Evolved Creative Intelligence for Computer Generated Forces

Linus J. Luotsinen, Farzad Kamrani,
Peter Hammar and Magnus Jändel†
FOI - Swedish Defence Research Agency
SE-164 90 Stockholm, Sweden
Email: {linluo, farkam, petham}@foi.se

Rikke Amilde Løvlid
FFI - Norwegian Defence Research Establishment
NO-2007 Kjeller, Norway
Email: rikke-amilde.lovlid@ffi.no

***Abstract*—This paper provides an example of using genetic programming for engendering computational creativity in computer generated forces, i.e. simulated entities used to represent own, opponent and neutral forces in military training or decision support applications. We envision that applying computational creativity in the development of computer generated forces may not only reduce development costs but also offer more interesting and challenging training environments.**

In this work we provide experimental results to strengthen our arguments using a predator/prey game. We show that predator behavior created by a computer, using genetic programming, surpasses predator behavior manually programmed by humans and argue that the sparse automatically generated code is unlikely to be generated by a human and therefore can be considered as a good example of computational creativity. Although the experiments are not conducted in a real-world training simulator they provide valuable insight that exemplifies the opportunities and the challenges of computational creativity applied to computer generated forces.

I. Introduction

Computer Generated Forces (CGF) are automated actors within simulations of military scenarios. Such simulations are used for the purpose of guiding military operations, as a virtual testing ground for military equipment, and for training military personnel and civilian relief workers [1]. CGFs are often used in large-scale military staff exercises that are otherwise too expensive to populate using human role-players. CGFs with intelligent behavior are useful for making exercises more realistic, thereby enhancing the value of training as preparation for real-life action. Smart CGFs can also help commanding officers to prepare for expecting the unexpected and gracefully handling the collapse of well-prepared plans in confrontation with intelligent opposition. A CFG could represent an individual soldier, a unit, a vehicle, or any other congregation of military units and logistic resources.

Despite the many applications and uses of CGFs they rarely exhibit realistic behavior [2], [3]. This is particularly true in tactical military training applications where CGFs often represent individual soldiers or vehicles that autonomously have to reason, act, interact and even collaborate with players/trainees and other CGFs.

† Deceased

In this work we introduce, apply and evaluate techniques used within the computational creativity research field to gain insight into how such techniques can be used to reduce CGF development efforts/costs and to improve the behavioral aspects of CGFs.

II. Background

A. Darwinian Evolution and Memes

Quantum computation luminary David Deutsch asserts [4] that the only true source of creativity is Darwinian Evolution (DE). The creativity of biological DE is obvious in the many forms and variations of living creatures that doubtless have come about by DE (doubtless according to the authors and the dominating and evidentially well-supported school of Darwinism in biology).

DE means that selectable items (the genotype) are replicated by a mechanism including some randomness and that selection for replication includes evaluation (of phenotypes) according to a fitness criterion corresponding to the performance of the phenotype in a test environment. The genotype can be viewed as a code containing instructions for how to build the phenotype, although it is known that environmental factors also are important for the structure of biological phenotypes. The phenotype represents the physical properties and behavior of an individual [5]. It is interesting to note that all DE depend on the presence of some machinery for decoding the genotype and based on that build the phenotype. The brilliant invention of biological life is that this translation machinery is part of the phenotype and that the phenotype archives the genotype.

Memes are pieces of information that replicate by spreading between communicating members of a society [6]. Humans carry memes such as gestures (waving), religious ideas (creationism), scientific ideas (DE), cultural ideas (music, sports) etc. Deutsch argues [4] that evolution of human cultures can be explained as DE of human-created memes eventually resulting in endless streams of innovation.

B. Genetic Programming

Genetic programming (GP) [7], [8] is DE applied to a population of computer programs. Variation is engendered by applying simulated reproduction and mutation to the population of programs. Evaluation is done by testing the performance

978-1-5090-1897-0/16/$31.00 ©2016 IEEE

of each individual program against a fitness criterion designed by a human.

In our case fitness is performance with respect to controlling a CGF in a simulated environment. Snippets of code (genes) from high-performing programs propagate to the next generation of programs in the GP reproduction process. The code of each individual controls the corresponding CGF behavior in the test environment. The program code in GP is hence the genotype and behavior in the simulation is the phenotype.

The GP algorithm needs some human input. A human must design input and output formats and provide a library of functions that can be used by the GP algorithm to build computer programs. A human also must design the fitness function so that it encapsulates all requirements on a solution.

C. Computational Creativity

Computational Creativity (CC) is an emerging branch of computer science focusing on making creative computer programs and on understanding how creativity can be automated [9]. There is no universally accepted definition of CC and no hard criterion against which an ostensibly creative program can be tested. We apply a human-centric definition of CC, which means that a program is creative if a human observes or experiences that the program contributes or outputs creative ideas and solutions to the task at hand. It is generally agreed that creativity, or more precisely the products of creativity, must be novel and useful. Creativity can hence not be achieved by repeating ideas that previously have been found to be useful and it can also not be completely random, since random ideas are unlikely to be of use in a practical context.

III. Methods

In this paper, we study two different approaches used for behavior modeling in simulations:

- the traditional approach, in which a domain expert provides suitable tactics and doctrine for the CGF to be modeled by a programmer and,
- the computational creativity or machine learning approach, in which a generic algorithm such as GP is used to automatically generate a behavior model from observed behavior or self-play [10], [11].

In the results section, we compare the approaches with respect to implementation effort and behavioral performance. The remainder of this section lays out the general rules of the simulation, the method used to acquire hand-written programs, and the implementation of the genetic programming approach.

A. General Rules of the Hunting Game

The CGF behavior is implemented in the setting of a simulated hunting game, where a wolf preys on sheep. The game is played on a board of 800*800 positions, each represented by a pixel. Distances are Euclidean in units of pixels. Time is measured in units of game ticks. A game tick is one internal cycle of the game during which all participants make a move and after which the screen is refreshed. The objective of the wolf is to kill all of the sheep in as few time steps (game ticks) as possible. The game ends when the wolf has killed all the sheep or after a preset time.

1) Wolf Behavior: The wolf kills a sheep based on a proximity condition. It will kill sheep within a distance of less than 10 units. Note that the wolf can kill several sheep within one tick, meaning wholesale destruction of all sheep caught within a radius of 10 units from the wolf. The speed of the wolf is constant at 6 pixels per tick, and its constant angular velocity when turning is 0.08 radians per tick. The wolf-control algorithms discussed here have no means for adapting the speed or angular velocity.

The wolf's sensors pinpoint the center of the herd. Its turning behavior is controlled by a program. The inputs to the program are the coordinates of the center of gravity of the herd. The output is one of three actions: *GO_STRAIGHT_AHEAD*, *TURN_RIGHT* and *TURN_LEFT*. *TURN_RIGHT* means that the wolf turns right with the maximum allowed angular velocity. In the next tick the control algorithm decides if the turn will continue or not. While pursuing the herd, the wolf will kill the sheep that happens to fall within its strike radius. The wolf will, however, not hunt isolated sheep unless an isolated sheep falls in its killing zone while the wolf heads for the herd center. This is because no information about individual sheep coordinates is available to the wolf control algorithm.

2) Sheep Behavior: Sheep behavior is a variant of flocking behavior [12] in which each flocking individual displays automated behavior controlled by few simple rules, including short range repulsion to avoid collisions with immediate neighbors, and cohesive behaviors causing each individual to follow companions in the direct neighborhood. Applying the flocking rules result in a herd that moves collectively as expected from a believable herd of sheep. Note that individual sheep do not avoid the wolf in an optimal or very intelligent way. Flocking in nature as well as in games offers safety in numbers rather than optimal predator-avoidance for each individual member of the herd.

The algorithm in [12] has been updated in [13] for the purpose of taking into account that sheep in contrast to birds and fish move in a 2D geometry and that sheep avoid predators which means that a fourth behavior "escape" has been added to the three basic behaviors of Reynolds' model [12]. Sheep behavior in the present model is represented by the total velocity of the sheep as given by,

$$\begin{aligned}\mathbf{v}_s = k_{coh}(1+\sigma(r)k_{wcoh})\mathbf{coh}(s) + k_{sep}\mathbf{sep}(s)\\ +k_{alig}(1+\sigma(r)k_{walig})\mathbf{alig}(s) + k_{esc}\mathbf{esc}(s),\end{aligned} \tag{1}$$

in which bold symbols are vectors, all vectors are velocities, and the four vectors on the right-hand side are due to cohesion, separation, alignment and escape behaviors respectively. The variable r is the distance between the sheep at hand and the wolf, while $\sigma(r)$ is a sigmoid function according to:

$$\sigma(r) = \frac{1}{\pi}\tan^{-1}\left(\frac{300-r}{20}\right) + 0.5. \tag{2}$$

This term ensures that sheep behavior is influenced by the distance to the wolf in a reasonable way so that escape

behavior dominates when the wolf comes very close and the herd behaves as a normal herd of grazing sheep when the wolf is farther away than 300 pixels. Consider that the escape behavior component, **esc**, intrinsically depends strongly on the relative position between the sheep and the wolf and hence is not multiplied by the sigmoid function.

The constants k_{coh}, k_{wcoh}, k_{sep}, k_{alig}, k_{walig} and k_{esc} in Equation 1 are used for tuning sheep behavior and are given in Table I. When $\sigma(r)$ takes a large value because the wolf is close, cohesion and alignment behavior is strengthened so that the sheep at hand will mimic the behavior of neighboring sheep more than if $\sigma(r)$ is small which is the case when the wolf is far away. Note that we are not experts on sheep behavior so we do not know if this is correct from an ethology point of view.

TABLE I: Constants used in Equation 1.

Parameter	Setting	Parameter	Setting
k_{coh}	0.4	k_{wcoh}	40.0
k_{alig}	0.3	k_{walig}	2.0
k_{esc}	900	k_{sep}	1100

Flock cohesion means that all sheep are attracted to the center of the flock according to,

$$\mathbf{coh}(s) = \frac{\mathbf{p}_{centroid} - \mathbf{p}_s}{|\mathbf{p}_{centroid} - \mathbf{p}_s|}, \quad (3)$$

where $\mathbf{p}_{centroid}$ and $\mathbf{p}_s$ are the vector positions of the flock centroid and the sheep at hand, respectively.

Collision avoidance between sheep is guaranteed by the separation term in Equation 1, calculated by

$$\mathbf{sep}(s) = \sum_{i \neq s}^{n} \frac{\mathbf{p}_s - \mathbf{p}_i}{|\mathbf{p}_s - \mathbf{p}_i|} (|\mathbf{p}_s - \mathbf{p}_i| + \varepsilon)^{-2}, \quad (4)$$

where s is the index of the sheep for which the velocity is computed, n is the total number of sheep, and ε is a small number to avoid division with zero. All sheep in the herd contribute to the sum in Equation 4. An explicit separation term with the inverse quadratic form is necessary to avoid too cohesive herds in which case the wolf could exterminate the entire herd in one fell swoop.

Neighboring sheep adapt their speed to each other via the alignment mechanism,

$$\mathbf{alig}(s) = \sum_{i \in N_{nb}(s)} \mathbf{v}_i, \quad (5)$$

in which all sheep in the neighbor group $N_{nb}(s)$ contribute, $\mathbf{v}_i$ is the velocity of the i^{th} member of the neighbor group. The neighbor group comprises all sheep within a radius of 50 pixels from the sheep, s, for which we are refreshing the velocity.

In the presence of a predator, a strong repulsive component is added to the sheep behavior before calculating the velocity of each individual sheep, resulting in the flock avoiding the wolf [13] according to,

$$\mathbf{esc}(s) = \frac{\mathbf{p}_s - \mathbf{p}_{wolf}}{|\mathbf{p}_s - \mathbf{p}_{wolf}|} \left(\frac{|\mathbf{p}_s - \mathbf{p}_{wolf}|}{10} + \varepsilon \right)^{-2}, \quad (6)$$

in which $\mathbf{p}_{wolf}$ is the position of the wolf, $\mathbf{p}_s$ and ε are defined as before.

B. Manual Programming

For the human-written programs we have asked several experienced programmers to develop the most efficient and intuitive wolf-behavior they could think of. The programmers were first introduced to the game and its objectives. Next, we provided a template program to describe how to read/interpret wolf sensor data and how to invoke wolf actions. Implementation time was limited to 2 hours, which is also the stopping criteria for our GP implementation.

C. Genetic Programming

In this study we used the ECJ-toolkit[1] which implements GP algorithms described by [7]. Unless otherwise stated, the GP algorithm used in this work follows the default parameter settings as defined by the ECJ-toolkit.

1) Program Representation: In GP a program is typically represented using a tree-structure that consists of terminal and function nodes. Terminal nodes represent leaves in the program tree and consist of either constants or variables such as the location and orientation of the wolf. Function nodes take as input one or more nodes, such as a leaf or the result of another function, and return a single value that in turn can be fed into another function node. In this work we have used a strongly typed GP algorithm in which the nodes in the program tree have constraints associated to their inputs and output. The nodes available to the GP are presented in Table II.

TABLE II: List of all terminal (T) and function (F) nodes used by the GP algorithm.

Name	Node	Description
x_w	T	x-coordinate of wolf.
y_w	T	y-coordinate of wolf.
α_w	T	Orientation of wolf in degrees.
x_h	T	x-coordinate of herd centroid.
y_h	T	y-coordinate of herd centroid.
π	T	The constant π in radians.
$-$	F	Subtraction.
$atan2(x, y)$	F	Calculates the radian angle θ from the conversion of rectangular coordinates (x, y) to polar coordinates (r, θ).
$deg2rad(a)$	F	Converts a from degrees to radians.
$rad2deg(a)$	F	Converts a from radians to degrees.
$dist(a, b)$	F	Calculates the shortest angular distance in range $[-\pi, \pi]$ between radian angles a and b.
$norm(a)$	F	Normalizes radian angle a to range between $[-\pi, \pi]$.

An example program tree, manually implemented by one of our programmers, is illustrated in Figure 1. This program translates to a behavior where the wolf always turns and strikes towards the center of the herd.

[1] https://cs.gmu.edu/~eclab/projects/ecj/

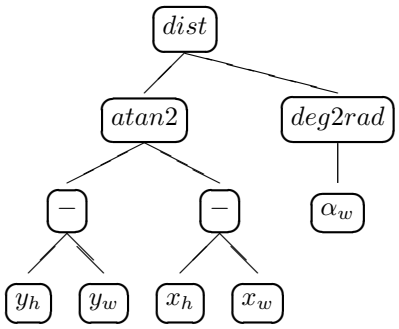


Fig. 1: Program, developed by a human programmer, representing an intuitive behavior where the wolf turns and strikes towards the center of the herd. A GP tree-structure is used to visualize the program. Leaf nodes, at the bottom of the tree, represent input variables from the wolf's sensors and the root node, at the top of the tree, represents the return value used to determine the wolf's desired direction. Table II explains the meaning of each node in the tree.

2) Fitness Function: The fitness function used in our experiment evaluates the performance of each program by calculating a weighted sum of the average distance between the wolf and the centroid of the herd, c_i, and the number of living sheep at the end of the simulation, c_j, as follows:

$$fitness = c_i * w_i + c_j * w_j, \tag{7}$$

in which, w_i and w_j are parameters (weights) for tuning the fitness function,

$$c_i = \sum_{t=0}^{T_{end}} \frac{\sqrt{(x_w(t) - x_h(t))^2 + (y_w(t) - y_h(t))^2}}{T_{end} * maxDistance}, \tag{8}$$

and

$$c_j = \frac{numLivingSheep(T_{end})}{numSheep}. \tag{9}$$

In Equations 8 and 9, $maxDistance$ is the maximum distance between two points on the game board, T_{end} is the number of game ticks processed by the end of the simulation, and $numLivingSheep(T_{end})$ is the number of living sheep by the end of the simulation. (x_w, y_w) and (x_h, y_h) are the x-y coordinates of the wolf and the herd centroid, respectively. Fitness is a value that varies between 0 and 1. Lower fitness value represents a more efficient hunter than a higher fitness value.

3) Experimental Setup: The GP algorithm used here was configured to use a population of 1000 programs and to stop its search after 10 generations. The GP algorithm uses an elitist approach which means that the most fit program of a generation is always included in the next generation. The aforementioned fitness function was initialized as defined in Table III. For each fitness evaluation the simulator executed at most 600 ticks. The weights w_i and w_j were set to reward programs, or wolf behaviors, capable of killing sheep as opposed to herding them. We empirically selected these configuration values to ensure that the GP algorithm would be able to finish within 2 hours, which is also the implementation time given to the human programmers.

TABLE III: Parameter settings for the fitness function.

Parameter	Setting	Description
T_{max}	600	Maximum simulation ticks to execute.
$numSheep$	46	Number of sheep at start of simulation.
$maxDistance$	$\sqrt{2} * 800$	Maximum distance between two objects on the game board.
w_i	0.1	Weight of average distance fitness component.
w_j	0.9	Weight of killed sheep fitness component.

IV. Results

Let us now compare the wolf behavior developed by skilled human programmers with the wolf behavior developed by the computer using GP.

The code or program of the GP-evolved wolf behavior is shown in Figure 2. In contrast to the best human-generated program in Figure 1, in which the wolf goes for the center of the herd, we have not found any easy, intuitive, explanation for the GP-evolved behavior. Most programmers that we have asked to review the GP-evolved program consider it to be poorly programmed. Yet, as we shall discuss next, it performs better than the human-generated program in simulations.

The fitness plot in Figure 3 shows the fitness value of the best GP-program, that is, the one with the lowest fitness value, for each generation. From the figure we observe that the best program (i.e. Figure 2) was found in the 7^{th} generation. We also observe major improvements in fitness, or innovation, in generation 3 as well as in generation 6 and 7. Note also, as a result of the elitist GP approach, that the fitness value from one generation to the next never worsens. It either improves or remains constant.

A. Comparing Human and GP-generated Code

We tested both variants of wolf-behavior code in the simulator using identical starting conditions, finding that the best human-programmed code, see Figure 1, killed all the sheep in 1717 game ticks while the machine-generated code in Figure 2 required only 953 cycles to kill all sheep, thus outperforming the human-generated code. Figure 4 plots the performance with respect to number of living sheep for all implementations over time. In this plot we observe that the GP-program, unlike the human-generated code, is able to efficiently and consistently kill sheep no matter the size of the herd.

Since the code of Figure 2 is incomprehensible in its strategy and outshines the code of Figure 1, which implements the simple strategy of turning towards the center of the herd, the former code could be construed as being creative in that it is both novel (incomprehensible and breaks implicit rules for quality in programming) and provides value (kills sheep faster than the competition).

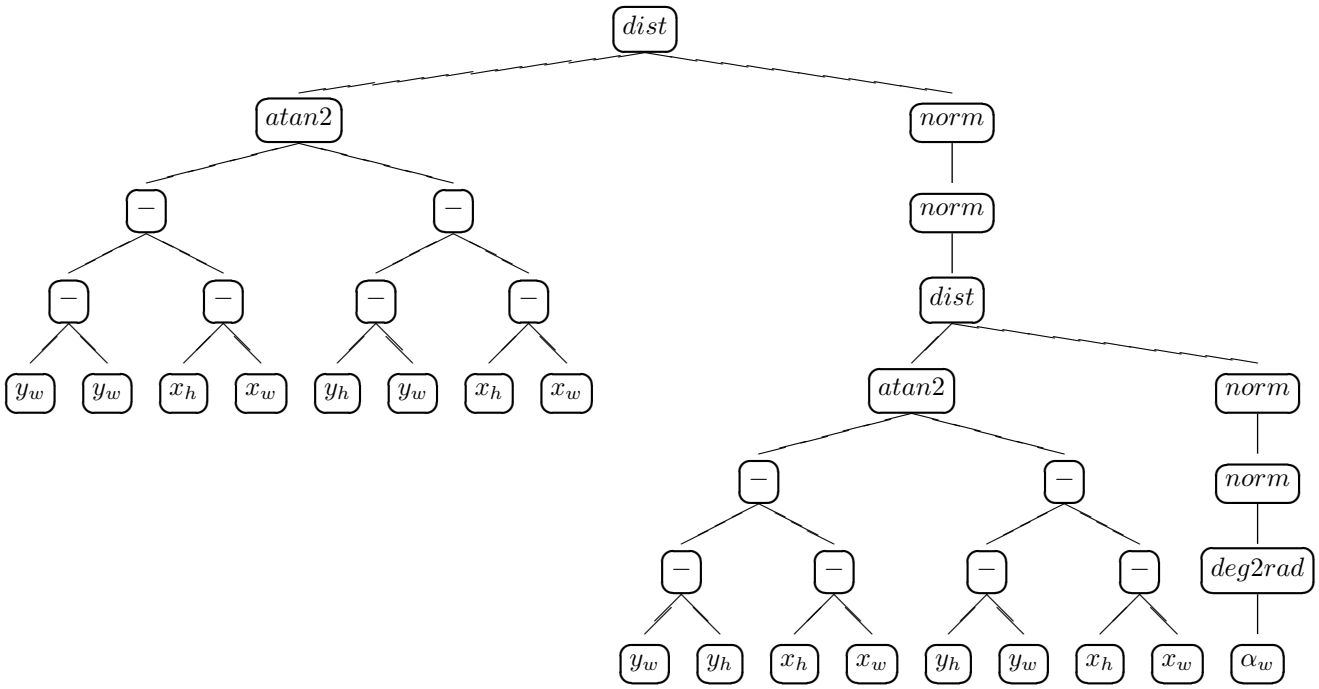


Fig. 2: Wolf behavior as evolved by GP. Refer to Table II for an explanation each node in the tree.

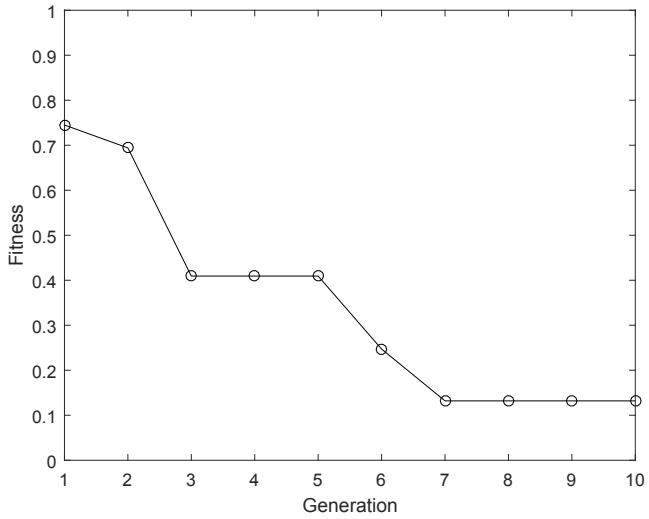


Fig. 3: Fitness of the best GP-program within each generation.

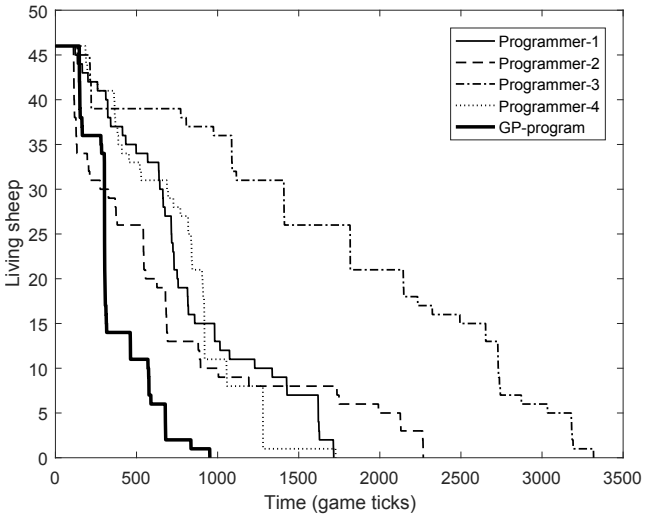


Fig. 4: Performance plot of all implementations over time.

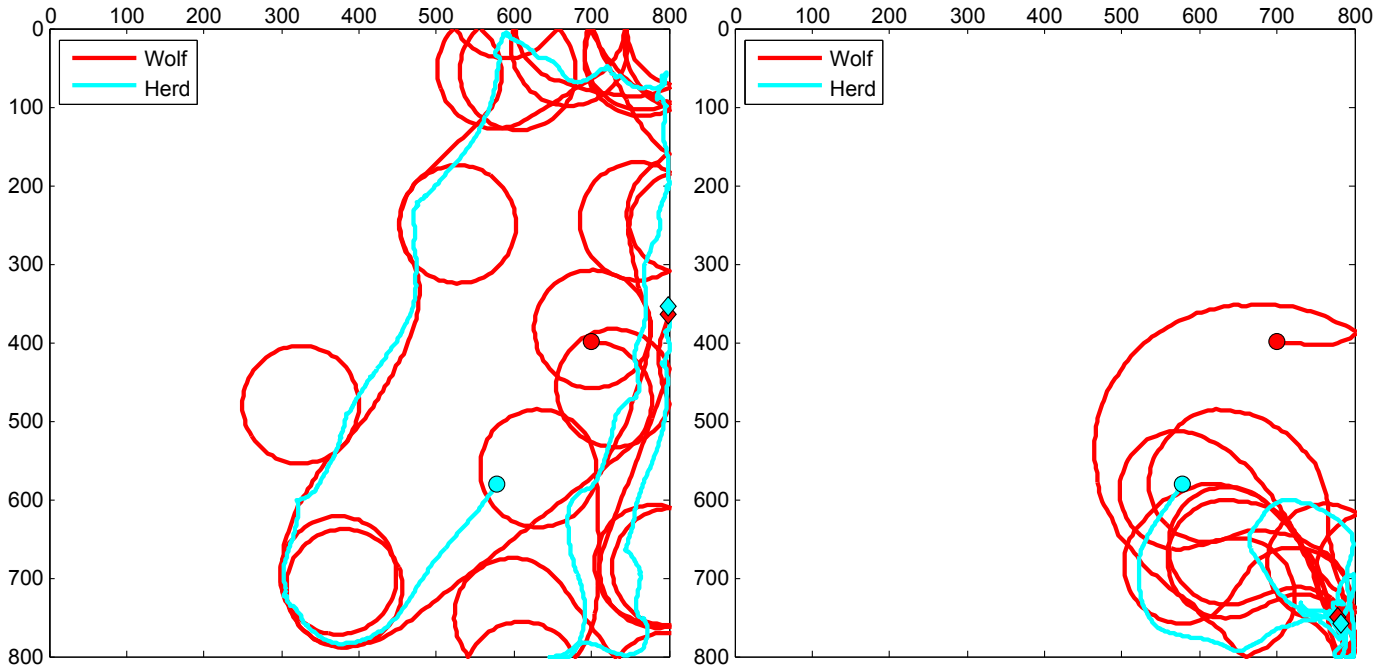
B. Emergent Intelligence

In Figure 5 we use trace-plots to render the behavior of the best human-programmed code and the GP-code. The trace-plots where generated using the same starting positions of the wolf and the sheep. Figure 5b shows that the GP-wolf herds the sheep towards the corner while occasionally raiding the concentrated flock. The best GP-wolf is a curious combination of white-fanged killer and placid herding dog. It herds the sheep to concentrate near the corner and intermittently strikes thereby finding many sheep within its killing zone. This deadly combination has emerged spontaneously from the evolutionary process and we think that this is a good example of emergent creativity.

We admit that the trace plot of the manually programmed wolf in Figure 5a also displays herding behavior. The GP-wolf is, however, much faster in driving the herd into a corner and can as a result exterminate the sheep within a shorter time.

C. Generalization

The results presented so far were created using the same starting positions of the wolf and the sheep. That is, although these experiments provide valuable insight, they do not measure how well the different implementations generalize and



(a) Trace plot of best manually programmed wolf.

(b) Trace plot of GP-evolved wolf.

Fig. 5: Trace plots illustrating the wolf's and herd's (centroid) movement behavior. Circles and diamonds indicate starting and ending positions respectively.

perform using different starting conditions. In this section we measure and compare the generalization capabilities of each implementation by randomly initiating the wolf's starting position in the simulator. We then execute each implementation in the simulator 100 times and measure the number of game ticks each program requires to kill all sheep. We limited the simulator to execute at most 7200 game ticks. Results from this experiment are summarized using descriptive statistics as presented in Table IV and using a box plot as illustrated in Figure 6.

From Table IV it is clear that the GP-program on average performs better than any other implementation. Furthermore, looking at the standard deviation (Std) and the confidence interval (CI 95%) we can conclude that the GP-program is also the most robust and stable implementation. Noteworthy is also that the GP-program, at least once, were able to kill all sheep in 438 game ticks at a rate of 0.11 kills/tick. In the worst case, the GP-wolf killed all sheep in 2664 game ticks which is similar to the average performance of Programmer-3 and Programmer-4 and significantly better than the average performance of Programmer-2.

Looking at the box plot in Figure 6 we can also conclude with 95% confidence that (since the notch of the GP-program does not overlap with any other notches) the true median of the GP-program differs from the medians of the manual implementations. From the box plot we can also observe that the worst performing wolf-behavior was written by Programmer-2. Programmer-3 and Programmer-4 performed similarly. The best human program (see Figure 1) was written by Programmer-1.

TABLE IV: Descriptive statistics measuring the number of game ticks each program requires to kill all sheep using data collected from 100 simulations per implementation. CI 95% represents the confidence interval using a 95% confidence level.

Program	Mean	Std	Min	Max	CI 95%
Programmer-1	1934.28	500.94	1098	3811	±99.40
Programmer-2	4281.12	2114.09	1246	7200	±419.48
Programmer-3	2641.48	688.67	1121	4783	±136.65
Programmer-4	2735.54	799.07	826	6401	±158.55
GP-program	1444.71	369.14	438	2664	±73.24

V. Discussion

Our choice of rules for sheep behavior is based on experimentation and on selecting rules and parameters to fulfill our intuitions about likely sheep flocking behavior.

We confess that our intuition about sheep could be wrong, since all of the authors are city boys/girl and not professional sheepherders/shepherdess. We have not attempted to include the latest scientific research with respect to sheep and wolf behavior in this paper. We are well aware that the game is quite crude from an ethology point of view and is hence not an accurate description of naturalistic sheep's and wolf's behavior.

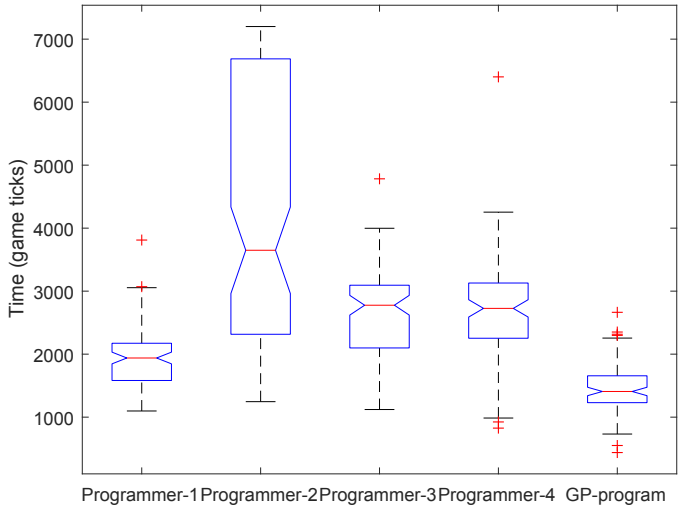


Fig. 6: Box plot comparing all implementations with respect to medians (50th percentile), 25th and 75th percentiles, min, max, outliers (+) as well as comparison intervals (notch).

We consider that individually smarter sheep with better sensors would be much more difficult to prey. Sheep avoid being caught only by the logic of *the selfish herd* [14] according to the collective flocking rules described above in which all sheep share the same behavior algorithm. The present simulation assumes that sheep lack accurate sensors, knowledge about the predator capabilities, and tactical shrewdness. Since prey animals have evolved under the dominating selection pressure of predation, these seem to be rash and perhaps unwarranted assumptions. We should give more credit to sheep intelligence in future work, perhaps by co-evolving sheep and wolf intelligence. This could well result in a much smarter wolf with a much higher capacity for hunting smart as well as stupid sheep.

Furthermore, based on our observations from running the simulation many times, we observe that the herd is quickly exterminated once it is driven into a corner of the field. One of the advantages of the GP-wolf is that it seems to herd escaping sheep back into the corner by running back and forth imitating the behavior of a herding dog.

The reason for the rapid demise of a cornered herd is that Equation 1 to Equation 5 include no information on the positions of the corners, which makes it possible for the sheep to accumulate in significantly large groups within the wolf striking distance of 10 pixels, so that the wolf can kill many sheep in one move once the herd is cornered. It would be possible to modify the flocking model to make the herd avoid corners and even by include a scatter mechanism if the sheep are packed too tightly.

Finally, we want to clarify that we do not claim that wolves in the wild behave as in our simulation that is indiscriminately killing all prey animals within reach. We do not want to add to unnecessary vilifying of predators.

VI. Conclusions

Based on Deutsch's conjecture that all creativity is based on Darwinian evolution we have demonstrated emergent computational creativity by using genetic programming for developing intelligent behaviors in a simple game simulating a wolf hunting flocking sheep. We found that the wolf hunting behavior evolved by genetic programming is not only more efficient but also able to generalize better than the corresponding behavior programmed by proficient humans, and that it is unlikely that a human would develop the same code as the genetically programmed one. Perhaps, the most interesting finding in this work is that the results indicate that a computer, running a GP algorithm, can generate CGF behaviors that appears to be too complex to model or implement using the manual programming approach.

Even though the results in this work were acquired using a toy-problem they provide valuable insight into the opportunities as well as the challenges of our computational creativity method and, ultimately, its real-world application. In future works we intend to apply the approach to generate CGFs in the context of the Virtual Battlespace 3 (VBS3) warfare simulator (developed by Bohemia Interactive Simulations[2]). VBS3 is a simulator used for tactical training within the armed forces of Sweden, Norway and many other nations worldwide. Major challenges, identified in this work, related to the planned VBS3 experiment include:

1) Real-time simulation: VBS3 is a simulator where actions, performed by CGFs or human players, are executed in real-time. Hence, unlike the toy-problem presented here, fitness evaluation using VBS3 will also execute in real-time, ultimately, limiting number of evaluations that can be performed in the search for a creative and intelligent CGF behavior. The most commonly used approach to resolve this issue is to distribute fitness evaluation using high-performance computing and clusters. Another interesting approach that we are investigating in parallel [10], [11] to this work is to use a hybrid learning approach where the initial population of CGFs are initiated, not by random, but with CGFs that first have been trained using observational learning (implemented using supervised machine learning techniques) [15].
2) Fitness function: The fitness function is critical to the successful application of GP. The purpose of the fitness function is to define the desired behavior of the CGF. However, defining the desired behavior in a single function is difficult and requires careful consideration. To the best of our knowledge there is no silver bullet that can be applied to this issue. Hence, developing a fitness function to generate intelligent CGF behaviors in VBS3 will most likely be a painstaking, iterative trial-and-error, process. The goal would be to create a fitness function that, similar to the fitness function presented in this work, is modular and easily modified by the end-user using weights.

[2]https://bisimulations.com/

3) Complexity: The complexity with respect to the synthetic environment, physics, level of details, model fidelity, sensors and available actions in a warfare simulator such as VBS3 will significantly impact the search space of the GP algorithm. In a tactical warfare simulator the CGF must be able to realistically interact with the synthetic environment, as well as with both human role-players and other CGFs. The input variables (leaf nodes in the GP tree) must provide in addition to the position of the CGF itself, the positioning of (relative to the CGF's perception) visible teammates, opponents, environmental objects, etc. Also, the list of function nodes have to be extended to ensure that the inputs can be transformed into a meaningful behavior through the GP's mutation and crossover operators.

Needless to say the above challenges are indeed intimidating. However, the potential payoff of successfully applying the computational creativity approach to model better, more creative and intelligent CGFs for military training applications is significant. It would improve training efficiency and lower costs. It would reduce the number of human role-players required to support and stimulate the trainees. Ultimately, it is our belief that it could also better prepare military commanders, staffs and soldiers for surprising, unexpected and high-impact events.

ACKNOWLEDGMENT

This paper is the result of a collaborative effort by the Swedish Defence Research Agency (FOI) and the Norwegian Defence Research Establishment (FFI).

The work was supported by the FOI research project "Synthetic Actors", which is funded by the R&D programme of the Swedish Armed Forces, and by the FFI research project "Cost efficient training for the Norwegian Armed Forces".

This paper would not have been possible without knowledge accumulated in the project "Computational Creativity" commissioned by FMV, the Swedish Defence Materiel Administration.

REFERENCES

[1] M. Tambe, W. L. Johnson, R. M. Jones, F. V. Koss, J. E. Laird, P. S. Rosenbloom, and K. Schwamb, "Intelligent agents for interactive simulation environments," *AI Magazine*, vol. 16, no. 1, pp. 15–39, 1995.

[2] N. Abdellaoui, A. Taylor, and G. Parkinson, "Comparative analysis of computer generated forces' artificial intelligence," *RTO-MP-MSG-069 - Current uses of M&S Covering Support to Operations, Human Behaviour Representation, Irregular Warfare, Defence against Terrorism and Coalition Tactical Force Integration*, 2009.

[3] A. Toubman, G. Poppinga, J. J. Roessingh, M. Hou, L. Luotsinen, R. A. Løvlid, C. Meyer, R. Rijken, and M. Turčaník, "Modeling CGF Behavior with Machine Learning Techniques: Requirements and Future Directions," in *Proceedings of the 2015 Interservice/Industry Training, Simulation, and Education Conference (I/ITSEC)*, Orlando, Florida, 2015.

[4] D. Deutsch, *The beginning of infinity: explanations that transform the world*. Penguin Books, 2012.

[5] F. B. Churchill, "William johannsen and the genotype concept," *Journal of the History of Biology*, vol. 7, no. 1, pp. 5–30, 1974.

[6] R. Dawkins, *The selfish gene*. New York: Oxford university press, 1976.

[7] J. R. Koza, *Genetic programming II: automatic discovery of reusable programs*. Cambridge, MA, USA: MIT Press, 1994.

[8] ——, *Genetic programming: on the programming of computers by means of natural selection*. Cambridge, MA, USA: MIT Press, 1992.

[9] J. McCormack and M. d'Inverno, *Computers and creativity*. Springer, 2012.

[10] L. J. Luotsinen and R. A. Løvlid, "Data-driven behavior modeling for computer generated forces," in *NATO modelling and simulation group symp. M&S support to operational tasks including war gaming, logistics, cyber defence (MSG-133)*. NATO, 2015, pp. 1–13.

[11] F. Kamrani, L. Luotsinen, and R. A. Løvlid, "Learning objective agent behavior using a data-driven modeling approach," *IEEE International Conference on Systems, Man, and Cybernetics*, 2016.

[12] C. W. Reynolds, "Flocks, herds and schools: a distributed behavioral model," *SIGGRAPH Comput. Graph.*, vol. 21, no. 4, pp. 25–34, Aug. 1987.

[13] M. Barksten and D. Rydberg, "Extending Reynolds' flocking model to a simulation of sheep in the presence of a predator (bachelor's thesis)," 2013.

[14] W. Hamilton, "Geometry for the selfish herd," *J. of Theo. Biology*, vol. 31, no. 2, pp. 295–311, 1971.

[15] G. Stein and A. J. Gonzalez, "Building high-performing human-like tactical agents through observation and experience," in *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, 2011, pp. 792–804.