MAIG Group Project

Claus B. Wind
M.Sc. Game Technology
IT University of Copenhagen
Copenhagen, Denmark
Email: clbw@itu.dk

Mats Stenhaug
M.Sc. Game Technology
IT University of Copenhagen
Copenhagen, Denmark
Email: msten@itu.dk

1. Introduction and problem statement

We wanted to create a simulation of human life in a village. Having the people that ”live” there meet each other, find love, and have kids so that the village population would increase.
To fight the people, we wanted to create diseases that would damage and try to kill the whole population of people if possible.
However, in order to do so, the diseases need to evolve, and that means that we have to have an evolutionary algorithm.

Therefore, our problem statement goes like this:

> Explore competitive co-evolution in the setting of a virtual village, where people and diseases evolve to battle each other.

The idea was to make each generation of new people, that were born. Evolve allowing for them to be a ”mutation” of with the genes of both parents.
As for the diseases, we wanted to have them kill as many people as possible, so evolving the lethality, spreading rate and the resistance rate would be crucial.

The village life simulator would be an interesting project to work with, as we will hopefully see how two genetic/evolutionary algorithms work against each other in real time. Who will win? People or Disease?

In this document we will elaborate and discuss on the implementations we made, although, we weren't able to implement everything we had wished to. This will be elaborated and reviewed in the discussion and conclusion section.

2. Background

To the best of our knowledge, there has not been any project quite like this. Our goal was to create a village life simulator that took as much information as possible into account, but whether we are able to include it all in this short amount of time, remains to be seen. There are models and texts on epidemics and diseases, which could possibly be useful to our project [1] [2]. As to which approach we will take to integrate it into our evolution, remains to be explored. Can we better our epidemics simply by using an evolutionary approach?

The interesting concept of this project, is that we will have two or more genetic algorithms fight each other for domination. Either the village/population will be extinct, or it will overcome diseases and thrive.

There has been much research on Genetic Algorithm and evolutionary algorithms in general [3], but using them to fight each other for domination, is something we have yet to see, and is why we thought it could be very interesting to see if we are able to implement it into our solution.
One interesting paper we found, was about competitive co-evolution [4], where some key ideas were described as to how we could possibly make both parties, people and diseases, evolve together and fight each other during evolving as well as in actual run-time. Sharing information and states of the simulation would be a key component here, as well as having some sort of fitness sharing.

During our time of this project, our main focus would be on evolving the diseases, and then if we have the resources and time, connect the two parties into making some sort of competitive evolution, if not competitive ”co”-evolution.

3. Game mechanics

Technically we are not making a game, but more of a simulation. If we have enough time to implement the algorithms and make all the connections, we will try adding a UI to the system.
By having this UI, we can see if we can find some way for the user to act on behalf of the humans, or act as the creator of different diseases that can mutate and change.

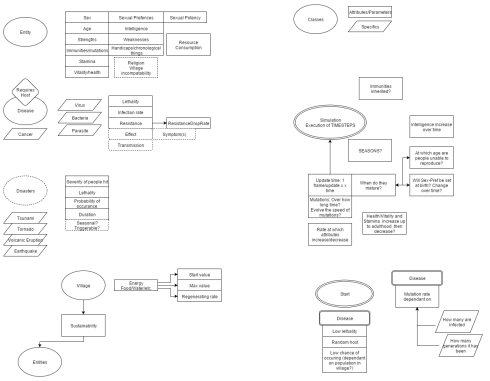

Figure 1. Initial brainstorm of the features we would like to implement.

The AI we are implementing is essential, as without it, it will not be a simulation of evolution, but merely an imagined hard-coded painting. We want our ”game” to come to life, change and mutate. Ideally in real-time. So by having our simulation using Genetic Algorithms, we can make the fight between people and sickness come alive.

Our original thought was to have some sort of competitive co-evolution [5].
We wanted to have the inhabitants of the village evolve at the same time as the diseases were evolving. Having both elements evolve at the same time, would mean that we would have to simulate the next iterations of the game state and share fitness for the evolving parties as well as sharing the state itself. Sharing this information between the parties, would allow them to competitively evolve based on the other party’s evolution. In our case this would mean that if a disease would evolve to a high-lethality/low-spread disease, our people would then maybe want to evolve into something that had more resistance to being infected. This would counter the effect of the disease, and the people would hopefully make it. But then the disease would want to change up again, so that it was sure to spread more, and get the better fitness.
As this process would go on, both parties would competitively fight each other in order to try to evolve into the better fitting product.

Since we weren’t able to implement this competitive co-evolution in the time-frame we had, we had to turn to a more basic approach, which was to have simpler, yet highly functional, genetic algorithm for just the diseases. As you will see in the results and in the discussion section of this paper, the behaviour of the genetic algorithm turned out to be quite interesting.

4. Methods

4.1. Genetic Algorithm Evolution of the diseases

4.1.1. The Genome. For initiation, we start by having the generator create X genomes of a disease, where we randomly create the gene with chromosomes representing its lethality, spread rate and what we call ”resistance drop rate” [1]. These chromosomes are then made by using formulas based on the lethality.
For example, the spread rate of a disease is directly based on the lethality as in: 100% - lethality = chance of spread.

4.1.2. The Evolution. After the initial 5 genomes of diseases are made, the algorithm starts creating mutations of them, and keeps testing these until it has made as many mutations as we want it to. In our case, we set this number to be 40 (the population size).
Based on the initial value for the lethality on each of the genomes, the algorithm then mutates that value either +1 or -1 for each of the genomes. These mutations will then be made into the children of the original genome, and will be tested in the simulation along with all the other genomes in the population.

4.1.3. The Simulation. After the 40 mutated and evolved genomes are created, we simulate the next 10 iteration steps for each of them, and gets returned a fitness value for each of them, based on how much damage was dealt, and how many people was killed[2].
The simulation itself is made by using a copy of the current world state, and running a continuation of that for each of the potential diseases.
After each genome has been simulated in the game environment, the next step is to select the genome with the highest scoring fitness.
The algorithm will then proceed creating a new generation based on the current utilizing the selection function below. It will continue looping between simulation, selection and deciding the gene with the best fitness for the predetermined number of generations, in our case 5. Ultimately, creating a disease object out of the best from the pool of best genes from every generation, and send it into the actual game environment.

4.1.4. Roulette selection on the Genetic Algorithm mutation. We originally had our genetic algorithm set up so that when two diseases would create an offspring that would become the new ”improved” disease, it would go

1. The ”resistance drop rate” in the current implementation is the rate at which people grow immune to a disease, but in terms of immunology it can be considered how easily the body can generate anti-bodies (very simplified interpretation of how immunology works [6]).

2. Fitness score calculated as follows: 300 for kills, 1 per damage dealt to the population.

through our population list of current possibilities, and just match up two at a time in a manner of taking the first two, pairing them up, creating offspring, and then go on to the next two in the list, and so on.
However this turned out not to be such a good way of doing it, since the list was already sorted by the fitness of each disease. Which means that the first two would have the highest fitness, and then the next two would have lower, and so on.
By implementing the roulette selection, we could have randomly selected diseases, that did not necessarily take too much thought into the fitness.

How it works[3]:
It generates the fitness for each disease like before, but it also has an average fitness.
Then we create a random percentage based on said average, and use that to match up against the next disease. If that other disease then would match the criteria, it would be selected.
After successfully implementing this algorithm into our Genetic Algorithm, we saw that the diseases that came out had more of a ”smart” approach to them.
For instance, the first disease would weaken the population, and then the second one would come in and deal a lot of damage. And if that wasn't enough to kill all the people, it would create another disease that would make sure this happened.

4.2. The People

Each person has individual stats and variables.
Every one of those stats have some sort of meaning, for example: age, health, gender and sexual preference.
A person can only get pregnant if it's female, and only by having sex with a man that is capable of having kids.
Basically we have tried to make all the variables and stats abstractly represent features in reality, though many more variables and conditions would be required to accurately represent a human being.

4.2.1. Simulating the village. The algorithm initializes an initial pool of 50 people, with ages ranging from 15-25, male or female.
Then the people ”walk around” and meet other people.
If two people meet, like each other, and are of opposite gender from one another, then chances are they will mate and make a child, which is the only way of increasing the population of our simulated village.
After people have interacted with each other, the simulation goes over to the disease(s) and if possible, introduces a new disease to the world. In the beginning of the game simulation, we have made a sanctuary period of 10 years/iterations, which allows the villagers to mature, interact, and try to increase the village's population. After this period ends, the diseases will start coming.
A disease is being made based on the chance of it appearing. The chance is set so that if there aren't any diseases, it will automatically create one the first time around. After that, however, the chance for a disease to appear, is based on a couple of key factors:

- How long it has been since a disease was introduced. (the longer, the higher the chance.)
- How many people are in the village. (If the population of the village is bigger, the chance for a new disease spawning is greater.)

When a disease has been introduced, that disease infects as many people as it can, based on its spreading rate, and after that, it will deal damage to them according to it's lethality and the infected people's resistance to said disease. If however the disease has killed all its hosts, or they have all grown immune to it (and there are no other diseases present), the people are rewarded a new sanctuary period, allowing them to recover [4].
When the diseases have had their turn, the next step of the simulation is to damage the infected people. This is simply done by going through all the people in the village, checking if they are infected, and then damaging them based on the lethality of the disease they are infected by, and by the individual's resistance to said disease.
After all the damage has been dealt, the next step is to clean-up and update the people that means removing those who died, giving birth to the babies (1 iteration is 1 year), and removing old people who no longer have any contribution to the village/simulation [5].
After all that is done, the cycle returns back to the top, and a new iteration is started.

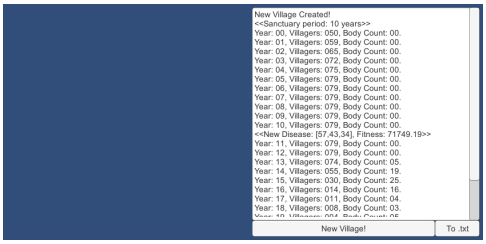

Figure 2. Snapshot from the current user interface made with Unity 5.2.

4.3. Interface

The ”game” is a real-time simulation and our user interface is quite minimal, it can be seen in Figure 2.

3. Implementation inspired from [7]. While we had not dug too much into this algorithm in the present state of the program, we found some interesting article [8] taking this selection further.

4. Diseases will however have a chance appearing after the same progression of time from a sanctuary period from another disease, if the other disease has not yet been gotten ridden off.

5. Currently this happens when they reach age 42 (sorry).

We have a text panel with a scrollbar, which allows the user to see the state of the simulation for each iteration every simulated ”year”. Along with this, there are also two buttons. The first one is to start a new simulation, which clears the text-box (which would otherwise contain all the data from the previous simulation), and starts printing out new data from the current, newly started, simulation. The second button allows the user to save the data to a text file. A new text file is created with every click, and the name of the file is simply a formatted pattern of the current date and time-stamp.

Having this interface, allowed us to do quicker tests, and to easily scroll through the output data and statistics from the simulation, as well as allowing us to quickly start a new simulation without having to restart the application. This is however nowhere near the initially envisioned GUI/game we had in mind when we started working this project.

4.4. Visualized Data

As described in the segment above, where we go into our simple interface, we print out a lot of data. The purpose of this is to be able to see how the genetic algorithm is working, and what impact the finished result have on the ”village” we are simulating.
During our development process, we changed what we displayed multiple times, as at different times, we wanted to examine different sets of data.
The data we were most interested in visualizing were as follows:

- Iteration number (which year we currently are in)
- Phenotype of the Diseases created by the Genetic Algorithm
- Amount of people that are alive in the village
- Amount of people that are infected by the disease
- Amount of people killed by the disease
- Amount of new people being born each iteration
- Total Kills and Deaths
- Amount of people that died from old age (not by the diseases)
- Total damage done by the disease

These are some of the most commonly used data that we visualized for both debugging purposes, as well as for result gathering.

5. Results

5.1. Experimenting and thoughts about the current implementation

Unfortunately, we were unable to implement all the features we initially wanted, as shown in Figure 1, we had a lot of ideas that we wanted to implement somehow. Although we were unable to complete all the tasks we wanted, we were able to make the most important part work; the genetic algorithm for mutating and creating diseases.

50/500 with 5 iterations

Lifespan	Disease	Fitness
09	1	71988.16
13	6	40787.93
09	2	42285.89
10	2	76082.22
10	4	30651.78
12	3	30272.74
09	2	35273.95
10	3	29201.43
12	3	24900.14
10	3	21598.59
10.4	2.9	40304.283

100/500 with 5 iterations

Lifespan	Disease	Fitness
09	1	144157.30
10	4	43924.62
13	3	97805.93
12	3	136139.83
10	2	161155.45
10	2	164141.45
12	2	154609.30
10	2	119703.96
09	3	195418.87
10	1	156274.20
10.5	2.3	265881.260

50/500 with 10 iterations

Lifespan	Disease	Fitness
13	1	90810.27
14	3	49816.86
13	3	41797.22
19	4	29914.41
14	3	29745.64
12	2	63100.59
14	1	75508.05
13	1	61623.95
13	3	29538.19
14	5	32699.82
13.9	2.6	50445.500

100/500 with 10 iterations

Lifespan	Disease	Fitness
14	3	61018.53
14	4	55524.38
13	1	232163.30
11	2	177002.15
14	2	109896.68
14	1	185291.80
12	2	156898.07
14	2	76036.45
15	3	173425.03
15	3	61841.79
13.6	2.3	128909.818

50/500 with 15 iterations

Lifespan	Disease	Fitness
14	5	103328.92
17	2	82210.36
13	1	81541.56
13	1	101505.90
13	2	68591.16
15	2	42628.48
12	1	97943.52
13	1	86292.06
15	2	46835.93
26	6	215150.50
15.1	2.3	92602.839

100/500 with 15 iterations

Lifespan	Disease	Fitness
15	2	41585.33
11	2	104729.45
12	3	78307.07
14	2	76492.84
12	3	68966.57
17	4	128281.88
12	4	111038.49
11	5	52633.92
13	2	102228.71
12	3	133362.31
12.9	3.0	89762.657

TABLE 1. VILLAGE LIFE SIMULATIONS WITH 10 TRIALS AGAINST GA-OPTIMIZED DISEASES.

As mentioned earlier in this paper, the mutation of the diseases were originally created using a ”straight forward” type of selection.
At a later time, though, we switched to using a roulette selection in order to get a more pool of results. As shown in Table 1: the calculated average fitness of the diseases made with roulette selection is 111317.726, whilst in Table 2, you see that using the non-roulette selection, the average fitness was 71810.294.
Having the roulette selection implemented also allowed us to see some more interesting behaviour from the diseases. For instance, we saw that the diseases usually behaved in a pattern.
That pattern was to either weaken the population first, and

then kill them off with a high lethality disease afterwards, or the opposite, where it killed off as many as possible first, in order to make them unable to reproduce and then finish them off.

It was quite interesting to see this sort of behaviour occur, just from implementing a new way of selecting which genomes that should be paired up to make a new child.
Before we used the roulette selection, the results were more random and usually just killed off as many people as possible, or infected as many as possible. It was usually one or the other. There were no proper connection between the population and the type of disease that was created.

5.2. Roulette Selection

Non-roulette: Always creating diseases from the highest fitness, so that there will not really be any variation. It is either a very high fitness, or a very low. Which means, either a High-Lethality/Low-Spread (HL/LS) disease, or a Low-Lethality/High-Spread (LL/HS) disease.
Some phenotype examples for this version are:
High Leth/Low Spread:
G[92, 8, 62] – G[Lethatlity, Spread, DropRate]
Low Leth/High Spread:
G[3, 97, 15] – G[Lethatlity, Spread, DropRate]

Roulette: Generating all sorts of variations from HL/LS diseases to LL/HS diseases, and everything in between; Medium/medium, med-high/med-low, etc.
Phenotype examples for the roulette selection, excluding HL/LS and LL/HS are:
Medium Lethality/Medium Spread:
G[50, 50, 42] – G[Lethatlity, Spread, DropRate]
Med-high Leth/Med-low Spread:
G[64, 36, 29] – G[Lethatlity, Spread, DropRate]

5.3. Simulation and speed of the program

The simulation will take longer (especially when creating new diseases using the GA), the more people are in the village. The time the genetic algorithm uses is also heavily based on how many simulation steps used for simulations that is running through for every potential diseases tested. Our entire system is closely connected, whether it be the amount of people, the number of iterations, number of genomes in the population, or a mixture of all of the above.

6. Conclusion

This project has shown us that evolutionary algorithms is capable of inspiring interesting results. Had we been able to implement two competing ones, it could make for some very interesting results. Even though we were unable to implement this competitive co-evolution, we did implement a very successful and interesting evolutionary algorithm for the diseases.
The fact that we could clearly see differences between behaviours when switching from non-roulette to roulette selection, made the whole concept of having two algorithms fight each other even more interesting. Imagine having two sorts of behaviours constantly fighting each other in order to maintain control?

Seeing how genetic mutation of a virtual disease can change the outcome of a village's population was quite fascinating, and we spent a lot of time changing around variables and testing against different parameters.
As shown in Tables 1 and 2, we tried with a lot of different variations. Doing this we found that even though one variation was not distinctly "better" than the other (perhaps

Default settings

Lifespan	Disease	Fitness
13	1	90810.27
14	3	49816.86
13	3	41797.22
19	4	29914.41
14	3	29745.64
12	2	63100.59
14	1	75508.05
13	1	61623.95
13	3	29538.19
14	5	32699.82
13.9	2.6	50445.500

People regeneration disabled

Lifespan	Disease	Fitness
13	1	100051.10
13	5	28284.08
14	1	80936.50
14	1	60818.35
13	1	65849.00
13	2	40888.69
14	1	96665.30
15	2	50637.58
14	1	78020.98
15	2	31386.04
13.8	1.7	63353.762

5 Simulation steps

Lifespan	Disease	Fitness
13	2	35785.54
13	1	61838.00
14	3	22259.17
13	1	57306.85
11	3	55459.15
13	2	35451.54
15	1	40386.61
14	2	23676.17
15	6	67234.64
12	2	81024.60
13.3	2.3	48042.227

15 Simulation steps

Lifespan	Disease	Fitness
13	3	26687.39
15	3	34510.81
14	3	38854.18
14	3	29499.69
26	10	239797.62
14	3	29140.77
11	5	130825.77
16	2	36893.06
14	1	80887.41
15	2	28564.16
15.2	3.5	67566.086

GA Population 100

Lifespan	Disease	Fitness
14	3	32777.65
13	1	79797.64
14	1	85458.33
13	2	37114.22
14	1	73605.61
14	1	77875.90
14	3	30455.83
15	4	36588.93
11	4	48179.46
13	2	48650.16
13.5	2.2	55050.373

Not Roulette Selection

Lifespan	Disease	Fitness
15	2	45112.97
14	1	84312.51
13	1	94652.02
14	1	95293.23
15	2	34246.59
13	2	40564.57
14	2	54092.32
14	1	108408.10
15	2	53359.63
15	1	108061.00
14.2	1.5	71810.294

TABLE 2. VILLAGE LIFE SIMULATIONS (START POPULATION 50, 10 SAFETY-ITERATIONS) WITH 10 TRIALS USING DIFFERENT SETTINGS. *(simulation steps: 10, population size 40)*

because of the randomness of it all), there were more behaviours occurring when we introduced the roulette selection.

So even though we were unable to implement all the functionalities, stats and behaviours that we wanted, we still leave this project with a sense of accomplishment. Our genetic algorithm and evolution works, and we are now capable of creating ”virtual” diseases that can reduce a village’s population to 0 within a 100 simulated years!

7. Discussion

For further development of this project we would like to implement more real-looking stats and connections between human beings and diseases. Add disasters like earthquakes and whatnot. Make the algorithms work better. Fix the fitness functions. Make more diseases. Have GA on the humans as well, to make the ”Fighting” aspect really come to life. Etc.

Passing on of immunities from parents:. Having some sort of algorithmic calculation of the chance for newborns to inherit immunities from the parents, as well as to how strong an immunity said newborn would get [6].

Genetic algorithm for the people:. Make the inhabitants of the village be created with some sort of genetic algorithm so that they will take the existing diseases into account, and try to build their stats based on how dangerous it is to ”live” and also how many people are in the village.
If there were a lot of people (closing up to the maximum supported amount in the village) the chances for a new disease to occur would be greater, and therefore the new people might want to have higher resistances to disease, but lower health or something along those lines.

Better and more realistic numbers and calculation of the fitness:. This would apply both for the Genetic Algorithm of the diseases as well as the one for the people. Had we had more time, and had been able to implement the genetic algorithm for the people, we would have loved to have better fitness functions for both, and to make them as realistic and good as we possibly could manage.

Natural disasters:. Have some sort of system that would sometimes make a natural disaster occur, like a tornado coming in, or an earthquake.
The thought was that the more people in the village, the more people would be affected as the ”space” where you could ”hide” (for example) would be significantly smaller.

Fighting between people:. When two people would meet, we would have liked to have another system in place that would be able to make conflicts between them occur; say if two males meet, and they both fancy the same woman, they might fight (very primal, yes) to see who would eventually go on to ”win” the rights to be with her.

Roles, tasks and resources:. Tasks and roles for the villagers, allowing them to do more than simply breed like rabbits for a whole year, say the village might have some people to gather food or medical herbs that would allow them to withstand diseases and disasters better.

References

[1] W. O. Kermack and A. G. McKendrick, "A contribution to the mathematical theory of epidemics," in *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, vol. 115, no. 772. The Royal Society, 1927, pp. 700–721.

[2] E. Ericson, T. Ericson, and J. K. Mller, *Klinisk mikrobiologi : infektioner, immunologi, sygehushygiejne*. Kbh: Gad, 2010.

[3] D. E. Goldberg and J. H. Holland, "Genetic algorithms and machine learning," *Machine learning*, vol. 3, no. 2, pp. 95–99, 1988.

[4] C. D. Rosin and R. K. Belew, "New methods for competitive coevolution," *Evolutionary Computation*, vol. 5, no. 1, pp. 1–29, 1997.

[5] ——, "Methods for competitive co-evolution: Finding opponents worth beating." in *ICGA*. Citeseer, 1995, pp. 373–381.

[6] (2015) Den Store Danske immunolog. [Online]. Available: http://www.denstoredanske.dk/Krop,_psyke_og_sundhed/Sundhedsvidenskab/Immunologi/immunologi

[7] (2012) Roulette Selection implementation. [Online]. Available: http://stackoverflow.com/a/10949834

[8] A. Słowik and M. Białko, "Modified version of roulette selection for evolution algorithms–the fan selection," in *Artificial Intelligence and Soft Computing-ICAISC 2004*. Springer, 2004, pp. 474–479.

[9] J. H. Holland, *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

[10] (2015) Wikipedia crossover. [Online]. Available: https://en.wikipedia.org/wiki/Crossover_(genetic_algorithm)

Acknowledgments

We would like to thank Marco Scirea for supervising us during this project and for suggesting roulette selection. Thank you for the feedback.