Path-Finding with a Full-Vectorized GPU Implementation of Evolutionary Algorithms in an Online Crowd Model Simulation Framework

Anton Aguilar-Rivera

Barcelona Supercomputing Center, Barcelona, Spain
anton.aguilar@bsc.es

Abstract. This article introduces a path-finding method based on evolutionary algorithms and a fully vectorized GPU implementation of it. The algorithm runs on real-time and it can handle dynamic obstacles in maps of arbitrary size. The experiments show the proposed approach outperforms other traditional path-finding algorithms (e.g. A*). The conclusions present further improvement possibilities to the proposed approach like the application of multi-objective algorithms to represent full crowd models.

1 Introduction

Crowd behavior has been widely studied in the literature [6], being crowd models especially important to the digital entertainment industry, where animation of large groups of characters are desired. Moreover, crowd models are also important to architectonic design and emergency planning. For example, the number and position of facility exits should be carefully selected to minimize evacuation time in the case of an emergency [8].

Crowd simulation is a complex problem. Both map size and number of agents are concerns when devising scalable simulators. Besides, a natural behavior of agents is usually desirable, specially for visualization use. Also, researchers working with crowd simulations may be interested in the over-all effect of subtle changes in the model, specially in social sciences studies [13].

In their survey, Ijaz, Sohail, and Hashish [6] made a classification of the different types of crowd models. They can be classified by resolution in the following manner: Macroscopic, mesoscopic, and microscopic. Macroscopic models describe the general motion of the crowd only. Mesoscopic models are based on cellular automaton and provide better resolution; movement rules are applied to the grid instead to agents. On the other hand, in microscopic models the crowd is composed by individual agents who make their own decisions. As expected, computational cost increases with an increase of resolution.

Microscopic models should provide information about the agent's position, speed, acceleration, intended stops, and others. Algfoor, Sunar, and Kolivand [1] remark the importance of path-finding for crowd simulation problems. In a microscopic setting, a path from each agent's current position to their goals should be defined. The problem can include dynamic obstacles as well. Besides,

ICCS Camera Ready Version 2019
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-22750-0_17

agents are usually programmed to avoid collisions with each other. Realism level and terrain texture are also possible restrictions to the path-finding problem.

This article is concerned with path-finding in the context of crowd simulation. This means the proposed approach should be able to provide paths to a large number of agents and be fast enough to be used for real-time visualization. A zone-based hybrid approach is proposed to allow studying of psychological factors and other subtle elements of the model. Also, this work intends exploring an evolutionary computation approach to path-finding problems. Further arguments in favor of the proposed approach are explained below.

1.1 Background

A general classification of solution approaches will be explained first to later discuss the proposed method under the light of the current state-of-the-art. Ijaz et al. [1] classify the crowd simulation approaches in the following categories: Zone-based models, layer-based models, and sequential models. Each of these categories may use a combination of methods.

Zone-based models divide the map and apply different resolution levels to each part of it. This approach allows handling large maps efficiently. Although, space restriction also restrict the possible solutions to the problem. Also, this means the optimal path could change if the zones are defined in a different manner.

Layer-based models apply both macro models and micro models simultaneously but they are applied in different layers. This allows to separate global planning from local navigation. Techniques like cellular automaton are used to model the global level, using simple rules to guide agents, while the refined movement is computed in other layers. Their problem is psychological factors are not included in the global level and the approach is still dependent of crowd density because its application of microscopical models to individual agents.

Finally, sequential models apply both macro models and micro models, one after another. The macroscopic model is applied until more refined movement is needed. Then, the system switches to a microscopic model. Synchronization is important while applying this approach. These models are better suited to cases where crowd movement is stable.

In regards of path-finding techniques, Algfoor et al. [1] classify techniques in two categories: Terrain-based methods and hierarchical techniques. The former is further divided into regular or irregular grids. Regular grids differ in their geometric shape (square, triangular, etc). They mention visibility graphs, mesh navigation, and waypoints to be techniques to create irregular grids. Some examples of hierarchical techniques are probabilistic road maps, quadtrees, and rapidly explored random trees.

The use of genetic algorithms (GA) for crowd simulation and path-finding problems is mentioned in the literature. Most of the works using GAs have limited their use to secondary parts of their respective solution approaches. For example, Johansson and Helbing [8] have used GAs to define the number of position of facility exits to minimize evacuation time. Vigueras, Lozano, Orduna,

ICCS Camera Ready Version 2019
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-22750-0_17

and Grimaldo [18] presented a zone-based crowd simulation model where GAs are used to determine map partitions. Zones were defined using Convex-hulls. Junior, Musse, and Jung [9] proposed using neural networks to estimate crowd density in subway stations. Optimization of the method was performed using GAs, among other techniques. Zhong et al [20] used GAs to calibrate their crowd simulation model. Bera and Manocha [2] also used GAs, this time to allow a model to learn crowd movement patterns from data.

Direct application of GAs to solve path-finding problems seems to be limited in the literature, with less publications than the approach explained above. Naderan-Tahan and Manzuri-Shalmani [11] presented a GA specifically designed to solve path-finding problems. In their approach, each gene represented a point in the map space, and assumed that points were joined by linear segments to form the path. The chromosomes could have variable length. The first and the last genes of the chromosome were always the initial point and the goal. Some of this approach's drawbacks are the initial population should be obtained using other methods besides random initialization, which could bias the result towards premature convergence. Besides, the reported times are too high to be applicable to online visualization, although, their method was intended for robot navigation instead.

Song, Wang, and Sheng [16] proposed improvements. They used Bézier curves to define the paths instead of linear segments. The former approach is better suited to provide smoother paths than the latter. Besides, the paths are fully-derivable. In their approach, the map is divided by a fixed grid, where tile centers become potential control points to Bézier curves (i.e. paths). The GA searches for clear paths using these control points only. This feature simplifies the search, but it limits path resolution; there could be cases where the available control points are not enough to find a clear path for some complicated parts of the map.

Their method solves the global path-finding problem (i.e. the full map) using a one-level microscopic approach. Therefore, a global optimization process should be applied for each agent. This will have an impact on performance in large maps. On the other hand, their experiments show grids of 16×16 only and execution times were not reported, therefore, the performance of the approach is unknown. The reported setup suggests their approach was not intended for online simulations. Although, the main focus of that article was the use of GAs to path-finding problems instead of high performance.

This analysis seems indicate a novel implementation approach is necessary to make GAs useful for online path-finding. The references where GAs are explicitly used for path-finding are few because their execution times. Nevertheless the use of hybrid computation technologies (e.g. GPUs) open new possibilities to the application of GAs.

The combination of GPUs with machine learning techniques is a popular trend. Neural networks and deep learning are some examples [4]. Although, the evolutionary computation community has made efforts to accelerate their methods with GPUs and other distributed technologies [5]. The problem of evolution-

ICCS Camera Ready Version 2019
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-22750-0_17

ary algorithms is they are inherently sequential, being necessary the population of the last generation to compute the next one. Although, state-of-the-art implementation have been reported in the literature.

For example, Pospichal, Jaros, and Schwarz [15] presented an implementation for NVIDIA GPUs using CUDA. Nowotniak and Kucharski [12] reported a GPU-based GA inspired on quantum systems. Wang and Sheng [19] introduced a GPU GA for task planning. Jaros [7] reported a multi-GPU island-based genetic algorithm for solving the knapsack problem.

Therefore, the approach proposed in this work is an extension of the current effort to use GAs for path-finding problems [11], [16]. This approach introduces an GPU-based GA implementation for path-finding-problems. The implementation was designed with performance in mind, and the details of how this goal is reached are explained in sections below. It is a dynamic zone-based model-free method, where macroscopic and microscopic models are managed in layers. The implementation is able to handle maps of any size and generate paths at frame-rate time even when dynamic obstacles are present in the implementation. The implementation is tested against standard path-finding implementations and suitable experimental results are presented to show the performance improvement of the proposed approach.

The rest of the article is organized in the following sections: section 2 introduces the approach, covering both the path-finding algorithm and the GPU, full-vectorized, GA implementation. Section 3.1 explains the experiments. Section 3.2 presents the results. Section 4 is the discussion, and the conclusions appear on section 5.

2 Proposed Approach

This article follows the trend of the references mentioned above [11], [16]. In a similar manner, Bézier curves are used to describe paths because they can be handled by GAs easily and because of their mathematical properties. Bézier curves are defined in the following manner:

$$\mathbf{B}(t) = \sum_{i=1}^{r} \binom{n}{i} (1-t)^{r-i} t^i \mathbf{P}_i. \tag{1}$$

Where t is a parametric variable in the range $[0, 1]$ and $\mathbf{P}_i$ are the control points. An explicit version of Eq. 1 is

$$\mathbf{B}(t) = (1-t)^r \mathbf{P}_0 + \binom{r}{1}(1-t)^{r-1} t \mathbf{P}_1 + \cdots + \binom{r}{r-1}(1-t) t^{r-1} \mathbf{P}_{r-1} + t^r \mathbf{P}_r. \tag{2}$$

GAs encode these control points, allowing them handling complete curves using a few parameters only. Smooth transition between path segments can be achieved when the derivatives of the curve are included in the optimization process. Further detail can be found in the references [16].

ICCS Camera Ready Version 2019
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-22750-0_17

2.1 GA-GPU Implementation

The implementation both considers the encoding of Bézier curves and a GA implementation on GPU. This approach makes use of the Julia language [3]. The implementation also uses ArrayFire [10], a parallel computing library that interfaces with either CUDA or OpenCL. The library frees the user from the usual burdens of managing GPUs dedicated hardware. Although, high performance with ArrayFire can only be achieved by using full-vectorized code. Therefore, vectorization is a priority of the present approach.

The implementation proposes a traditional GA where selection, crossover, mutation, and evaluation of individuals will be performed for a fixed number of generations. The sequential nature of these operation avoids a generation-wise, parallel implementation. Parallelization is performed at population level, processing a large number of individuals simultaneously.

Coding Individuals are encoded using virtual genes [17]. Differing from the references, This implementation uses 2 values to represent the control points, one for the x coordinate and other for the y coordinate. The values of the coordinates are random numbers in the range $[0, 2^b - 1]$, where b is the number of bits. In our case $b = \log_2(N)$, where N is the size of the map. The implementation is intended to work with $N \times N$ map tiles. The values of the current position and goal position should be passed to the GPU and appended to the chromosomes to perform further operations. We will call this variable g_d from now on. Finally, we assume the population has n individuals and m genes. Therefore, the population is of size $n \times m$.

Selection The vectorized implementation is better expressed in equations. This operator is a variation of tournament selection. Let us define f_s to be the vector of fitness values of the population and f_{sp} the vector of fitness values of the shuffled population. Shuffling is performed using the Julia rand() command to create a AFArray with values from 1 to N. Also, we have i_x, which contains the original indices of the individuals, and i_{xs} are the shuffled indices. Assuming minimization, we define the following variables:

$$\Delta_1 = \operatorname{sgn}\left(\operatorname{sgn}(f_{sp} - f_s) + 1\right), \tag{3}$$

$$\Delta_2 = \operatorname{sgn}\left(\operatorname{sgn}(f_s - f_{sp}) + 1\right). \tag{4}$$

Where sgn is the sign operator. The indices of the selected individuals are

$$i_{xn} = i_x. * \Delta_1 + i_{xs}. * \Delta_2. \tag{5}$$

Where the $.*$ operator denotes element-wise multiplication. Δ_1 is always 0 when $f_s > f_{sp}$ and the contrary is true for Δ_2. In this way, it is easy to discriminate the tournament winner. Individuals with index i_{xn} will reach the next generation and will be subject to further operations.

ICCS Camera Ready Version 2019
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-22750-0_17

Crossover Crossover operation involves 2 tasks. One is swapping the chromosome around the crossover gene, the other is applying inter-bit crossover to it. The first part requires splitting the population into left, right, and center parts. Let us assume c_r, is a matrix of random values in the range $[1, m]$ (Julia uses 1-based indexing) and c_c is a matrix with the cumulative sum of c_r, column-wise. Then, the linearized crossover indices should be

$$x_{\text{vec}} = v_m + n. * x_{r_{0,m-1}} \tag{6}$$

Where v_m is a vector with consecutive values from 1 to m, and $r_{0,m-1}$ is a vector of random values in the range $[0, m-1]$. Therefore, the c_c value at the crossover point should be $x_{\text{ix}} = c_c\left[x_{\text{vec}}\right]$. Let us now assume x_{mix} is a matrix composed by m copies of x_{ix}. The discrimination values m_d of the left, right, and center part of the chromosome are

$$s_{\text{gs}} = \text{sgn}\left(c_c - x_{\text{mix}}\right). \tag{7}$$

s_{gs} will be 0 at the crossover point, the left part will be -1 and the right part will be 1. We can create a mask for each part with the following equations:

$$m_X = 1 - \left|s_{\text{gs}}\right|, \tag{8}$$

$$m_L = \|0.5(-s_{\text{gs}} + 1)\| - m_X \tag{9}$$

$$m_R = \|0.5(s_{\text{gs}} + 1)\| - m_X \tag{10}$$

Where m_L, m_R, and m_X are the masks for the left, right, and center parts, respectively. $\| * \|$ denotes the round operation. g_d can be multiplied by any of these masks to extract the corresponding part of the chromosome for all the individuals in the population simultaneously.

The second task is performing inter-bits crossover at the selected gene. Valenzuela-Rendón [17] explains the needed integer value to extract the low parts from the binary string representing the crossover gene from parents p_1 and p_2 with virtual genes is

$$\mathcal{X}_m\left(p_1, p_2\right) = p_1 \text{mod} 2^{m_c} - p_2 \text{mod} 2^{m_c}. \tag{11}$$

Where m_c is a random number between $[1, m]$ (assuming 1-based indexing). The post-crossover population is finally computed in the following manner:

$$g_d = gd_1. * m_L + g_{d_2}. * m_R + \left(g_{d_1} - \mathcal{X}_m\right). * m_X \tag{12}$$

Where g_{d_1} and g_{d_2} denote the original population and a scrambled copy of it.

Mutation Mutation follows the same approach used on crossover. This time, 2 masks are created to separate the mutation gene from the rest of the chromosome. In this case c_m is a matrix of size $n \times m$ of random numbers in the range $[0, 1]$. Then the mask would be

$$m_m = \text{sgn}\left(\text{sgn}(c_m - p_m) + 1\right) \tag{13}$$

ICCS Camera Ready Version 2019
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-22750-0_17

Where p_m is mutation probability. To obtain the mutated gene, we use the following mask:

$$m_{m_1} = 1 - m_m \tag{14}$$

Finally, the post-mutation population is

$$g_d = g_{d_1} . * m_m + g_{d_r} . * m_{m_1} \tag{15}$$

Where g_{d_1} is the original population, and g_{d_r} is a matrix of random genes.

Evaluation The particular implementation is focused on finding clear paths, although, other optimality criteria could be used as well. Evaluation is composed by 2 tasks: Computation of Bézier curves and computation of the fitness function.

We need g_d and the values of t to compute the curves. Eq.2 is used for an efficient computation on the device (i.e. GPU). Let us assume t_d is a matrix of size $(r+1) \times (1/\Delta)$, where Δ is the resolution of t. t_d holds the terms of Eq.2 for each value of vector t. t is defined from 0 to 1 with a step size Δ. Therefore $g_d \times t_d$ will compute the points of the curve.

In the actual implementation, the columns of t_d are duplicated to manage separately the x coordinates from the y coordinates. This means t_d has the form

$$t_d = \begin{bmatrix} (1-t_{0x})^r & (1-t_{0y})^r & \dots & t_{0x}^r & t_{0y}^r \\ (1-t_{1x})^r & (1-t_{1y})^r & \dots & t_{1x}^r & t_{1y}^r \\ \vdots & \vdots & \dots & \vdots & \vdots \\ (1-t_{\Delta x})^r & (1-t_{\Delta y})^r & \dots & t_{\Delta x}^r & t_{\Delta y}^r \end{bmatrix} \tag{16}$$

To allow an effective multiplication a pair of masks are defined: st_x and st_y. Where

$$\mathrm{st}_x = \begin{bmatrix} 1\,0 \dots 1\,0 \\ \vdots\,\vdots \quad \vdots\,\vdots \\ 1\,0 \dots 1\,0 \end{bmatrix}, \tag{17}$$

while st_y would be

$$\mathrm{st}_y = |\mathrm{st}_x - 1| . \tag{18}$$

These matrices have successive columns of ones and zeros to avoid multiplication of the wrong values. Therefore, the values of the path will be

$$\mathbf{B}_x(t) = (g_d . * \mathrm{st}_x)\, t_d, \tag{19}$$

$$\mathbf{B}_y(t) = (g_d . * \mathrm{st}_y)\, t_d. \tag{20}$$

The fitness function to evaluate individuals is based on clearance. To compute the metric we need the Euclidean distance between the origin and any other point in the path. Let us call this variable d_{ab}. Also, we need to compute collisions

ICCS Camera Ready Version 2019
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-22750-0_17

along the path. Let us assume M_d is the map loaded on the device. We can use linearized indices from $\mathbf{B}_x(t)$ and $\mathbf{B}_y(t)$ to handle it

$$\mathbf{B}_{\mathrm{xy}}(t) = \mathbf{B}_x(t) + N\left(\mathbf{B}_y(t) - 1\right). \tag{21}$$

Collisions will be simple the indexing

$$c_d = M_d\left[\mathbf{B}_{\mathrm{xy}}(t)\right] \tag{22}$$

Using these variables, we can compute the fitness value in the following manner:

$$f_d = \left(c_d. * d_{\mathrm{ab}}\right) V_1. \tag{23}$$

Where V_1 is a vector of ones used to perform the final multiplication step.

Initialization Data transfer between the host and the device should be used carefully because it has a direct impact on performance. Relevant variables should be created on GPU memory to avoid unnecessary overload. These variables are initialized before the run to save computation time. g_d, f_s, t_d, st_x, st_y, V_1 are all suitable candidates to initialization. Also, the algorithm returns the best-so-far individual instead of the final population to avoid unnecessary overhead time.

2.2 Path-finding Algorithm

Path-finding is divided in a static stage and a dynamic stage. Fig. 1 shows a flowchart with the path-finding algorithm.

Static_Field() refers to the static path-finding stage that is computed off-line to save time. Breadth-first search is used to create a gradient field on the map. This approach was preferred because it is able to provide paths to any number of agents. Other methods could have been used as well.

Initialization() refers to the setup mentioned in section 2.1. The necessary variables are loaded on the device beforehand to save time. Besides, these variables will be used constantly along the run, therefore, overhead is reduced drastically by applying this step.

Static_Path() deals with providing a static path to a particular agent. Given the field, this step is straightforward. The static path is used to guide the agent in a global scale. We say it is a macroscopic path in the sense it provides the general path movement of the agent, which will be subject to corrections when dynamic obstacles appear in the way.

The actual path-finding algorithm starts with the Move() function. This function moves the agent subject to different conditions, depending on the type of simulation desired. While moving, the agent is constantly applying Is_Goal() and Is_Obstacle() functions. Their names are self descriptive, but we can say Is_Obstacle() search for obstacles s_g steps ahead in the path. The agent moves freely as long as the path remains clear, until the goal is reached.

ICCS Camera Ready Version 2019
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-22750-0_17

The path-finding algorithm is called when an obstacle is on sight on the path. Tiling() is a function that takes a piece of the map of size $2s_g \times 2s_g$. The current position of the agent becomes the local starting point and the closest path point to the tile edges will become the local goal. Tile size should be large enough to allow the necessary freedom to the agent to find an alternative path. Also, dynamic tiling is necessary because we cannot guarantee the tile will contain a clear path to the local goal because the static field was created without obstacles information. Dynamic tiling overcomes this occurrence. Also, tiling centers the obstacle to the tile to avoid obstructions with the edges. Dynamic tiling size could be used to guarantee a path will be found. In the experiments, it was found dynamic tiling was enough to prevent blockage.

Path-Finding() refers to the application of GAs to find clear paths at frame-rate times for the given tile. Tiling is necessary to guarantee GPU's memory is not overwhelmed by large maps. The device capabilities allows to run large populations (in the order of thousands) to speed-up the search. According with GAs theory, larger populations contain a much larger schemata contents and they have more probability to contain the optimal ones. Given speed is our main concern, we need to work with the largest population possible. Also, the number of generations is limited, because the sequential execution has direct impact on performance. Also, the number of control points should be the lowest possible to keep performance up and avoid the path to be unnecessarily complicated. Although, if the GAs cannot find a clear path with the current number of control points, this one is increased and the GA is run again. This will happen until a clear path is found.

Finally, once the alternative path is found, the global path is updated with the new segment. The agent will follow the new path until it finds a new obstacle in sight or the goal is reached.

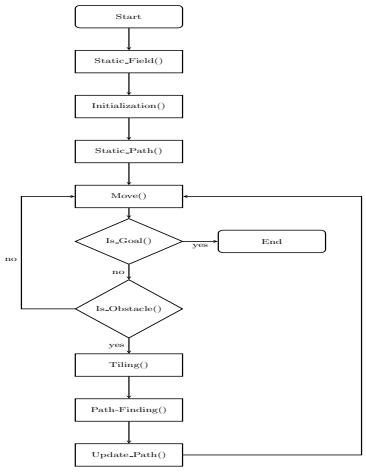

Fig. 1. Path-Finding Algorithm

ICCS Camera Ready Version 2019
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-22750-0_17

3 Experiments

Experiments were conducted to test the performance of the proposed approach. In this case, we used Voronoi maps [14], which were algorithmically created for the experiments. Maps have a size of 2048×2048, while tile size is $2s_g = 32$. Initial and goal points were randomly selected, but they were chosen in a way the goal is reachable from the initial point. Also, we assume obstacles are previously inflated, therefore, there is no potential collision risk involved if paths stick to them.

The approach assumes the necessary time for agents to move along the path is much longer than the time needed to generate alternative paths, therefore, we can assume the algorithm works with snapshots of the world. The path-finding algorithm is activated when Is_Obstacle() function returns true.

3.1 Experiments Description

The experiments tested 100 different maps where the system should provide paths to agents. Dynamic obstacles appear randomly in the path, and the agent system should provide an alternative path for each obstacle. The approach is compared against traditional path-finding methods: Breadth-first search, Dijkstra algorithm, and A*. All the algorithms were run against the same path-finding problems, and the same obstacles. The average running time per obstacle is computed from each method. The experiments were run on a computer equipped with an Intel Core i7-7700HQ CPU @ 2.80GHz 8 processor, 7,7 GiB of RAM, and NVIDIA GeForce GTX 1050/PCIe/SSE2 GPU.

3.2 Experiments Results

The results are shown in figure 3. They present the average time it takes the algorithms to find an alternative path when an obstacle has been detected. The number of obstacles is random, but it is adjusted to have more or less one obstacle per tile size. The average time is obtained dividing the total computation time by the number of obstacles solved. The results are presented on frames/obstacle, where a frame is $1/60s$. We can assume the movement of agents and obstacles is much more slower to be measured on frames, but a faster algorithm will be able to serve a larger number of agents. Fig. 3 shows the comparison between A* and the proposed approach. The experiments using breadth-first search and Dijkstra's algorithm are not shown because they are order of magnitude slower than A*. This happens because the number of revised nodes by A* is much more less than the nodes revised by the other algorithms (Fig. 2). This result was expected. On the other hand, we can see the average of the distribution of GA's running time is much more lower than the average of A*. Fig. The median of GA is less than 1 frame, while the median of A* is around 80 frames per obstacle. Both box plots show a few outliers, where we can assume particular conditions caused a delay for the algorithms.

ICCS Camera Ready Version 2019
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-22750-0_17

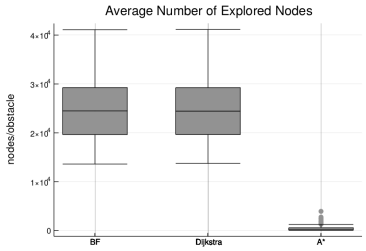

Fig. 2. Average number of nodes explored by search algorithms.

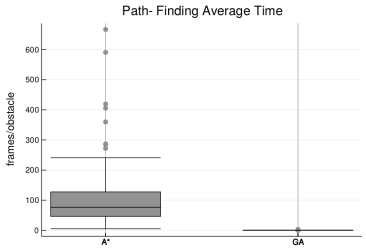

Fig. 3. Average execution time to solve path-finding with dynamic obstacles. Results shown in frames/obstacle, (one frame = $1/60s$)

4 Discussion of Results

Fig. 4 shows examples of paths found by different algorithms. We can see breadth-first tends to find paths that are relatively far away from the obstacle, this occurs due the particular manner this algorithm expands the frontier. On the other hand, A* has a tendency to stick to obstacles to minimize the deviation form the optimal path. In the case of GAs, paths are conditioned to the particular tile used by the algorithm. This happens because both the path and the control points must be located inside the tile, causing the rest of the map to be off-limits to the algorithm. There is a possibility a better path could be found if we change the tile size or position. Therefore, an optimal global path could be rather different from the paths computed using a global approach. Nevertheless, optimizing the whole map could be impossible because of hardware limitations (e.g. GPU memory).

We can observe a slightly erratic movement on the paths generated using Dijkstra's algorithm. We have to say this algorithm is designed to find the less cost path in weighted graphs, which is not the case of this problem. Therefore, cost does not provide guidance to the path-finding process. Finally, we observe the GA has a tendency to generate smoother curves than other algorithms because of the nature of Bézier curves. By the moment, clearance (i.e Eq. 23) is the only fitness metric considered. Of course, we can include minimum distance, smoothness of curves, deviations from a constant speed or acceleration,

ICCS Camera Ready Version 2019
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-22750-0_17

etc. The integral inclusion of all these metrics could probably be achieved with multi-objective evolutionary algorithms.

In regards to outliers, it was found there were cases where the GA ran out of time and it could not find a clear path. This happened when the particular tile was specially complicated. The initial number of control points is 1 and it was increased when the GA failed to find a clear route. It was found sometimes even 5 control points were necessary to find a clear path. We wish to keep the number of control points the minimum possible because of two main reasons: One is to minimize execution time. Also, a Bézier curve has a tendency to become unnecessarily complicated with a large number of control points. Having a way to estimate the necessary control points beforehand will have a positive impact on performance.

In relation to the classification explained in section 1.1, the proposed approach is inherently zone-based because it uses tiling to solve the path-finding problem with dynamic obstacles. Also we can say the method is layered when we consider the global path-finding to be on the macroscopic level and obstacles avoidance belong to the microscopic level. Also it is a terrain-based technique where octagonal nodes are used to allow both straight and diagonal motion.

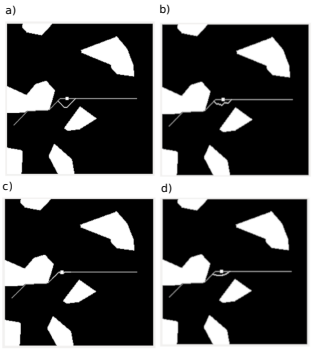

Fig. 4. Example of application of path-finding algorithms: a) Breadth-first search, b) Dijkstra's algorithm, c) A*, d) GAs. The small white square is a dynamic obstacle.

5 Conclusions

This article presented a novel approach to path-finding problems using evolutionary algorithms. The approach is intended to be applied to crowd simulation applications, where online path-finding is desired. The main problem for the application of evolutionary algorithms was the required speed to serve a large number of agents online. The article presented an evolutionary algorithm based

ICCS Camera Ready Version 2019
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-22750-0_17

on Bézier curves to handle the path-finding problem, and a vectorized GPU implementation that allows managing large populations at frame-rate times. The experiments showed the proposed approach is faster than traditional path-finding methods, being A* the most remarkable one.

There are many possibilities of future work for the proposed approach. For example, a multi-objective version of the algorithm can be devised to simultaneously include minimum distance, clearance, and social forces. The advantage of this approach is it will generate a Pareto front of optimal solutions, providing with variety to the simulations. This will happen because the agent will have the possibility of choosing one of the solutions based on its own preference, allowing the inclusion of "personality" to them.

The current method is subject to further improvement. It was explained above how execution of GA was affected by the fact the number of control points should be gradually incremented until the algorithm can generate flexible enough paths to traverse the tile. A method could be devised to estimate this value beforehand, and a good estimation will have a positive impact on performance. The integration of this improvement with a multi-objective version of the algorithm will be able to online full crowd modeling. Also, the algorithm can be compared against other custom path-finding methods and be subject of further improvement.

The current implementation was made using the ArrayFire library. Probably a lower level implementation could be desirable (e.g. using CUDA). This vectorized approach has the advantage porting should be straightforward, because most of the operations used are matrix operations: A task GPUs are specially useful. Also, other approaches to path-finding could be combined with evolutionary algorithms to find novel approaches. This implies the exploration of sequential approaches, hierarchical techniques, and others.

6 Acknowledgments

– The author thanks Consejo Nacional para la Ciencia y Tecnología (CONACyT) for the postdoctoral fellowship at Barcelona Supercomputing Center.

References

1. Zeyad Abd Algfoor, Mohd Shahrizal Sunar, and Hoshang Kolivand. A comprehensive study on pathfinding techniques for robotics and video games. *International Journal of Computer Games Technology*, 2015:7, 2015.
2. Aniket Bera and Dinesh Manocha. Reach-realtime crowd tracking using a hybrid motion model. In *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pages 740–747. IEEE, 2015.
3. Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

ICCS Camera Ready Version 2019
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-22750-0_17

4. Henggang Cui, Hao Zhang, Gregory R Ganger, Phillip B Gibbons, and Eric P Xing. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 4. ACM, 2016.
5. Yue-Jiao Gong, Wei-Neng Chen, Zhi-Hui Zhan, Jun Zhang, Yun Li, Qingfu Zhang, and Jing-Jing Li. Distributed evolutionary algorithms and their models: A survey of the state-of-the-art. *Applied Soft Computing*, 34:286–300, 2015.
6. Kiran Ijaz, Shaleeza Sohail, and Sonia Hashish. A survey of latest approaches for crowd simulation and modeling using hybrid techniques. In *17th UKSIMAMSS International Conference on Modelling and Simulation*, pages 111–116, 2015.
7. Jiri Jaros. Multi-gpu island-based genetic algorithm for solving the knapsack problem. In *Evolutionary Computation (CEC), 2012 IEEE Congress on*, pages 1–8. IEEE, 2012.
8. A Johansson and Dirk Helbing. Pedestrian flow optimization with a genetic algorithm based on boolean grids. In *Pedestrian and evacuation dynamics 2005*, pages 267–272. Springer, 2007.
9. Julio Cezar Silveira Jacques Junior, Soraia Raupp Musse, and Claudio Rosito Jung. Crowd analysis using computer vision techniques. *IEEE Signal Processing Magazine*, 27(5):66–77, 2010.
10. James Malcolm, Pavan Yalamanchili, Chris McClanahan, Vishwanath Venugopalakrishnan, Krunal Patel, and John Melonakos. Arrayfire: a gpu acceleration platform. In *Modeling and Simulation for Defense Systems and Applications VII*, volume 8403, page 84030A. International Society for Optics and Photonics, 2012.
11. Mahmood Naderan-Tahan and Mohammad Taghi Manzuri-Shalmani. Efficient and safe path planning for a mobile robot using genetic algorithm. In *Evolutionary Computation, 2009. CEC'09. IEEE Congress on*, pages 2091–2097. IEEE, 2009.
12. Robert Nowotniak and Jacek Kucharski. Gpu-based tuning of quantum-inspired genetic algorithm for a combinatorial optimization problem. *Bulletin of the Polish Academy of Sciences: Technical Sciences*, 60(2):323–330, 2012.
13. Xiaoshan Pan, Charles S Han, Ken Dauber, and Kincho H Law. A multi-agent based framework for the simulation of human and social behaviors during emergency evacuations. *Ai & Society*, 22(2):113–132, 2007.
14. Evanthia Papadopoulou and Maksym Zavershynskyi. The higher-order voronoi diagram of line segments. *Algorithmica*, 74(1):415–439, 2016.
15. Petr Pospichal, Jiri Jaros, and Josef Schwarz. Parallel genetic algorithm on the cuda architecture. In *European conference on the applications of evolutionary computation*, pages 442–451. Springer, 2010.
16. Baoye Song, Zidong Wang, and Li Sheng. A new genetic algorithm approach to smooth path planning for mobile robots. *Assembly Automation*, 36(2):138–145, 2016.
17. Manuel Valenzuela-Rendón. The virtual gene genetic algorithm. In *Genetic and Evolutionary Computation Conference*, pages 1457–1468. Springer, 2003.
18. Guillermo Vigueras, Miguel Lozano, Juan Manuel Orduna, and Francisco Grimaldo. A comparative study of partitioning methods for crowd simulations. *Applied Soft Computing*, 10(1):225–235, 2010.
19. Kai Wang, Zhen Shen, et al. A gpu-based parallel genetic algorithm for generating daily activity plans. *IEEE Trans. Intelligent Transportation Systems*, 13(3):1474–1480, 2012.
20. Jinghui Zhong, Nan Hu, Wentong Cai, Michael Lees, and Linbo Luo. Density-based evolutionary framework for crowd model calibration. *Journal of Computational Science*, 6:11–22, 2015.

ICCS Camera Ready Version 2019
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-22750-0_17