

Viral infection genetic algorithm with dynamic infectability for pathfinding in a tower defense game

Gabriel Teixeira Galam
 UBI (Universidade da Beira Interior)
 Covilha, Portugal
 gabriel.galam@ubi.pt

Tiago P. Remedio, Mauricio A. Dias
 Game Research Academy (GRAFHO)
 Computer Engineering, Engineering Department
 Herminio Ometto Foundation - FHO
 Araras - SP, Brazil
 {remedio,macdias}@fho.edu.br

Abstract—Hardware designed for gaming computers and platforms is achieving impressive results on different benchmarks suggesting that the response time of the algorithms have to be improved in order to achieve current real-time requirements. Also important is to consider to maintain the high standard of algorithms results. Pathfinding is an important and computationally complex feature of Non-Player Characters in several video game genres and is still being solved with well-known A* and Dijkstra algorithms or variations. The problem is that both techniques depend on specific design choices as using a graph as a map for Dijkstra or static environments for the classic A*. Due to good results achieved using these classical solutions researchers are inclined to include these algorithms in their proposed variations even when a dynamic environment is considered. As a solution this work presents a viral infection genetic algorithm for pathfinding that is able to provide new paths for dynamic changes in the environment and also considering the final amount of the NPC life. Results showed that the proposed algorithm is able to work in real-time obtaining acceptable paths as results.

Keywords—pathfinding; genetic algorithm; viral infection; NPC; games; artificial intelligence;

I. INTRODUCTION

The beginning of video-games can be pointed as the appearance of *Tennis for Two* in 1958 [35]. Since the evolution of this area resulted in a huge industry, game hardware and software designers try to improve their results in order to meet players expectations. Among the many factors to be optimized in the development of games it is possible to point out graphics hardware and processing power as key factors that are correlated in many different ways.

Graphics cards are an important feature for any game platform and based on their processing power the entire platform can be classified as suitable or not for a certain game to run on minimal requirements. Several benchmarks are used to rank these video cards and last results of real-work game benchmarks showed that the frame rate at 1080p (resolution used on the benchmarks) is at 163.9 *Frames per Second* (FPS)¹. This rate needs a response time of less than

0.006s of algorithms that are controlling everything else in the game but the players movements.

Processing power is also being increased and the final results showed that processor's hardware is not a problem. Desktops are achieving better performances with AMD and Intel multicore processors that are up to 18 cores able to manage 36 threads². This evolution is also affecting mobile processors up to 8 heterogeneous cores that are able to work with different workloads to use power in a more intelligent way^{3,4}.

Graphic cards and processors seem to be able to deliver the processing power needed by game designers. The question now is why are games still having trouble with these features? One part of the problem is the constant need for more realistic graphics that results in games with millions of polygons to be rendered in each character and, consequently, more processing power is needed. This demand is a result of the massive production of games in which the only appeal is the graphics quality. As a result the other part of the problem is that processors have to deal with a huge amount of computer graphics routines and the game control algorithms which overloads the processor.

One of the main problems in game control are the Artificial Intelligence (AI) algorithms that controls Non-Player Characters (NPCs). These algorithms are designed in many different ways to solve several problems. One of the most famous games that has an AI control for NPCs was Pac-Man [21], that used a State-Machine (SM) with three states: chasing, scattering and frightened. Until mid-1990's, all NPCs were sophisticated Pac-Man ghosts since there was a SM that controlled their behaviour and became considerably predictable after some time [21]. After the concepts of Sense Simulation, decision trees, goal oriented action planning and other techniques were designed, they led to better NPCs. AI control algorithms have to deal with a common problem for different types of games that is the

¹gpucheck.com/gpu-benchmark-graphics-card-comparison-chart

²<https://www.pcmag.com/roundup/366303/the-best-cpus>

³shorturl.at/moZ05

⁴<https://www.apple.com/iphone-xs/a12-bionic/>

pathfinding [21]. Some NPCs have to move around the world (using predefined actions or random walks) and doing it using fixed routines is easy and useless. More complex NPCs will have to go from one point of the map to another and calculate a suitable route according to the level of the game. Figure 1 shows that pathfinding is an important step between any decision making regarding movement and the movement itself.

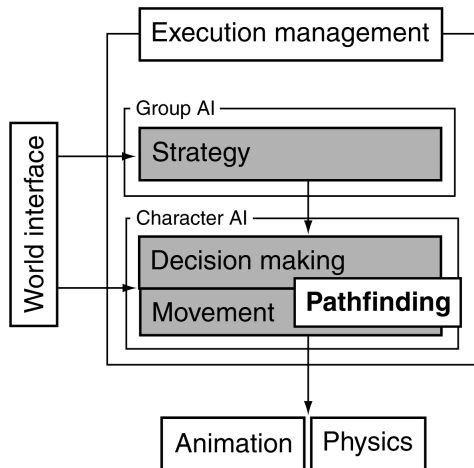


Figure 1. AI model proposed in [21] where pathfinding is between decision making and movement of characters.

Considering the described scenario, this work proposes a viral infection genetic algorithm for NPC pathfinding. The idea is to provide a solution for the problem in dynamic environments without the dependency of map representation or an heuristic function, problems that are a part of the most used techniques (Dijkstra and A*) and their proposed variations. This work does not compare its results with Dijkstra and A*, but proposes a solution for its problems. Proposed solution achieved the best solution in acceptable execution time suggesting that it is a good solution for the problem.

In section II the theory about A* and Dijkstra algorithms are presented in order to allow a better understanding of the problem together with the correct analysis of this work's results. Section III presents the related works and their contribution to this paper. Section IV presents the algorithm and the scenarios used for performance evaluation. The results of the proposed work are shown in Section V. Conclusion is presented in section VI followed by the bibliographic references.

II. BACKGROUND THEORY

This section will present an extensive analysis of different solutions for pathfinding problems with focus on game design. However, some initial background is needed for correct understanding the problem and the analysis of related works. A simple superficial search about pathfinding can

show that the most used algorithms to solve this problem are A* and Dijkstra, with custom variations. These two algorithms have their limitations and their variations are intended to solve the problems but the main problem is not completely solved due to the fact that motion planning problems are NP-Complete [19].

Pathfinding problem can be defined as follows: considering an initial configuration of rectangular objects in a rectangular two-dimensional box the problem is to determine whether there is a continuous coordinated motion between the initial and final configurations without interceptions [12].

This problem is proved by Hopcroft et al. [12] to be NP-Hard by reductions: to a formal string manipulation problem that represents the objects movements by changing positions of letters in a string (a combinatorial problem), then to a motion problem of irregular body shapes and finally to a simpler rectangular motion problem. The idea of the first reduction is explored by Reif [25].

As a NP-Hard problem, the use of heuristics to solve it is indicated, but also search algorithms are an interesting choice. One of the largely used search algorithm to solve the pathfinding problem is Dijkstra with time complexity of $O(|V|^2)$ in worst-case scenario [4], where V is the number of vertices in the graph that represents the map. A* algorithm time complexity depends directly on the heuristic function and in the worst-case scenario the complexity of the algorithm is $O(b^d)$, where d is the depth of the solution tree and b is the branching factor (that is the average number of successors per state) [26]. Most used algorithms time complexity suggests that they should be modified or replaced by more efficient solutions, but to understand these algorithms, an analysis of their behaviour and complexity is required.

Dijkstra is an algorithm that searches for the shortest path between nodes in a graph. In its classical definition it considers only an initial and a final node but in the most known form of the algorithm, it starts at a *source* node and builds the shortest-path tree to all other nodes in the graph [4]. Basically the algorithm uses a greedy strategy of choosing the shortest path to each other non-visited achievable node. Nodes are visited for this procedure only once. The problem is that for each new visited node all costs of the possible paths are recalculated and the result changes if there is a path that is shorter than the previously chosen one.

This algorithm have a high execution time for relatively small graph sizes for games (with more than 10.000 nodes in a top processor it became seriously critical) and another serious problem is that the environment have to be static. A simple change on the distance values during the algorithm execution is enough to invalidate the entire shortest-path tree and the algorithm has to be executed again from scratch. Another problem in Dijkstra's algorithm is that edges cannot have negative costs.

Dijkstra's time complexity of $O(|V|^2)$ is the result of visiting all unvisited nodes from a *source* node in the worst-case scenario, even if there is no need to calculate all short paths between every node. This unwanted cost is called the *fill* of the algorithm. A solution for this high-fill problem is the A* algorithm that can be considered a low-fill version of Dijkstra [21].

A* algorithm is the most popular choice for pathfinding because of its flexibility. The success of the algorithm is the intelligent combination of information about favoring vertices that are close to the starting point (same as Dijkstra) and favoring vertices that are close to the goal (same as greedy best-first-search)⁵. A* algorithm uses a function to calculate the next move (equation 1) where $g(n)$ represents the exact cost from the source to node n and $h(n)$ is the heuristic estimated cost from node n to the target node [11].

$$f(n) = g(n) + h(n) \quad (1)$$

It's really important to choose a good heuristic because it should estimate correctly the cost to the goal node and can be used to control A* algorithm's behaviour. The relation between the heuristic function and A* algorithm results can be summarized as follows⁵:

- if $h(n)$ is 0 then A* is turned into Dijkstra's algorithm.
- if $h(n)$ is always lower or equal to the cost of moving from n to the goal, then A* is guaranteed to find a shortest path. This is usually a result of a large expansion causing A* to be slower.
- if $h(n)$ is exactly equal to the cost of moving from n to the goal, then A* will be trapped in the best path without expanding anything else. This can be a problem in some situations.
- if $h(n)$ is a little higher than the cost of moving from n to the goal, then the A* is not guaranteed to find the shortest path but can run faster.
- if $h(n)$ is higher than the cost of moving from n to the goal, then A* is turned into a Greedy Best-First-Search.

This analysis of A* behaviour shows how the algorithm's result depends on a good heuristic. It is a serious problem because it is not easy to find a good function for every situation. Also A* is not able to deal with dynamic environments considering the iterative nature of the algorithm.

III. RELATED WORKS

Pathfinding algorithms in games have to deal with dynamic environments and achieve execution time requirements and this has been possible with the use of modified versions of the presented algorithms. Some of these versions are presented and discussed next.

This state-of-the-art review will cover three different groups of research works. First the most used variations

of A* algorithm are presented and discussed, after that, pathfinding related recent works are analyzed followed by works on pathfinding presented in previous SBGames editions.

A* variations are suitable for different kinds of problems. One simple variation is based on *beam search* [26] that limits the size of the structure that stores nodes that may need to be searched to find a path. This algorithm is indicated when memory is a critical requirement and the results can be significantly affected.

Iterative Deepening is another A* variation that proposes a cut-off for nodes whose values of the function f are higher than a threshold [15]. This variation is not used as much in games because the value of the cost function for the nodes is relatively stable. Other alternative is to apply weights for the exact cost and for the heuristic cost, this will result in a faster search method that also affects the results [26].

A bandwidth heuristic is proposed by [10], where the search is a modification on the $h(n)$ part of equation 1 that is turned into $h'(n)$ according to equation 2 where e is an error and d is an estimator.

$$h(n) - d < h'(n) < h(n) + e \quad (2)$$

In practice there is now an upper bound and a lower bound for the heuristic function that can be guided to the result faster but more carefully than previously presented variations. Another possibility is the algorithm proposed in [31] called *bidirectional search*. The idea is simple, two searches are started at the same time, one from the start to finish and another from the finish to the start. When the two searches meet it indicates that a good path was probably found. Good results of this algorithm are presented by Chen et al. [3].

None of presented A* modifications are able to deal with dynamic environments. As pathfinding problem is also addressed in robotics, a solution for path planning in dynamic environments was proposed. D* algorithm [32] is a modification of A* algorithm that is able to redefine the path if an obstacle gets in the way of the planned path. If the value for the heuristic changes for a group of nodes these nodes are reinserted on RAISE and LOWER list to be reevaluated. The memory consumption of the algorithm to store all processed nodes is very high and its not indicated to NPC's control. Presented variations are used and modified in different situations for programming AI in games.

Fanton's work [6] presents a family for dynamic pathfinding with focus on quick generation of individual paths. The idea is to find midpoints that can reduce the final path cost and allow fast recalculation. Results presented in [6] showed that the proposed approach is very specific and fail to achieve the goal as a general purpose pathfinding algorithm.

⁵shorturl.at/cxDQY

Different game scenarios were compared to evaluate previous modifications of A* algorithm by Krishnaswamy [17] and the result showed that D* is more indicated to games with high number of obstacles (static or dynamic) but A* is faster. Games were also used to compare hierarchical versions of A* algorithm [33]. Hierarchy in pathfinding context is the idea of divide-and-conquer applied to a long path, considering as a single problem each sub-path contained in the original path. Algorithms are presented and compared by Vermette [33] but not analyzed. Problems were found in each presented algorithm and no solution has been provided.

Despite all efforts to modify A* and present a viable solution for real games, the pathfinding problem is still a challenge. Cui and Shi [5] presented a version of A* that considers exploring a NavMesh network that is better for the algorithms results since the paths are already predetermined but was not able to improve the results in real game scenarios due to the restrictions of A* algorithm and the heuristic function problem.

Multi-agent (MA) based solutions are also explored as presented by Sigurdson et al. [28] but this approach adds multi-agent control and coordination problem to pathfinding and solutions have to deal with both. A modular solution is proposed in [28] and the results are promising compared to other MA-based algorithms but still not solving the problem.

Adversarial environments are the most challenging problem in pathfinding because the solution should also consider a way to keep the unit far from enemies along the path. This problem was named SANE (SAfe Navigation in adversarial Environments) by Keidar [14] and different solutions were proposed improving state-of-the-art results without completely solving the problem due to specific features of each solution making them very specific. Good results were obtained by Amador and Gomes [1] using regions of positive and negative influence during path calculation. This technique showed to be promising but the problem is how to find or to consider influence regions in every game scenarios. Some regions will have to be created or defined depending on the situation and it is another problem to this solution.

A small number of research works presented on previous editions of SBGames addressed the pathfinding subject. 2007 edition had three works with titles related to pathfinding problem but the proceedings were not available for download. A modified A* [30] presented an algorithm designed for CUDA architecture to be executed in a GPU, but the problem is that proposed modifications were designed because of hardware limitations or features, and results showed that GPU is able to deal with a high number of NPCs without achieving the reference speedup.

A pathfinding solution, presented by Machado et al. [20] uses a simpler version of A* algorithm that calls a genetic algorithm to find a way out of an obstacle performing a local search. The main question is that the algorithm is called RTP-GA (Real-Time Pathfinding Genetic Algorithm) and the

only analyzed result was the ability of finding the exit of a maze without even considering the execution time. Also this simpler A* algorithm is not presented or analyzed and its an important feature of the proposed method. Results presented in [20] cannot be completely analyzed or even reproduced without this information.

The genetic algorithm proposed by Santos et al. [27] is based on patterns to improve pathfinding of a regular Best-First-Search algorithm. This concept is based on building blocks [8] that are the result of the search for patterns. Results showed a significant reduction on search effort for maps that have patterns. Mixed maps or maps without patterns showed that the algorithm proposed in [27] is indicated only for pattern-oriented maps.

A comparative analysis of pathfinding algorithms for mobile games was presented by Silva [29]. Results showed that A* algorithm outperforms the other algorithms in average execution time besides always finding the shortest path. Authors should also consider to compare A* variations because the algorithm is hardly used in its original form in games. The last pathfinding-related work presented in SBGames was the FPGA-based co-processor, by Nery [22]. Results of the co-processor suggests that a hardware implementation can be an interesting choice for execution time problems. The only problem is that in [22] authors focused on Xilinx High-Level Synthesis tool results, and this tool only works for Xilinx FPGAs, so proposed hardware structure as a result is useless if this hardware need to be implemented using an Intel FPGA.

Analyzed research works suggest that pathfinding problem is far from being completely solved. An extensive comparison of techniques [16] showed that better solutions are D*-Lite and Theta* for dynamic environments and A* for static environments (rarely found in games). New heuristics with better results and execution time are most welcome for AI game design, justifying this work's goal. Next section presents the proposed viral infection genetic algorithm for pathfinding.

IV. VIRAL INFECTION GA FOR PATHFINDING

This work proposes a Viral infection genetic algorithm with dynamic infectability for pathfinding. Algorithms presented in previous sections have problems that a Genetic Algorithm (GA) can work around. Instead of implementing the regular GA, authors decided to take advantage from building blocks [8] since pathfinding is a NP-Hard problem [12]. An unusual implementation of modified GA was used in this work. The standard GA is not able to take advantage from building blocks [2] because the individual (chromosome) is considered as a set of unrelated genes. One possibility of considering building blocks to achieve better results is the viral infection procedure. The idea is to implement a function that creates a virus (a building block) that will be

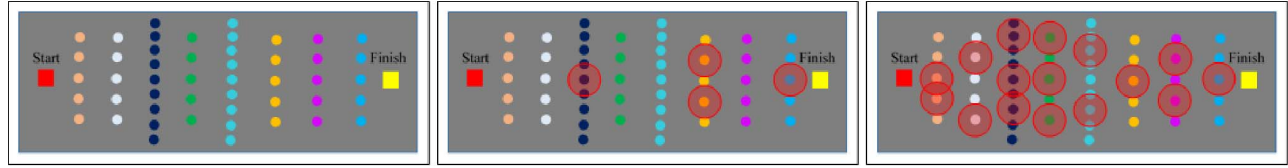


Figure 2. Layouts: the left layout is a base layout with no enemies, the center layout have a small number of enemies (represented by red circles) and the right layout used for a complex number of enemies experiment. These layouts were chosen considering an increasing complexity way in order to evaluate the algorithms in different situations.

propagated to the population individuals based on biological viruses behaviour.

There is an issue on how viruses theory and evolutionary theory should work together in computing. The work of Kubota et al. [18] is one of the first works to present the concept, proposing a virus-evolutionary genetic algorithm (VEGA). This approach is composed of a virus population and a host population, which coexists in time. In a specific generation the virus population and the host population propagate between each other. Regarding to all time, each population inherits genetic information from ancestors to offspring.

Other way to implement this behaviour is proposed by Franco [7]. The idea is to create a virus with an infectability rate. This rate should be controlled by the algorithm in order to propagate viruses that are generating better individuals and kill viruses that are generating bad individuals.

The use of genetic algorithms with virus-evolution has been seen in many different works to solve different problems, including the Traveling Salesman Problem (TSP) [9], logistics distribution [24], dynamic route planning for car navigation [13], classifier for pedestrian detection [23], and many others. Using this approach of pathfinding problem in games is the focus of this work.

The study uses three fixed layouts to test the proposed solution for the intelligent pathfinding problem in order to comparatively test the algorithm. For each layout four different versions were executed: the basic genetic algorithm, the basic genetic algorithm with a custom mutation, the basic genetic algorithm with the viral infection procedure, and, finally, the basic genetic algorithm with viral infection and custom mutation. For each of these four version, four empirically-based rate combinations were used: 60% crossover and 5% mutation, 60% crossover and 15% mutation, 80% crossover and 5% mutation and 80% crossover and 15% mutation.

Proposed layouts (Figure 2) used in tests try to simulate situations where a non-player character (agent) needs to reach a destination passing through a possible dangerous environment. A layout is composed by waypoints that should be reached by a NPC in order to cross the entire scenario. The first layout is a clean scenario, without enemies. The objective is to start in the red square and reach the yellow square (for future comparison of values, in the simulations

used the distance from start to end is exactly 29.41). A second scenario analyzed have an average number of enemies. This layout will make the agent try to avoid the red circles (which decrease the agents life). The last layout is used trying to simulate many enemies in the same environment. In this case the agent needs to try the best route and it is desired that the time to calculate the path is low enough that the player does not visualize a hiccup in the game (normally around 1-2 seconds).

A. Genetic Algorithm

A basic genetic algorithm was implemented using the idea presented by Goldberg [8]. Chromosomes were codified using integer numbers and the number of genes is equal to the number of waypoints lines. This work's layouts (Figure 2) had 8 waypoints lines so the chromosome had 8 genes whose values represented which waypoint of the line would be considered for the path. An example of a random chromosome and the path it would generate in a game (considering the waypoints positions) is shown in Figure 3. A population of 8 individuals was used and initialized randomly, respecting the number of waypoints in each gene's line.

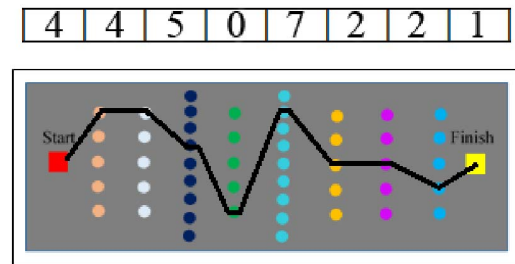


Figure 3. A random path and it's representation in this work's algorithm.

Fitness function designed for this problem is based on the fact that the better individual is able to cross the entire scenario in the shortest path losing the least possible amount of life. So the problem is not only considering to find the shortest path but also a path that is safer. A fitness function proposed for this problem is presented in equation 3.

$$f_i = \frac{l_i}{\sum_{j=0}^n d_{i,j}} \quad (3)$$

Fitness function f_i is composed by l_i , the numerical value for individual's i life, divided by a sum of $d_{i,j}$ that represents the entire path divided in distances from each waypoint j to the immediate next for i individual calculated from *start* point until it reaches the *finish* point.

In basic genetic algorithm the population is sorted from the best fitness to the worst. Individual's selection to crossover considers all the population in pairs of consecutive individuals (first with second, third with fourth and so on). These two procedures together generates the elitism implemented in the base GA algorithm since high fitness individuals will never be mixed with the worst fitness individuals. For each generation, the worst fit individual is also replaced by the most fit individual ever found. Initial test executions used to set the algorithm showed that despite of decreasing genetic variability this procedure achieved better results. Other choice based on the results of initial tests execution is the choice for one point crossover. Other crossover methods changed the parents path abruptly resulting in lower fitness individuals.

A version of multi-point mutation was implemented where according to mutation rate the algorithm changes a gene value to a new random value among all the valid possibilities. It is possible to notice that *a priori* information was added in the base GA implementation by limiting each gene modification considering only valid intervals.

The last definition regarding base GA is how each generation of the algorithm works. Since each chromosome represents a possible path, the fitness of an individual is calculated by a simulation of an agent following this path through the proposed layout. As the agent reaches the *finish* waypoint the results of the final life status are obtained and the fitness can be calculated. Initially a 3D world (Figure 4) was designed in order to allow the analysis of the results, but even using a high-end hardware simulations were taking a prohibitive time to be executed. The solution for this problem was to design a simulation 2D environment that generated the layout images used in this and next sections.

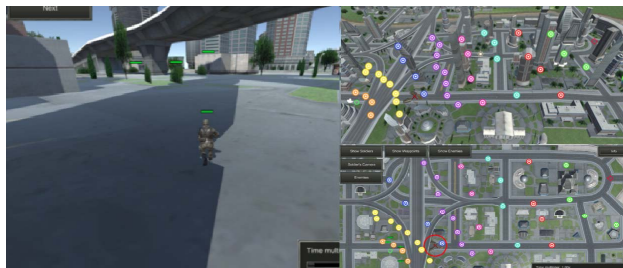


Figure 4. 3D scenario designed for test and analysis.

Considering all tested versions, other two procedures were chosen for the final version of the algorithm and are presented next.

B. Custom Mutation

One of the best procedures design in this works was the Custom Mutation (CM). Usual mutation works by changing a specific waypoint index to a new random waypoint in the group. But the normal way may produce new results that will greatly increase the total path distance, as a random new waypoint may be on the other side of the scenario.

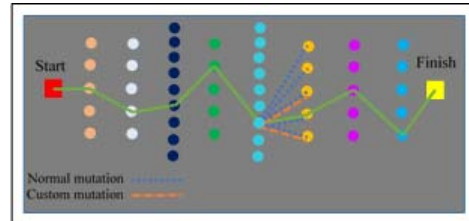



Figure 5. Custom mutation and regular mutation compared.

Custom mutation approach also mutates the current waypoint, but only to the immediate neighbor, up or down. This way, the new individual wont be too discrepant of the previous, which may lead to a faster converge speed. An example can be found in the Fig.5. The blue dotted lines correspond to the normal mutation, which can lead to any other waypoints. The custom mutation, represented by the orange dotted lines, can only lead to the immediate neighbors of the current solution.

C. Virus Infection

The virus infection approach used in this work is based on the idea that a virus enters an organism if the organism is weak or the virus is strong. This is simulated by the virus having a dynamic infection rate. This rate considers if the virus provided a better fit result than the normal individuals and, if so, its rate is increased proportionally.

Virus



| | | | | | | | | |
|---------------------|---|---|---|---|---|---|---|--|
| Normal individual | | | | | | | | |
| 2 | 1 | 7 | 0 | 3 | 5 | 2 | 5 | |
| Selected area | | | | | | | | |
| 2 | 1 | 7 | 0 | 3 | 5 | 2 | 5 | |
| | | | | | | | | |
| Infected individual | | | | | | | | |
| 2 | 1 | 7 | 0 | 4 | 3 | 2 | 5 | |

Figure 6. Example of a viral infection.

Also the block that will be changed by the virus is chosen randomly, which means that the same virus can produce different results from the same individual. Lastly, for each individual that had its fitness increased by the virus will contribute to increase the infectability rate, otherwise it will decrease. Infectability rate starts at 30% and is increased or decreased in 5%. If the infectability reaches zero the virus is destroyed and a new one is generated.

In this work the virus size is 20% of the chromosome and the virus information is randomly generated. An individual

that has contracted the virus will have its path information replaced by the virus information on the position that the virus attacked. This procedure is exemplified in Figure 6.

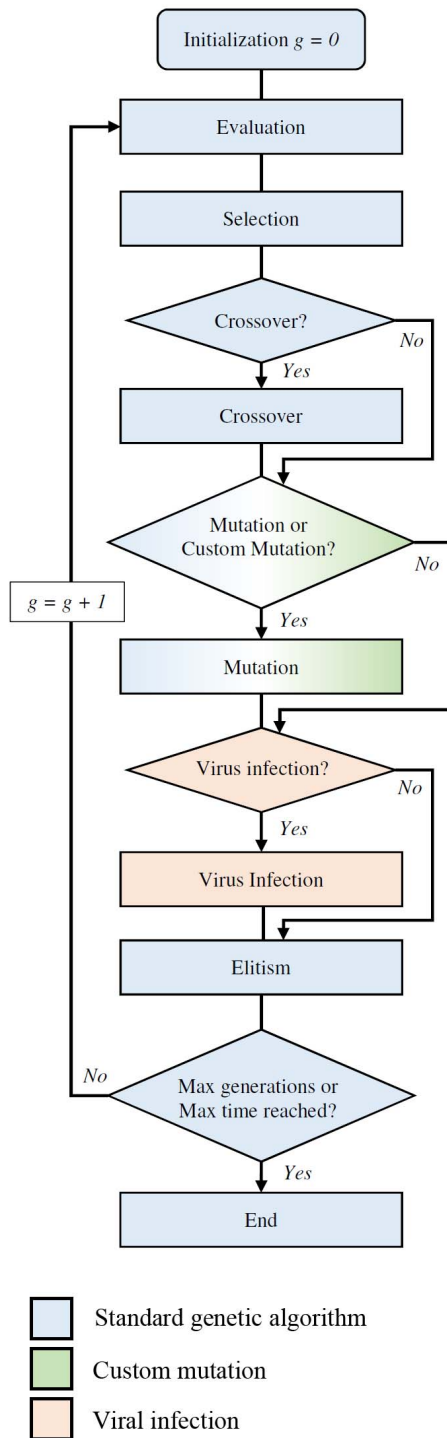


Figure 7. Diagram of the experiments.

D. Dynamic Environments

In this work's scenarios the dynamism is represented by a new enemy (or enemies) added near a certain waypoint. When this situation happens there is a specific approach by the algorithm to deal with it. An external event is generated by the new enemy and is handled in the next generation first step. The algorithm in the beginning of the generation changes the chromosome gene that represents the closest waypoint to the enemy. The value of the gene is randomized, that means, a new value is generated randomly for this gene in order to allow the algorithm to explore the search space again. This procedure allows the algorithm not to be trapped in a previously evolved individual chromosome that is probably turned into a local minimum in the best case. This approach is used to reduce convergence time to find the new best path, since the same genetic algorithm started from scratch could also find this new path, but will probably take more time due to the fact that this work's algorithm variability is mainly based on mutation that usually have a low rate.

Considering the situation presented in Figure 5 an enemy could be inserted next to line 2. Since individual's chromosome is represented by $\{2, 1, 3, 3, 1, 2, 0\}$ according to the green line, the algorithm would mutate the second gene of all individuals (which, in this case, is equal to the first number 1) allowing the algorithm to explore the new search space.

The complete flowchart of the implemented algorithm can be seen in the Figure 7. The algorithm was implemented on Unity 2019.1.3f1 using C# language and executed using an Alienware M15R2 (2015) with MS Windows 10.

V. RESULTS

This section shows all the results from the configurations described in previous section. For each combination the algorithm is executed 20 times. Extracted information is the best fit, convergence generation number and the time elapsed to converge. Converged generation is obtained from a variable that saves the generation number in which the best fitness was achieved. From these 20 executions it is shown the mean value with its standard deviation. According to [34] this number of experiments is enough for statistically relevant results.

Results are presented side-by-side in Figure 8. The first row presents each scenario used for the executions, the second row presents the best fitness, the third row presents the number of the generation which the best fitness were found and the fourth row presents the execution time needed to find the best fitness.

Considering the best fitness row (second row of Figure 8) it is possible to see that both the first and the second scenarios were not a challenge for designed algorithms to find the shortest path that is the less harmful. The third scenario that contains more enemies is the real challenge

Figure 1 consists of three panels illustrating different pathfinding scenarios. The left panel shows a simple path from a red 'Start' square to a yellow 'Finish' square, passing through a series of colored dots (orange, white, dark blue, green, light blue, yellow, purple, and cyan). The middle panel shows a path from 'Start' to 'Finish' that avoids several red circular obstacles. A dashed line indicates a blocked path that would have gone through the obstacles. The right panel shows a more complex path from 'Start' to 'Finish' that avoids a larger number of red circular obstacles. A dashed line indicates a blocked path that would have gone through the obstacles.

205

for the algorithm. In this case the best algorithm was the most complete algorithm (base GA + custom mutation + viral infection) with crossover at 60% and mutation at 15%. Also the versions base GA, Base GA + viral infection and base GA + custom mutation + viral infection with crossover at 80% and mutation at 15% are statistically tied.

Third row of Figure 8 presents the generation that each version of the algorithm found the individual that had the best fitness. For the first scenario the best algorithms were the base GA + custom mutation + viral infection with crossover at 60% and 80%, and mutation at 15% and 5% respectively. Second scenario had the versions base GA and base GA + custom mutation both with crossover at 80% and mutation at 5% as the best algorithms. Base GA + custom mutation with crossover at 80% and mutation at 5% was the best algorithm for the third scenario followed by base GA + viral infection with crossover at 80% and mutation at 15% and base GA + custom mutation + viral infection with crossover at 60% and mutation at 15%.

Execution times of each execution are presented in fourth row of Figure 8. In the first scenario the fastest algorithm was base GA + custom mutation + viral infection with crossover at 80% and mutation at 5%. In the second scenario the best algorithm was the base GA with crossover at 80% and mutation at 5% followed closely by base GA + custom mutation with crossover at 80% and mutation at 5%. The most complex scenario had base GA + custom mutation algorithm with crossover at 80% and mutation at 5% as the fastest algorithm followed by base GA + viral infection with crossover at 60% and mutation at 15%.

Results showed that mutation is a key feature for a evolutionary-based solution for pathfinding using this work's representation and single-point crossover. Also ranking the results considering only the number of situations that a version is the best among other is possible to notice that the best versions of the algorithm are the base GA + custom mutation and base GA + custom mutation + viral infection suggesting that only the base GA is not able to handle the problem justifying other solutions research and design.

Considering achieved results, authors suggest that this algorithm should be used complete (base GA + viral infection + custom mutation) with crossover rate between 70% and 80% and mutation rate between 5% and 10%. No change on the best algorithms occurred when new enemies were dynamically added in each of the test layouts. The time and number of generations needed for convergence for about 200 different situations tested were lower than 150% of the complex layout.

VI. CONCLUSION

This work proposes a viral infection genetic algorithm with dynamic infectability to solve the problem of pathfinding in a dynamic tower defense environment considering both the size of the path and the level of life when the

agent crosses the scenario. Algorithms presented in section II showed different solutions for the problem but none of them solved the problem completely. Results of this work showed that the proposed algorithm was designed to be able to handle dynamic environments achieving this goal.

Proposed solution depends only on waypoints layers in any representation chosen by the game designers, achieving the requirement of not being dependent of a specific map representation as a graph or a grid.

The best analyzed system reaches a 6ms frame time in 1080p resolution. The proposed algorithm achieved convergence in 87.27ms (worst case and in a weaker system), which means a 14-frame solution. However this work's algorithm allows the system to use a intermediate solution even in the first frame, when approximately 400 generations would have been executed. This result is expressive in comparison to solutions presented in section II because all of them need to run a complete execution to allow the NPC to move through the game.

It is important to notice that other algorithms are able to work around obstacles and our solution only moves forward. This is not a problem in this case because enemies are not able to block the path, but only damage the agent's life so it is not a deficiency for tower defense games. One solution to this problem in other cases is to hardly penalize the waypoints covered by a blocking obstacle directing the solution to a valid path.

If a certain game design requires artificial intelligence to find the best path regarding algorithm execution time and restrictions (for example: path size and agent's life), then this work's algorithm can be considered a good choice (as any evolutionary solution the challenge is to improve/change the base rates values proposed by the authors and, in this work's case, position the waypoints). If the problem is regular pathfinding regarding execution time in dynamic environments, then this work's algorithm results should be compared to stated solutions such as D*-Lite and Theta* before its use.

REFERENCES

- [1] Amador, G. P.; Gomes, A. J. P. xTrek: An Influence-Aware Technique for Dijkstras and A Pathfinders, *International Journal of Computer Games Technology*, Article ID 5184605, 2018.
- [2] Bridges, C. L.; Goldberg, D. E. An analysis of reproduction and crossover in a binary-coded genetic algorithm. 2nd intl conf. in genetic algorithms and their applications, 1987.
- [3] Chen, J.; Holte, R. C.; Zilles, S.; Sturtevant, N. R. Front-to-End Bidirectional Heuristic Search with Near-Optimal Node Expansions. *CoRR*, 2017.
- [4] Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; Stein, C. *Introduction to Algorithms*, Third Edition (3rd ed.). The MIT Press. 2009.

- [5] Cui, X.; Shi, H. A*-based Pathfinding in Modern Computer Games. *International Journal of Computer Science and Network Security*, VOL.11 No.1, 2011.
- [6] Fanton, A. Point seeking: a family of dynamic path finding algorithms. PhD Thesis. Rochester Institute of Technology.59 p. 2007.
- [7] Franco, D. S. C. Performance evaluation of Genetic Algorithm with Retroviral Iteration for Real-Valued Functions Optimization. Master Thesis, Institute of Technology - Federal University of Para - Brazil. 92p. 2015.
- [8] Goldberg, D. Genetic Algorithms in Search, Optimization and Machine Learning. Reading, MA: Addison-Wesley Professional. 1989.
- [9] Guedes, A. C. B. ; Leite, J. N. F.; Aloise, D. j. A viral infection genetic algorithm for TSP problem. in *PublCa*, v. 1, p. 16-24, 2005.
- [10] Harris, L. R. The bandwidth heuristic search. In *Proceedings of the 3rd international joint conference on Artificial intelligence (IJCAI'73)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 23-29. 1973.
- [11] Hart, P. E.; Nilsson, N. J.; Raphael, B. A formal basis for the heuristic determination of minimum cost paths, *IEEE Transactions on Systems Science and Cybernetics* SSC-4(2) , 100-107. 1968.
- [12] Hopcroft, J. E.; Schwartz, J. T.; Sharir, M. On the Complexity of Motion Planning for Multiple Independent Objects; PSPACE- Hardness of the Warehousemans Problem. *The International Journal of Robotics Research*, 3(4), 7688. 1984.
- [13] Kanoh, H. Dynamic route planning for car navigation systems using virus genetic algorithms, in *International Journal of Knowledge-based and Intelligent Engineering Systems*, vol. 11, p. 65-78, 2007.
- [14] Keidar, O.; Agmon, N. Safe navigation in adversarial environments. *Annals of Mathematics and Artificial Intelligence*, v. 83, n. 2. pp 83-121. 2018.
- [15] Korf, R. E. Depth-first iterative-deepening: An optimal admissible tree search, *Artificial Intelligence*, Volume 27, Issue 1, Pages 97-109, 1985.
- [16] Kraft, C. Implementation and comparison of pathfinding algorithms in a dynamic 3D space. Bachelor Thesis preseted in University of Applied Sciences Hamburg, Faculty of Design, Media and Information Department Media Technology. 106p, 2019.
- [17] Krishnaswamy, K. A Comparison of Efficiency in Pathfinding Algorithms in Game Development. PhD Thesis. The College of Computing and Digital Media DePaul University. 25 p. 2009.
- [18] Kubota, N.; Shimojina, K.; Fukuda, T. Virus-Evolutionary Genetic Algorithm, *Computers and Industrial Engineering*, v. 30, i. 4, p. 506-509, 1996.
- [19] LaValle, S. M. Planning Algorithms. Cambridge University Press, New York, NY, USA. 2006.
- [20] Machado, A. F. da V.; Santos, U. O.; Vale, H.; Goncalves, R.; Neves, T.; Ochi, L. S.; Clua, E. W. G. Real Time Pathfinding with Genetic Algorithm. *Proceedings of X SBGames - Computing Track*. 2011.
- [21] Millington, I. Artificial Intelligence for Games, Third Edition (3rd ed.). CRC Press Taylor and Francis Group. 2019.
- [22] Nery, A. S.; Sena, A. C. Efficient A* Co-processor for Re-configurable Gaming Devices. *Proceedings of XVII SBGames Foz do Iguacu - PR -Brazil*, pp. 448-457. 2018.
- [23] Ning, B.; Cao, X.; Xu, Y.; Zhang, J. Virus-evolutionary genetic algorithm based selective ensemble classifier for pedestrian detection. *Proceedings of the first ACM/SIGEVO Summit on Genetic and Evolutionary Computation*, Pages 437-442. 2009.
- [24] Ning, F.; Chen, Z.; Xiong, L. Virus evolutionary genetic algorithm for task collaboration of logistics distribution, in *Proceedings of SPIE*, vol. 6042, p. 1-5, 2005.
- [25] Reif, J. H. Complexity of the mover's problem and generalizations. *20th Annual Symposium on Foundations of Computer Science*, San Juan, Puerto Rico, USA, 1979, pp. 421-427. 1979.
- [26] Russell, S.; Norvig, P. Artificial Intelligence: A Modern Approach (3rd ed.). Prentice Hall. p. 103. 2009.
- [27] Santos, U. O.; Machado, A. F. V.; Clua, E. W. G. Pathfinding Based on Pattern Detection Using Genetic Algorithms. *Proceedings of XI SBGames Brasilia DF*, pp. 64-72. 2012.
- [28] D. Sigurdson; V. Bulitko; W. Yeoh; C. Hernandez; S. Koenig. Multi-Agent Pathfinding with Real-Time Heuristic Search. *IEEE Conference on Computational Intelligence and Games (CIG)*, Maastricht, 2018
- [29] Silva, P. V. F.; Villela, S. M. Applying pathfinding techniques on the development of an Android game. *Proceedings of XV SBGames So Paulo SP Brazil*, p 73-80. 2016.
- [30] Silva, A.; Rocha, F.; Santos, A.; Ramalho, G.; Teichrieb, V. GPU Pathfinding Optimization. *Proceedings of X SBGames - Computing Track*. 2011
- [31] Sint, L.; de Champeaux, D. An Improved Bidirectional Heuristic Search Algorithm. *Journal of ACM* 24, 2, 177-191.1977
- [32] Stentz, A. Optimal and efficient path planning for partially-known environments, *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*, San Diego, CA, USA, pp. 3310-3317. 1994.
- [33] Vermette, J. A Survey of Path-finding Algorithms Employing Automatic Hierarchical Abstraction. University of Windsor. Available in: http://richard.myweb.cs.uwindsor.ca/cs510/vermette_survey.pdf
- [34] Witte, R. S.; Witte, J. S. Statistics, 11th edition. Wiley, 496 p. 2017.
- [35] Wolf, M. J. P., ed. The Video Game Explosion: A History from Pong to Playstation and Beyond , Greenwood Press , Westport/London. 2008.