ministry of Education, Science, Research and Sport of the Slovak Republic (KEGA) and the contribution was elaborated as the part of the KEGA project KEGA 001UMB-4/2020 entitled *Implementation of Blended Learning into Preparation of Future Mathematics Teachers and Future Computer Science Teachers* and KEGA 003TTU-4/2018 entitled *Interactive Applications for Teaching Mathematics at Primary Schools.*

References

[1] Smith, David – Moore, Lang. *"The SIR Model for Spread of Disease – The Differential Equation Model."* In Convergence, 2004. Available at: ⟨https://www.maa.org/press/periodicals/loci/joma/the-sir-model-for-spread-of-disease-the-differential-equation--model⟩. Last accessed: 2020-09-15.

[2] *"SIR and SIRS models."* Generic Model documentation. Bill & Melinda Gates Foundation. Available at: ⟨https://idmod.org/docs/emod/generic/model-sir.html⟩. Last accessed: 2020-09-15.

[3] Weisstein, Eric W. *"SIR Model."* From MathWorld – A Wolfram Web Resource. Available at: ⟨https://mathworld.wolfram.com/SIR-Model.html⟩. Last accessed: 2020-09-15.

[4] Sanderson, Grant. *"Simulating an epidemic."* At YouTube, 3Blue1Brown, 27. 3. 2020. Available at: ⟨https://youtu.be/gxAaO2rsdIs⟩. Last accessed: 2020-09-15.

[5] Kölling, Michael. *"Java programming with Greenfoot (a simulation of the spread of a virus in a population series)."* At YouTube, Channel Greenfoot, King's College London, 18. 4. 2020. Available at: ⟨https://www.youtube.com/playlist?list=PLIWb-FtdAhJjNgdIljwkMEQMhwhVIiV63⟩. Last accessed: 2020-09-15.

[6] *"SEIR and SEIRS models."* Generic Model documentation. Bill & Melinda Gates Foundation. Available at: ⟨https://idmod.org/docs/emod/generic/model-seir.html⟩. Last accessed: 2020-09-16.

[7] Horváth, Roman. *"Vírus."* Trnava : Faculty of Education of Trnava University in Trnava, 5. 4. 2020. Available at: ⟨https://pdf.truni.sk/horvath/materialy?virus⟩. Last accessed: 2020-09-15.

[8] Horváth, Roman. *"Dokumentácia programovacieho rámca GRobot."* Trnava : Faculty of Education of Trnava University in Trnava, 2020. Available at: ⟨https://pdf.truni.sk/horvath/GRobot/⟩. Last accessed: 2020-09-17.

[9] Horváth, Roman. *"Programming framework GRobot."* At GitHub, 2020. Available at: ⟨https://github.com/raubirius/GRobot⟩. Last accessed: 2020-09-17.

[10] Horváth, Roman. "The Past Seven Years of Development of the Framework for Teaching Programming and the Students' Results." In *ICETA 2018.* Danvers : IEEE, 2018. ISBN 978-1-5386-7912-8, pp. 185–189.

[11] Horváth, Roman – Javorský, Stanislav. "New Teaching Model for Java Programming Subjects." In *Procedia – Social and Behavioral Sciences.* ISSN 1877-0428. No. 116(2014), pp. 5188–5193.

[12] Stoffová, Veronika – Horváth, Roman. "Didactic computer games in teaching and learning process." In *eLSE 2017.* Bucharest : Carol

978-0-7381-2367-7/20/$31.00 ©2020 IEEE

I National Defence University Publishing House, 2017. ISSN 2066--026X, pp. 310–319.

[13] Tromey, Tom – Blake, Eric – Gilbert, David. *"Source for java.awt.geom.Line2D."* GNU Classpath 0.95 Documentation. 2000, 2001, 2002. Available at: ⟨http://developer.classpath.org/doc/java/awt/geom/Line2D-source.html⟩. Last accessed: 2020-09-15.

[14] Graham, Jim. *"Java example source code file (Line2D.java)."* From Java examples. Available at: ⟨https://alvinalexander.com/java/jwarehouse/openjdk-8/jdk/src/share/classes/java/awt/geom/Line2D.java.shtml⟩. Last accessed: 2020-09-15.

[15] Kishenko, Denis M. *"awt/java/awt/geom/Line2D.java – platform/frameworks/native."* From Git at Google, The Android Open Source Project, 2008. Available at: ⟨https://android.googlesource.com/platform/frameworks/native/+/e09fd9e/awt/java/awt/geom/Line2D.java⟩. Last accessed: 2020-09-15.

[16] *"Shapiro–Wilk Calculator."* Statistics Kingdom. Available at: ⟨https://www.statskingdom.com/320ShapiroWilk.html⟩. Last accessed: 2020-09-24.

[17] Horváth, Roman. *"Raw listing of the GRobot's knižnica/Svet.java file."* At GitHub, 2020. Available at: ⟨https://raw.githubusercontent.com/raubirius/GRobot/master/kni%C5%BEnica/Svet.java⟩. Last accessed: 2020-09-17.

[18] Lile, Samantha. *"44 Types of Graphs and How to Choose the Best One for Your Data."* From Visime.co, 2017. Available at: ⟨https://visme.co/blog/types-of-graphs/⟩. Last accessed: 2020-09-18.

978-0-7381-2367-7/20/$31.00 ©2020 IEEE