Action Games: Evolutionary Experiences

Antonio J. Fernández[1] and Javier Jiménez González[1]

Dept. de Lenguajes y Ciencias de la Computación, Univ. de Málaga, Spain*
afdez@lcc.uma.es

Summary. This paper defends the employment of Evolutionary Algorithms (EAs) in action games by showing their virtues for both offline and online opponent controlling. The paper proposes (and also compares) several EAs applied offline in the solving of a classical path finding problem and used to provide intelligence to autonomous agents (e.g., the opponents) in an action computer game. The paper also presents an EA that has been successfully employed in real time (i.e., online) in an action game in which a player controls a military vehicle in a hostile enemy region.
Keywords: Evolutionary algorithms, game programming, real time.

1 Introduction and Related Work

Game programming [1] is an exciting research field because of two main reasons: (1) computer games (CGs) are a business that generates millions of dollars each year, and (2) game programming has a heterogeneous nature that involves very different computer science techniques such as graphical techniques, artificial intelligence algorithms, interactive music, etc.

CGs are becoming one area of increasing interest, and proof of it is the fact that the traditional relation between cinema industry and computer game world is now being reversed, that is to say, in the past, many CGs were directly based on film scripts whereas nowadays, many films are directly based on CGs stories (e.g., Resident Evil, Mortal Kombat, Final Fantasy series, Street Fighter, Tomb Raider, Super Mario, ..., just to name a few).

The aim of CGs is to provide entertainment to the player(s), and in the past, research on commercial CGs was mainly focused on having more realistic games by improving graphics and sound (i.e., having higher resolution textures, more frames-per-second, ...etc). However, in recent years, hardware components have experienced exponential growth and players, with higher

* This work has been partially supported by projects TIC2001-2705-C03-02, and TIC2002-04498-C05-02 funded by both the Spanish Ministry of Science and Technology and FEDER.

processing power computers, demand high quality opponents exhibiting intelligent behavior.

In many action commercial CGs, the opponent (i.e., the enemy) attitude is basically controlled by a fixed script previously programmed that often comprises hundreds of rules, in the form `if condition C is true then execute action A`. This is really a problem as these scripts are often complex programs that contain "holes" easily detected by an experienced player. As consequence, the reality simulation is drastically reduced and thus the interest of the player. This problem relies in the category of "artificial stupidity" [2]. To solve the problem, existing games employ some kind of artificial intelligence (AI) technique with the aim of giving intelligence to opponents and making the game more attractive to increase the satisfaction of the player.

In this context, AI plays an important role in the success or failure of a game and some major AI techniques have already been used in existing CGs [3]. Evolutionary algorithms[2] (EAs) offer interesting opportunities for creating intelligence in strategy or in role-playing games [4] and, in Internet, it is possible to find a number of articles related with the use of evolutionary techniques in CGs [4], although, in general, most of them are dedicated to show the excellence of the use of EAs in Game Theory and, particularly, in the solving of multi-person decision problems [5, 6]. E.g., [6] shows how to use genetic algorithms for evolving control sequences for game agents although the focus is on bot navigation (i.e., exploration and obstacle avoidance). EAs have also been used in games for pathfinding [7].

Surely, EAs are one of the least-understood technologies in the game AI field, and scientific literature lacks proposals of using EAs in realistic action CGs (those in which we are interested on). For us, there are two main reasons for it: (1) game AI and academic AI often evolve separately in spite of the fact that game and academic communities have much to learn one from each other, and (2) EAs are resource intensive and require much time for development and tuning which does not make them adequate for in-game learning [7]. In fact, in game development, each resource is very important to allow the simulation in real time, and EAs are computationally expensive, so that traditionally, game developers have preferred another AI techniques such as Artificial Life, Neural Networks, Finite State Machines, Fuzzy Logic, Learning and Expert Systems, among others [3, 7, 8].

In spite of their cost, EAs are considered as a promising technique that is on the forefront of game AI [9, 10, 11]. Moreover, game programming has already made successful use of EAs offline, that is to say, the EA works on the user computer (e.g., to improve the operational rules that guide the opponent actions) whereas the game is not being played and the results (e.g., improvements) can be used further online (i.e., during the playing of the game). Through offline evolutionary learning, the quality of opponent intelligence in

[2] We use this term in a broad sense making reference to any kind of evolutionary procedure, including genetic algorithms and genetic programming.

commercial games can be improved, and it has been proved that it is more effective than opponent-based scripts [12]. Also, genetic algorithms have been used to evolve combat strategies for agents or opponents between games (i.e., an offline learning) as it was done in the games *Unreal Tournament* [13] and *bSerene* [14]. In this sense, some realistic CGs that have used genetic algorithms are *return Fire II*, *The Creatures Series*, *Sigma* and *Cloak, Dagger and DNA* and *Quake III* [15, 16].

With respect to the online use of EAs in action computer game programming (i.e., EAs are executed in real time at the same time that the game is being played and simulated graphically in a computer), literature lacks studies about it. Perhaps the reason is that realistic action computer games require a real time graphical simulation that consumes a lot of computational resources (e.g., sound, music, and graphics) and EAs are costly; the direct consequence is that their combination seems to be too expensive computationally. Thus, to make a correct use of EAs in game programming one has to consider all these aspects and specifically has to know that the game AI and the game graphical simulation have to be executed together in real time. Speaking about the online evolutionary learning, we have only found one paper in this sense, [17], that proposes methods and strategies for the online coevolution of agents in an action game.

In this paper, we encourage the use of EAs to provide, in action games, AI to autonomous agents controlled offline as well as online. We describe our experience on the employment of EAs on a specific military action game where the scenario conditions change dynamically. We describe (and compare) some EAs used for offline control in this game and, also implement an EA that can be used online: the secret consists of simplifying the set of actions executed by the opponents in order to reduce drastically the search space and accelerate the computation process.

2 The Game

We have implemented a game [18] that recreates an armed forces plan to transport a military vehicle (i.e., the user vehicle), placed in a hostile enemy region (i.e., the scenario), from an origin position to a destination one. Each *scenario* (also called indistinctly *region* or *world*) consists of a two-dimensional non-toroidal heterogeneous hostile dynamic grid-world. The world is *heterogeneous* because the terrain is not uniform, *hostile* because there exist enemy agents whose mission is to destroy the user vehicle, and *dynamic* because the solution search tree continuously change depending on the actions executed by the vehicle and the enemy agents. The purpose of the game is to move the military vehicle from the origin location to the destination position avoiding the natural obstacles of the world and the direct confrontation with the enemy agents in order to prevent the vehicle destruction. To do so, the vehicle is capable to execute some elementary actions such as go straight ahead one single grid square and turning 90° to its left or to its right.

Initially, the vehicle has some resources that must be kept in a controlled way until reaching the target. Another objective of the game is to optimize the expense of these resources: fuel, resistance and time. (The multi-objective consists of minimizing the time and maximizing both fuel and resistance at the end of the game.) The resources decrease during the game: time continuously decreases, fuel goes down according to the action executed by the vehicle (e.g., 4 units to cross a flat ground, 1 unit to turn 90°, 8 units to cross a terrain with gravel,...,etc) and resistance decreases with the attacks from the enemies. There exists also a *timeout* to limit the time to reach the target.

The game offers three kind of worlds with different sizes: 15×15, 30×30 and 50×50, and 4 scenarios for each of them. For each scenario, we have also generated 5 test cases where the origin location and the target position differ from one case to each other. Thus the game manages 60 different worlds; the timeout is constant (300 units) and the fuel is variable according to the world size; 250, 500 and 800 respectively for the worlds with size 15, 30 and 50.

The mission of the enemies is to destroy the vehicle or, alternatively, avoid the vehicle reach the destination grid before timeout. If an enemy crashes into the vehicle, then the vehicle is immediately destroyed. Each enemy has also associated an *area of shooting*, that is to say a rectangular region around the enemy, in which the vehicle is visible and the enemy can shoot the vehicle to decrease its resistance.

3 Off-line Evolutionary Experiences

Our first trial consisted of transforming the game in a simpler (non-interactive) simulation in which the vehicle is an autonomous intelligent agent guided by some EA. The simulation emulates the game on different scenarios to evaluate the quality of our EAs to be used on action games. This means an *offline* intelligence as there is no human player to control the vehicle in real time.

During the game simulation, the vehicle does not know exactly the target position and is equipped with a *position sensor* that divides any world in four zones by the current position of the vehicle and indicates the zone where the target is (see Figure 1(left); the star points out the destination position). The vehicle also owns a *proximate sensor* that provides extra information (e.g., enemy presence, obstacle absence and terrain nature) about the adjacent grids located in the front of the vehicle as it is shown in Figure 1(medium).

The enemies are *dynamic* in the sense that they execute patrols following the cycle North-East-South-West as it is indicated in Figure 1(right).

In the following, by simplicity, if a is a variable belonging to a data type T, we write $a : T$, and if a is a value belonging to some data type T we write $a \in T$. Also, we write $T = (\text{field}_1 : T_1, \ldots, \text{field}_n : T_n)$ to indicate that T is the type composed by the set of all the tuples $(a_1, \ldots, a_n)$ where $a_1 \in T_1, \ldots, a_n \in T_n$. Moreover, if $a : T$ and $a = (a_1, \ldots, a_n)$, then $a.\text{field}_i$ denotes a_i $(1 \leq i \leq n)$. Below we show some of the data types used in the code of our offline EAs:

position = $(val\_x : \mathbb{N}, val\_y : \mathbb{N})$
resources = $(t : \mathbb{N}, f : \mathbb{N}, r : \mathbb{N},)$;
state = {success, timeout, death, withoutfuel, abort, outofrange, impassable};
terrain = {flat, target, river, fine gravel, puddle, mountain, bush, tree, hill};
grid = $(nature : terrain, time\_cost : \mathbb{N}, fuel\_cost : \mathbb{N})$
enemy = $(pos : position, \ldots)$
vehicle = $(pos : position, res : resources, \ldots)$
W_n = ARRAY $[1..n, 1..n]$ OF grid;

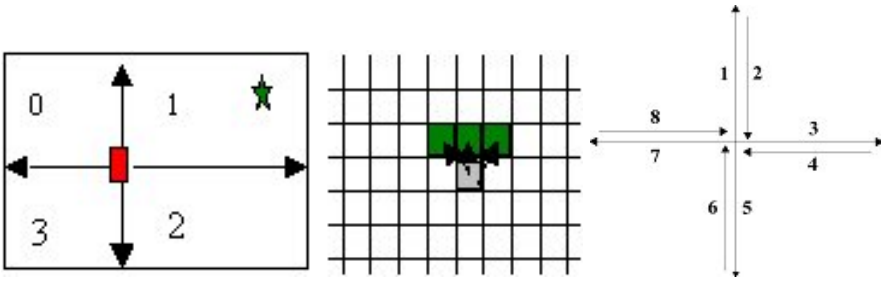

Fig. 1. (Left) Position sensor (Medium) Proximate sensor (Right) Patrol cycle

Basically, each world W_n is a grid bi-dimensional array of size $n \times n$ and, abusing the notation, we write $\alpha_{i,j} \in W_n$ to denote that $\alpha_{i,j}$ is the grid positioned in row i and column j in the world W_n. Grids can be represented as tuples or structures that contain the information about their nature (i.e., type of terrain) and some cost values (i.e., on time and fuel) that indicate the amount of resources spent by the vehicle in case that this crosses a specific grid. Each enemy has associated a position in the world that varies dynamically during the game, and the vehicle is identified, in each instant of the game, by a position inside the world and an amount of resources, i.e. time (t), fuel (f) and resistance (r). Both the vehicle and each enemy maintain also another information, not shown above, to register their actions for a further visualization of the game simulation.

The game objective consists of finding one (optimal) sequence of movements (i.e., a path, represented by a grid sequence associated to the vehicle) that enables the vehicle to reach the destination position safely.

All our offline EAs were implemented without using crossover operators. The reason is in the nature of the problem itself: observe that, in each stage of the game, resource consumption has to be taken into account as this influences the rest of the play. Thus, it is not easy to cross different paths since we have to consider not only segments of the paths (e.g., as in a classical one point or two point crossover) but also the correspondence between the resource consumption in the crossover points. A discussion about the usefulness of the crossover operator is beyond this paper and the reader is referred to [19].

Our offline EAs used these parameters: $pop\_size = 80$, $max\_generations \in \{300, 400, 500\}$ (depending on the world size), $mutation\_rate = 0.01$ and maximum length of chromosome = 250 (i.e., this parameter restricts the maximum number of movements as it is indicated below).

3.1 A Rule-Based Prototype

Fisrtly, we implemented a rule-based prototype (RBP) that defines a fixed script to control the vehicle. This script is composed by a set of behavior rules obtained by following the approach[3] given in [20]. The rules have the form *if direction sensor detects the target in zone i and proximate sensor informs about the nature of adjacent grid j then do action k*. Then information about the terrain nature and the resource cost associated to a grid in the world was used to optimize our set of behavior rules according to [20].

$k \leftarrow 0$
do{ $v[k+1].pos \leftarrow$ move_vehicle_according_rules($v[k].pos$)
$en[k+1] \leftarrow$ move_enemies_according_cycles($en[k]$)
$v[k+1].res \leftarrow$ update_resources($v[k].res, st$)
$i \leftarrow i + 1$
}**while** ($v[k-1].pos \neq (i_d, j_d) \wedge$ % Success
$v[k-1].pos \neq en[k-1, j].pos\ (\forall j \in \{1, \ldots, e\}) \wedge$ % Death
$res[k-1].r > 0 \wedge res[k-1].t <$ timeout $\wedge\ res[k-1].f > 0$)
% Termination: some resource is exhausted

Fig. 2. Basic schema of the RBP algorithm

The basic RBP schema is shown in Figure 2. Variable $k : \mathbb{N}$ denotes the iteration number and an expression $a \leftarrow b$ indicates the assignment from b to a; $\alpha_{i_d,j_d} \in W_n$ identifies the destination grid; *st:state* will contain the reason of the termination of the algorithm that is identified by one of the termination conditions of the loop *do-while*; v, and en are arrays of variable length where $v[k]$ and en[k,j] contain in the k'th-iteration, the information associated, respectively, with the vehicle (i.e., position and resources amount) and with the j'th-enemy, assuming that there are e enemies (for $1 \leq j \leq e$). Note that $v[0].pos$ corresponds with the starting position. Functions *move_vehicle_according_rules/1, move_enemies_according_cycles/1* and *update_resources/2* should be clear from their identifiers and basically update the state of the game in each iteration.

[3] It is a modification since the EA proposed in [20] is applied on static worlds without enemy agents i.e., basically it derives behavior rules to optimize a classical path finding problem.

As in the rest of all our offline EAs, we use this representation based on arrays not only to solve the posed problem but also to keep a trace of the movements executed by the vehicle and all the enemies in each instant of the game with the further aim of visualizing the simulation of the game[4]

Results. This approach solved 29 of the 60 worlds (i.e., 48.3%), a poor result although we also noted that this approach behaved much better on simple worlds (i.e., those not too heterogeneous) than on complex ones.

3.2 An Evolutionary Hybrid Solution

We also developed an Evolutionary Program (EP) based on the RBP procedure. Below we partially specify some new data types defined for the EP.

gen =(v : vehicle, en : ARRAY OF enemy);

individual = (path : ARRAY OF gen, length : $\mathbb{N}$, reason : state, fitness : $\Re$);

A *gen* is a tuple that provides information about the vehicle and the enemies in certain instant of the game and a *individual* is a 4-tuple that contains (1) a variable length sequence of genes that represents one of the possible path associated to one of the (possibly intermediate) states of the game[5], (2) the length of this path (i.e., number of stages or instants of the game played so far), (3) a value of termination (in case that the game is over), and (4) a fitness value that measures the path quality with respect to both target proximity and remaining resources amount. Note that, the last gen stored in the path (i.e., $ch.path[ch.length]$, for some $ch : individual$) contains information about the last instant of the game played so far. The EP schematic code is shown in Figure 3.

```
For currentGeneration ← 1 to max_generations
    For i ← 1 to pop_size
        Evaluate(pop[i],, α_{i_d,j_d});
        if better_chromo(better,pop[i]) then better ← pop[i];
    Mutate(pop);
```

Fig. 3. Simplified schema of the EP algorithm

The variable pop contains the individual population where `pop[i].path[1].v.pos` identifies the origin grid ($1 \leq i \leq$ pop_size), $\alpha_{i_d,j_d} \in W_n$ is as in the RBP, `better:individual` initially contains the best

[4] E.g., the vehicle information is stored in $v[0], v[1], v[2] \ldots$ and so on.

[5] In fact a chromosome represents the "story happened so far" i.e., if ch : $individual$, then $ch.path[i]$ and $ch.path[i-1]$ keep information about consecutive instants of the game, where an *instant of the game* is each of the discrete stages that produce some change in the game state. Thus i numbers each of the different game stages.

path calculated by the RBP, and the procedure `Evaluate/2` applies the RBP algorithm taking as origin the last gen kept in individual $pop[i]$, that is to say, it applies the RBP procedure taking $v[0] = pop[i].path[pop[i].length].v$ and $en[0] = pop[i].path[pop[i].length].en$ (see Figure 2). Observe that this means a trial to complete the path stored so far in pop[i] in order to reach the target. Of course, this EP is very dependant on the RBP algorithm and thus on the quality of behavior rules used.

The fitness is calculated as the inverse of the Euclidean distance between the position stored in the last gen of the chromosome and the destination position[6]. A function call `better_chromo(c1,c2)` defines a stratified fitness function that returns true if $c_1.fitness > c_2.fitness$, or $c_1.fitness = c_2.fitness$ and $c_2.path[c_2.length].v.res \leq_r c_1.path[c_1.length].v.res$ (i.e., c_1 keeps more resource values than c_2), and false otherwise, and where $(t_2, f_2, r_2) \leq_r (t_1, f_1, r_1)$ iff $t_2 > t_1$, or $t_2 = t_1$ and $f_2 < f_1$, or $t_2 = t_1$ and $f_2 = f_1$ and $r_2 \leq r_1$ (i.e., a lexicographic ordering). At the end of the execution, *better.path* contains the sequence of genes that compose the better path obtained, and *better.reason* indicates if the simulation was successful (i.e., destination reached) or not.

The mutation is a grid offset: for some individual $pop[i]$, one position m (with $2 \leq m \leq pop[i].length$), where $pop[i].path[m].v.pos$ differs from the origin location, is chosen randomly and transformed to an adjacent position (if necessary, this is linked with the gen $pop[i].path[m-1]$ via the addition of gens to the path to assure its continuity). Of course, the enemies have to be taken into account in this mutation process. Basically, the mutation process consists of replacing the gen $pop[i].path[m]$ by a new gen $g : gen$ where

$$
\begin{aligned}
&g.v.pos.val\_x \leftarrow pop[i].path[m].v.pos.val\_x \pm \{0, 1\}\\
&g.v.pos.val\_y \leftarrow pop[i].path[m].v.pos.val\_y \pm \{0, 1\}\\
&g.en \leftarrow \text{move\_enemies\_according\_cycles}(pop[i].path[m-1].en)\\
&\qquad \text{and also}\\
&pop[i].length \leftarrow m
\end{aligned}
$$

The last sentence shown above truncates the path stored in $pop[i]$ from the mutated position so that, in next iterations, this can be completed by procedure `Evaluate/2` as explained above. Note that this is basically a path reconstruction process.

Results. EP solved 45 worlds from 60 (i.e., 75%). Figure 3.2 shows the time difference between the executions of the RBP (dark color line) and the EP (light color line). The x-axis identifies the name of the distinct worlds whereas the y-axis indicates the time (in seconds) spent in the simulation execution. The gaps in the graphic indicate that the simulation finished unsuccessfully; note that EP solves more cases and takes less time than RBP.

[6] i.e., for some ch : $individual$, $ch.fitness$ contains the inverse of $\sqrt{(ch.path[ch.length].v.pos.val\_x - i_d)^2 + (ch.path[ch.length].v.pos.val\_y - j_d)^2}$.

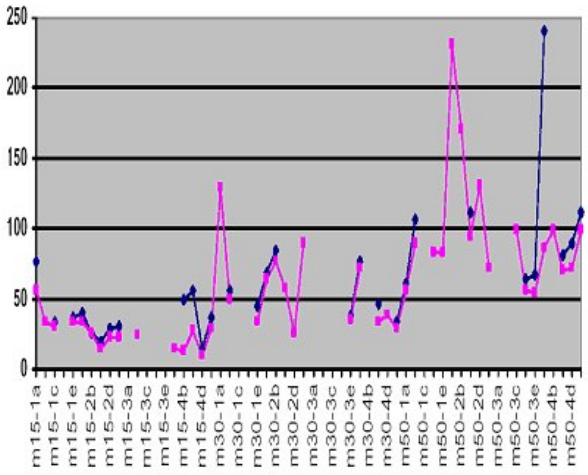

Fig. 4. Runtime comparison (measured in seconds) (RBP vs. EP)

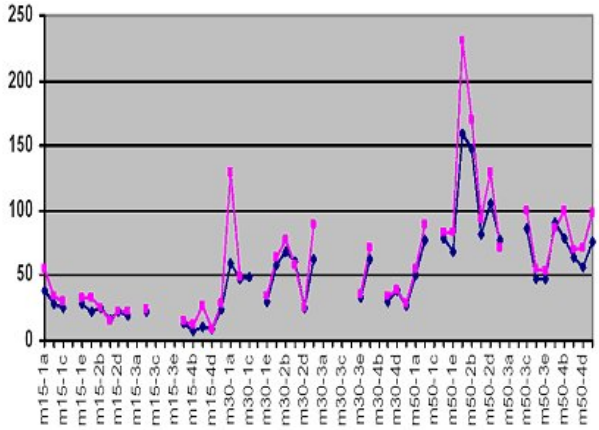

Fig. 5. Runtime comparison: Left (RBP vs. EP); Right (EP vs. EP+)

3.3 An Improved Evolutionary Solution

We modified EP by defining a new non-uniform mutation operator based on the number of remaining generations (i.e., *max_generations* − *current_generation*). This means that the gen to mutate is no more chosen randomly with the same probability for all the candidate genes (as done in the EP algorithm), but its probability to be chosen increases according to the relative position of the gene in the path: higher position, higher probability. As consequence, it is likely that the first part of the path will remain intact and the path will be reconstructed, via function `Evaluate/2`, from a nearer-to-the-target gen. This technique often gives good results as the differ-

ent paths stored in the population "differ less" in their initial segments than in their final ones. We call this algorithm EP+.

Results. EP+ improved EP notably: from 60 worlds, 40 (i.e., 66'6%) were improved, 15 (i.e., 25%) largely improved (we speak of an improvement about 85% or of reaching a solution in a world that previously was not solvable with EP), 4 cases (6'6%) made worse (i.e., they needed more time although it was only 1.09% more in the worst case) and 1 case (1'6%) exactly equal. Figure 5 shows a comparative graphic wrt. time between the EP (light color line) and EP+ (dark color line).

3.4 Hybrid (EP+)-A*

A classical approach for solving path finding problems to optimality is the well known A* algorithm. We applied the A* algorithm to a modified version of the game (that we call *static*) where the enemies do not execute patrols but they maintain a fix position during the game simulation. The heuristic function used was the Euclidean distance from a grid to the target one. To the real cost, we added the time cost to cross a grid. As it was expected, we obtained an optimal solution to the static version of the game[7].

We then proposed the (EP+)-A* algorithm that hybridizes the EP+ and A* algorithms to combine the excellent behavior of A* in static worlds with the flexibility of the EP+ in dynamic worlds. Initially, (EP+)-A* considers, as starting point, the optimal static solution calculated by using A* in the static version, and then applies an EP+ modification consisting of replacing, in the *Evaluate* call in Figure 3, the application of the RBP algorithm by the A* algorithm to re-construct the path as explained in Section 3.2. The path variations obtained by this modification generate better paths.

Results. This proposal gave excellent results as we obtained a successful simulation in 60 worlds (i.e., 100%). Figure 6 shows the time comparison between the EP+ algorithm (dark color line) and this new hybrid algorithm (EP+)-A* (light color line). Note that, in most of the cases, the time is also optimized. There is just one draw case (identified as m30-4e) wrt. time; however the fuel resource was optimized using the hybrid algorithm (EP+)-A*.

We also created 10 new complex worlds with very extremely difficult cases on which the RBP, EP and EP+ algorithms did not find a solution. However, the (EP+)-A* algorithm returned a solution in all these cases.

Although successful our offline proposals were computationally costly (wrt. time) to be employed online in a real game.

4 Online Evolution for the Game

The experience accumulated so far with our offline EA versions highlighted that we had to follow an alternative strategy for using EAs online in a real

[7] In fact, the static version of the game consisted in a classical path finding problem.

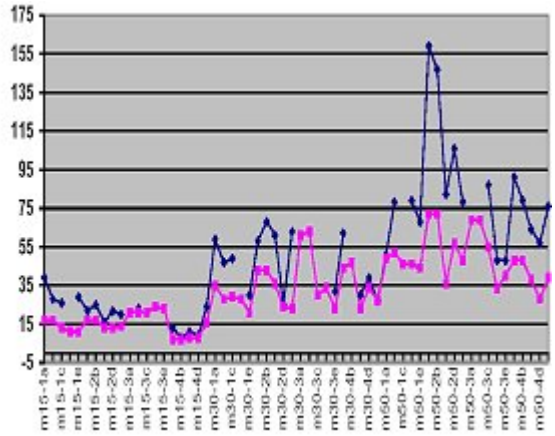

Fig. 6. Runtime comparison (measured in seconds): EP+ vs.(EP+)-A*

game. Firstly, we replaced the offline game simulation by an online version where we let a human player control the vehicle and decide its actions in real time at the same time that each of its enemies is driven by a very simple evolutionary algorithm whose purpose is to avoid, as soon as possible (i.e., by minimizing the game time), that the (player-controlled) vehicle reaches the target. In each stage of the game, each enemy has information about its (Euclidean) distance to both the human player (i.e., the vehicle) and the target position. This information is then translated into three degrees that depend on the world size. For a given world W_n and distance d, the degrees are: `near` ($d \in [0.0, n/6.0]$), `medium` ($d \in (n/6.0, n/3.0]$) and `far` otherwise. As consequence, each enemy receives information only in three discrete degrees. For each enemy, there are two targets (i.e., the player position and the destination location) so that there exist only 8 combinations of rules in the form of: *if human is at degree*$_1$ *and target at degree*$_2$ *then the enemy does some action A*. To accelerate the calculus we only offer 4 possible actions (enough for our action game) to be executed by the enemies. These actions are shown below:

- shot the player vehicle;
- attack the player vehicle (i.e., advance directly to destroy it by a collision);
- move nearer to the vehicle destination (in order to protect it) and,
- move randomly (i.e., move up, down, left or right). This action is added to make the enemy behavior less predictable and introduce intentional enemy errors in order to increase the user satisfaction (see Section 5).

A chromosome represents one of the possible actions to be executed by the enemy and is a value of type $action = \{shot, attack, move\_destination, random\_move\}$. Observe that actions can be coded using just 2 bits as there are only four possibilities. The simplified evolutionary code that guides each enemy is shown in Figure 7 where $enemy : chromosome$ is initialized to a random move and, to the end

```
enemy ← random_move;
InitializePopulation(pop);
(degree_v, degree_d) ← CalculateDistance(α_e, α_v, α_d, world_size);
For generation ← 1 to max_generations
    For i ← 1 to pop_size
      Evaluate(pop[i],enemy, degree_v, degree_d, res);
      if fitness(pop[i], degree_v, degree_d, res) < fitness(enemy, degree_v, degree_d, res)
            enemy ← pop[i];
      .....Apply the genetics operators.....
```

Fig. 7. On-Line evolutionary algorithm to control each opponent

of the algorithm execution, *enemy* will contain the action to be executed by the enemy; $\alpha_e, \alpha_v, \alpha_d \in position$ represent respectively, the position of the enemy, the vehicle and the destination; *pop* contains the chromosome population and the function $CalculateDistance$ obtains the degree information of the enemy position wrt. the player and destination positions as explained above in the form of a tuple $(degree_v : degree, degree_d : degree)$ where $degree = \{near, medium, far\}$.

The function `Evaluate` receives an individual and uses the distance degree information and the resource amount of the vehicle (stored in variable $res : resources$ where $resources$ is defined as in Section 3) to compute the best action to be executed by the enemy. The function `fitness/1` calculates the quality of a chromosome by considering all the possible situations considering again both the distance-degrees and the remaining resource amount. This is done by given some weights to the best actions and penalizing the worst situations for the enemy. The lowest fitness, the best chromosome.

We use a classical one-point crossover on the actions and the swap operator for mutation. The parameters used were the following: pop_size=5, max_generations=10, crossover_rate=0.9, mutation_rate=0.01.

Surprisingly the game works fine in real time[8]!!. We observe two key points for the success: the low number of generations and the small size of the population. Figures 8 and 9 visualize two different states of the game execution `in real time`. The game was implemented using the standard library OpenGL that provides advanced 2D/3D graphics capabilities.

5 Conclusions and Future Work

This paper highlights the power of evolutionary techniques to be used to program action games that also demand a computationally expensive (graphical)

[8] Note that now, it is not necessary to register the sequence of movements, as done in our offline EAs, as the game visualization is done online together with the execution of our EA!.

Fig. 8. The enemy shots the vehicle

Fig. 9. Game over with vehicle destruction

simulation. We have shown, by proposing and comparing different EAs specifically coded to control autonomous agents in the simulation of a war game, that EAs are adequate for offline use. This is an expected result as EAs are time-consuming to develop and resource intensive when they are in execution. More surprisingly, we have also demonstrated, on reverse to the general belief, that EAs (designed to control the opponent intelligence) can be successfully executed online (in real time!!) during game simulation: the key for this online use consists of evolving much quicker than offline evolution. To do this, we have simplified the set of actions to be executed by the opponents in order to

reduce drastically the search space and to accelerate the computation process. In this line, [17] proposes different methods to do online coevolution of agents.

Now, the question to answer is: should we use evolutionary algorithms for game programming?. In the following we briefly discuss this question.

`No, we should not.` Many game programmers suggest that EAs require too much CPU power and are very slow to produce useful results.

`Yes, we should.` Genetic (and evolutionary) algorithms appear as one of the most promising game AI techniques [9]. As it is well known, evolutionary algorithms are very good to find a solution in complex or poorly understood search spaces, and require a considerable computational effort that slows down the optimization process. Even although this is true, we defend the offline evolutionary learning that is not too dependant on the execution time.

Moreover, as it has been shown in the paper, evolutionary algorithms can be employed successfully on real-time. To do so, the evolution must happen very much quicker than offline evolution. We have shown how, by minimizing the set of actions to be executed by the opponent, this is possible. Of course, one may argue that, if there are only four actions, why should there be an EA to calculate the following action to be executed by each enemy if an easy enumeration algorithm would find the optimal solution in more real time? This is true although another issues have to be considered as for example the random component and the intentional errors of the opponent that make the game more attractive to the user. As argued in [2], it is important to provide human intelligence to the opponent to obtain a better simulation of the reality, but this does not correspond to make opponent necessarily smarter as, after all, the player is supposed to win.

Game development demands more and more research in a number of very "hot" areas such as: (1) *Multiplayer computer games* [21], (2) *Mobile game development* (i.e., games developed to run on mobile or cellular phones) and (3) *Simulation of army forces combat strategies*. With respect to (1), EAs could be applied to solve optimization problems focused in the reduction of networking resources requirements in distributed interactive real-time applications. With respect to (2), the proliferation and popularization of mobiles phones have produced the interest of game industry. This field demands professional game developers and active research on both hardware and software. Although the interest maybe in the local embedding of games into the phones, the area of downloadable games from the mobile phone seems to be particularly interesting [22]. With respect to (3), we think that EAs may be applied offline to simulate the emergent behavior of individuals (e.g., soldiers) in a combat between different army forces. Some AI techniques have already been applied in this sense in Hollywood films.

In any case, we claim that EAs are a unexplored AI technique in game programming and that they deserve particularly a wider study to be applied as a game AI technique. Moreover, we also assure that the future of game development is promising, and this paper encourages the cooperation between

the communities of the academic computer science researchers and the game industry programmers.

References

1. M. DeLoura (editor): Game Programming Gems. Editorial Charles River Media, INC., Rockland, Massachussetts (2000)
2. Lidén, L.: Artificial stupidity: The art of intentional mistakes. In: AI Game Programming Wisdom 2, Charles River Media, INC. (2004) 41–48
3. Johnson, D., Wiles, J.: Computer games with intelligence. Australian Journal of Intelligent Information Processing Systems **7** (2001) 61–68
4. James, G.: Using genetic algorithms for game AI. GIGnews.com (2004) `http://www.gignews.com/gregjames1.htm`.
5. Wong, T., Wong, H.: Genetic algorithms (1996) `http://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol1/tcw2/ article1.html`.
6. Buckland, M.: AI Techniques for Game Programming. Premier Press (2002)
7. Sweetser, P.: AI in games: A review. Available at `http://www.itee.uq.edu.au/~penny/publications.htm` (2002)
8. Woodcock, S.: Game AI: The state of the industry. (August, 1999, `http://www.gamasutra.com/features/19990820/game_ai.htm`)
9. Rabin, S.: Promising game AI techniques. In: AI Game Programming Wisdom 2, Charles River Media, INC. (2004) 15–28
10. Buckland, M.: Building better genetic algorithms. In: AI Game Programming Wisdom 2, Charles River Media, INC. (2004) 649–660
11. Sweetser, P.: How to build evolutionary algorithms for games. In: AI Game Programming Wisdom 2, Charles River Media, INC. (2003) 627–638
12. Spronck, P., Sprinkhuizen-Kuyper, I., Postma, E.: Improving opponent intelligence through offline evolutionary learning. International Journal of Intelligent Games & Simulation **2** (2003) 20–27
13. Dalgaard, J., Holm, J.: Genetic programming applied to a real time game domain. Master thesis, Aalborg University - Institute of Computer Science, Denmark (2002)
14. Alife Games: (`http://alifegames.sourceforge.net/bSerene/index.html`)
15. Sweetser, P.: (`http://www.itee.uq.edu.au/~penny/commercial_AI.htm`)
16. van Waveren, J., Rothkrantz, J.: Artificial player for Quake III arena. International Journal of Intelligent Games & Simulation **1** (2003) 25–32
17. Demasi, P., de O Cruz, A.: Online coevolution for action games. International Journal of Intelligent Games & Simulation **2** (2003) 80–88
18. Game, A.C.: (`http://tracer.lcc.uma.es/problems/index.html`, 2002)
19. Eiben, A., Schoenauer, M.: Evolutionary computing. Information Processing Letters **82** (2002) 1–6
20. Cotta, C., Troya, J.: Using a hybrid evolutionary-a* approach for learning reactive behaviours. In et al., S.C., ed.: Evo Workshops. Number 1803 in LNCS, Edinburgh, Scotland, Springer (2000) 347–356
21. Smed, J., kaukoranta, T., Hakonen, H.: Networking and multiplayer computer games - the story so far. International Journal of Intelligent Games & Simulation **2** (2003) 101–110
22. Mencher, M.: The future of game development: new skilss and new attitudes. GIGnews.com (2004) `http://www.gignews.com/gregjames1.htm`.