Automating Behaviour Tree Generation for Simulating Troop Movements (Poster)

Gabriel Berthling-Hansen*, Eivind Morch*, Rikke Amilde Løvlid†, and Odd Erik Gundersen*‡
*Norwegian University of Science and Technology (NTNU), Trondheim, Norway
†Norwegian Defence Research Establishment (FFI), Kjeller, Norway
‡Corresponding author: odderik@ntnu.no

***Abstract*—Computer generated forces are simulated units that are used in simulation based training and decision support in the military. These simulations are used to help trainees build a mental model of how different scenarios could play out, and thus give them a better situation awareness when conducting operations in real life. The behaviour of these simulated units should be as realistic as possible, so that the lessons learned while simulating are applicable in real situations. However, it is time consuming and difficult to build behaviour models manually. Instead, we explore the possibility of applying machine learning to generate behaviour models from a set of examples. In this paper we present the results of our preliminary experiments on using machine learning for behaviour modelling. We implement a follow behaviour by using behaviour trees that are evolved using genetic algorithms. The fitness of the evolved behaviour trees have been evaluated by comparing them with a manually generated behaviour tree that implements the behaviour properly. The genetic algorithm converges to a tree that is very similar to the manually generated behaviour tree, suggesting that the method works. Further work is necessary to test whether this approach will work on more complex behaviours.**

***Index Terms*—Behaviour tree, simulation, genetic algorithms**

I. Introduction

Computer generated forces (CGFs) are autonomous or semi-autonomous entities that represent military units, such as tanks, soldiers and combat aircrafts, in simulation software for military operations. CGFs are similar to non-player characters in computer games and are used in military simulation-based training and decision support applications. CGFs enable simulating large military operations as one operator is able to control several military units. The behaviour of the CGFs, e.g. how they move, where they look, when they shoot etc., should represent the behaviour of corresponding human soldiers or manned systems as accurately as possible. Ideally, a soldier training with a virtual simulator should not notice whether his teammates or opponents are human controlled entities or CGFs. Realistic CGF behaviour also makes it possible to simulate various plans or courses of actions to improve the situation awareness and get a good understanding of how a situation could play out [1]. Simulations can help build and train the mental model of the trainee by practising situation comprehension and projection, situation awareness level two and three in Endsley's model of situation awareness [2]. In aviation, around 20% of the errors are related to problems with the mental model according to Endsley and Garland [3]. Given that errors made by soldiers in a stress situation are similar to those made in aviation, improving their mental model is of high importance. This requires the CGFs to behave in a natural way, as their behaviour affects the situation comprehension and projection of the trainee.

There are several ways to represent the behaviour of CGFs. The most common way is to use state machines that describe different states that the CGFs can be in and the actions they can perform in every state. Lately, however, behaviour trees have grown very popular [4]. In any case, the behaviour models are typically made manually. This means that military experts have to tell programmers how they want CGFs to behave. This is a difficult and time consuming process [5], and we want to explore how to use machine learning to generate behaviour models from examples of desired behaviour.

One potential problem when building a behaviour model with machine learning is that the model often becomes opaque, meaning it becomes hard to interpret what the model has learned. In our work we decided to focus on trying to learn behaviour trees, i.e. the same type of model that can be used to model the behaviour manually. By using behaviour trees, the learned model for a CGF is explicit, which enables and simplifies explaining the behaviour. Core et al. [6] present a similar system with a separate module for explaining the CGF's behaviours. By contrast, they represent the behaviours as rules and not behaviour trees.

Using machine learning to generate behaviour models for CGFs has been discussed in the NATO Research Task Group IST-121 RTG-060 "Machine Learning Techniques for Autonomous Computer Generated Entities" [7]. The paper refers to different case studies performed by the participating nations. Worth mentioning is the work done by Totalförsvarets forskningsinstitut (FOI), who used machine learning to create autonomous agents that learn a tactical movement called bounding overwatch for dismounted infantry [8]. They divided the behaviour into different decision-models, like whether to move or not, where to aim and whether to stand or kneel. These models where learned separately and combined manually. This research was done with Virtual Battle Space 3 (VBS3), a game based military simulation system [9].

In this paper we present our preliminary work on using machine learning to generate a behaviour model for CGFs. The work has been done with a real, military simulation system called VR-forces from MÄK [10]. We have used a genetic algorithm (GA) to generate a model for a soldier who follows another soldier. The next section includes necessary background information on genetic algorithms and behaviour trees. Section III describes the simulation system architecture and how data generation and the actual training are performed. An experiment and results are presented in section IV, followed by a discussion of the results and related work in section V. Finally, conclusion and suggestions for future work are included in VI.

II. Background

A. Genetic Algorithms

GAs are stochastic search algorithms inspired by evolution. A GA generates and evolves a population of chromosomes, where each chromosome is a candidate solution for solving a problem. Chromosomes are assigned a value representing how well they solve the problem, called fitness. In each epoch, the GA produces a new generation of chromosomes through crossover and selection. Crossover is the creation of a new chromosome by combining the traits of two existing chromosomes to produce a hybrid solution. Selection is the process of deciding which chromosomes should be used for crossovers, and which should be potentially included as they are in the new generation, which is usually done by comparing fitness values. The chromosomes are also randomly mutated, making direct changes to existing solutions. See [11] for more information on GAs.

B. Behaviour Trees

Behaviour trees are trees of hierarchical nodes that control decision making and task execution, and have been popularly used for modelling the behaviour of computer-controlled units in video games [4]. They provide a scalable and modular solution for representing complex behaviour without the exponential scalability of Finite State Machines (FSM) and reusability-problem of Hierarchical Finite State Machines (HSFM) [12]. Behaviour trees are also human-readable, giving the opportunity for visual analysis of the represented behaviour.

Behaviour tree nodes can return one of three statuses: *running*, *success* or *failure*. Running means that the node is currently active, has not completed its tasks and needs more time to finish. Success is returned when a node is finished executing and its task was successfully completed, and failure is returned when the task finished unsuccessfully.

Behaviour trees are traversed from the root node and down. If all visited nodes are finished, the tree will be traversed from the root and down again on the next timestep. However, if one of the nodes return running, the tree will keep running that node every timestep until it returns either failure or success. Once the node is done, the tree will continue traversing from the position of the node.

1) Composite Nodes: A composite node is used to group nodes into a higher level task [12]. The type of the composite node dictates in which order it will execute children nodes, when to stop, and what status to return. The system described in this article uses two types of composite nodes, *sequence* and *selector*. A sequence node (displayed as $\rightarrow$) executes its children from left to right until one of the children returns failure or all return success. If a child returns failure, then the sequence will stop and return failure. If all its children return success, it will return success. A selector node (displayed as **?**) executes its children from left to right until one of the children returns success or all children return failure. If a child returns success, the selector will stop and return success. If all its children return failure, the selector will return failure.

2) Leaf Nodes: A leaf node has no children, and is either an action node or a condition node. An action node is used to perform a specific low-level action, e.g. move to a certain location. A condition node returns success or failure based on some condition, e.g. whether an object is within a specific distance or not.

3) Blackboard: A blackboard contains data that is accessible for all the nodes of the behaviour tree. Nodes may also alter data inside the blackboard. A blackboard is an important feature of a behaviour tree as it enables nodes to share and alter the same state representation, avoiding an exponential state complexity such as in FSMs.

III. Generating Behaviour Trees for CGF

A. System Architecture

For our experiments we used a real, military simulation system called VR-Forces from MÄK. The virtual terrain, physical simulation of entities etc. are simulated in this system. We made a separate system that can record data from VR-Forces, generate the behaviour models using machine learning and send commands to the entities in VR-Forces. Figure 1 shows an example of a virtual terrain in VR-Forces. Our system communicates with VR-Forces using high level architecture (HLA), a standard for distributed simulation that is commonly used in military simulation systems [13]. Other CGF systems that support HLA could be used in place of VR-Forces.

Systems that communicate over HLA must agree on data and data formats to exchange. This is formalised in a federation object model (FOM), and we have used the Real-time Platform-level Reference (RPR) FOM, which is a standard FOM that many military simulation systems support [14], [15]. However, this FOM does not include commands or the perceived truth of the CGF entities. For this we use an extra module for low level battle management language (LL-BML), which is made as an extension to the RPR FOM [16], [17]. Bruvoll et al. describe using a multiagent system to control a CGF system in a similar manner [1], [18].

Our system generates and evaluates behaviour models for different types of units in different scenarios. Different units, scenarios and objectives require different behaviour tree nodes

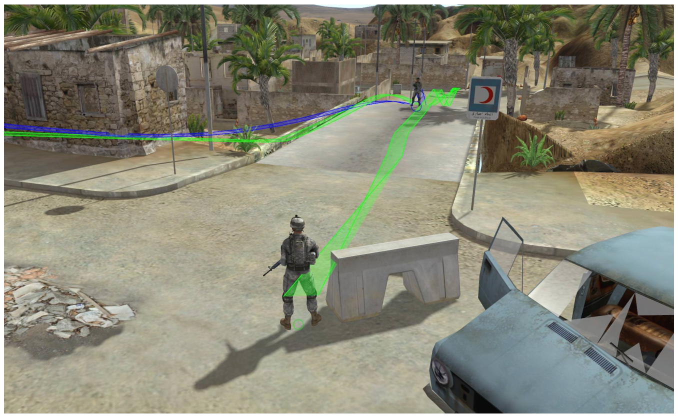


Fig. 1: VR-Forces simulation environment

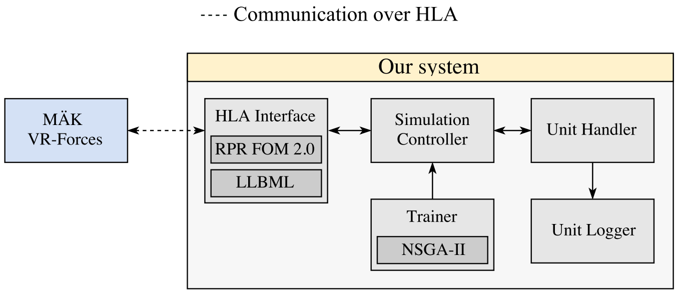


Fig. 2: Architecture Overview

with specialised tasks, different data collection and data processing, different performance evaluations (fitness), and different training algorithm tuning.

Figure 2 shows a high level overview of the system architecture. The *Simulation Controller* provides an abstraction level for the rest of the system to interact with VR-Forces, which it communicates with through the *HLA Interface*. These interactions include sending instructions to play, pause and load a scenario, as well as forwarding events of newly discovered units from VR-Forces to the *Unit Handler*. The Unit Handler handles the registration of simulation entities as local unit objects that can be used for logging data or giving commands. The Simulation Controller also instructs the Unit Handler to update unit data when the simulation time advances (tick). The *Unit Logger* writes data from the units registered by the Unit Handler to a database. The *Trainer* handles training of behaviour models. This includes deciding which scenario to simulate, initiating simulations, and creating, evaluating and modifying the behaviour trees. The main system has two modes—example recording and training.

1) Example Recording: The recording mode is used to record the behaviour of a unit that is controlled by an external source (human or script), in order to generate training data. This could be a person controlling a unit by joystick or mouse and keyboard through VR-Engage [19] while our system records data relevant to the performed task from the simulation. When recording, VR-Forces is initiated externally, and our system is only listening to and logging data updates,

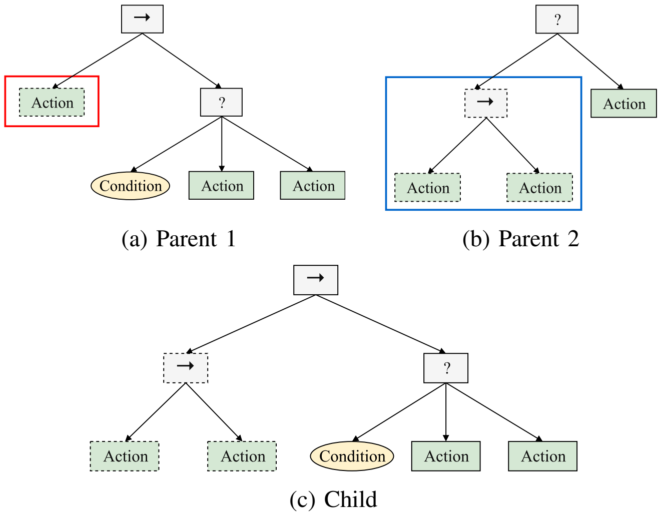


Fig. 3: Crossover

such as position, velocity and current engagement status. This data can later be used as examples of desired behaviour in the respective scenario.

2) Training: During training, the system uses a GA to generate, evaluate and improve behaviour trees. This is handled by the *Trainer*, shown in Figure 2, which takes a GA implementation as an argument upon initiation. NSGA-II [20], a multi-objective GA, is used for the experiments described in this paper. The *Trainer* also requires a list of scenarios and recorded example data, an object responsible for evaluating the fitness of behaviour trees, and an object responsible for collecting and holding the data to be used for evaluation.

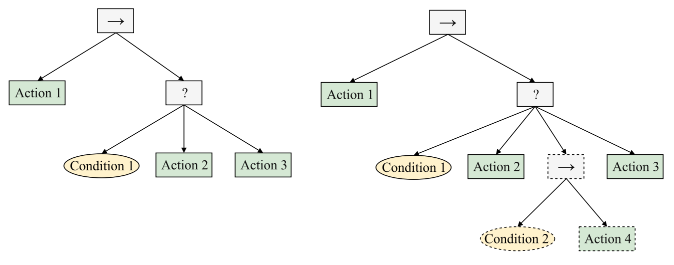
B. Evolving Behaviour Trees

The evolution of behaviour trees is done using NSGA-II, a multi-objective GA which sorts and selects chromosomes by non-domination [20]. A chromosome is said to dominate another chromosome if it has a better fitness value for one or more objectives and equal fitness value for the rest.

1) Crossover: The crossover operator chooses two behaviour trees as parents through tournament selection, and then chooses a random subtree in each of the parents, as illustrated in Figures 3a and 3b. A clone of parent 1 is then created and the subtree of parent 2 is inserted at the position of the subtree of parent 1. Figure 3c displays the final tree after crossover. This is the same crossover operator as is used in [4].

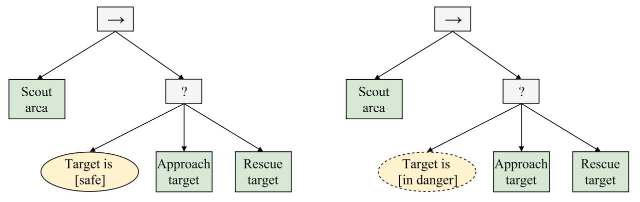
2) Mutations: The system uses six different mutations with varying level of impact on the behaviour trees. The probability of choosing one mutation over the others is decided by weights that are tuned per experiment. It is also possible to set weight factors for each mutation, altering the mutation weight depending on how many epochs the algorithm has been running. All mutations were designed by the authors of this paper for this specific system.

a) Add Random Subtree: This mutation generates a random subtree with a specified minimum and maximum number of nodes, and inserts it at a random position in the behaviour



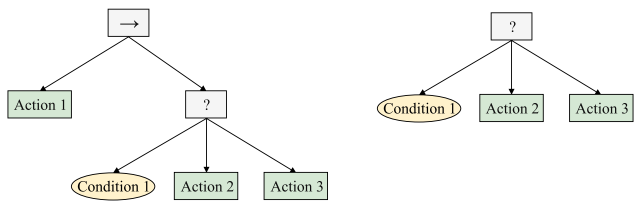
(a) Original (b) Result

Fig. 4: Add Random Subtree



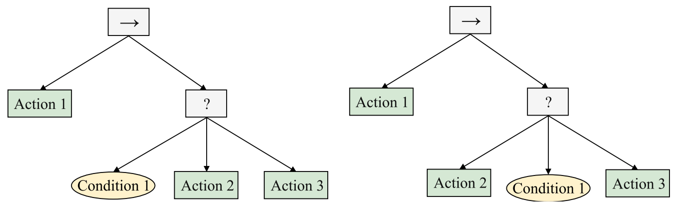
(a) Original (b) Result

Fig. 5: Randomise Variables of Random Node



(a) Original (b) Result

Fig. 6: Replace Tree With Subtree



(a) Original (b) Result

Fig. 7: Switch Positions of Random Sibling Nodes

tree. Figure 4 shows the insertion of a tree with three nodes, marked with a dotted line.

b) Randomise Variables of Random Node: Both action nodes and condition nodes can have variables that affect their functionality. This reduces the total number of nodes a developer has to make and also allows the system to fine-tune the behaviour of the behaviour tree. E.g., for a condition node that checks if the distance between two units is lower than a certain value, the value can be altered during training to check for different distances. This mutation randomises one or multiple variables in an action or condition node. In Figure 5 the value of the *"Target is ..."* condition node is changed from *safe* to *in danger*.

c) Remove Random Subtree: This mutation removes a random subtree from a behaviour tree.

d) Replace Random Node With Node of Same Type: This mutation replaces a random node with another random node of the same type. This means that a composite node can be replaced with another composite node (e.g. sequence to selector) or that a leaf node is replaced with another leaf node. Condition and action nodes are not treated differently, and may be replaced by any other leaf node.

e) Replace Tree With Subtree: This mutation replaces the entire tree with a random subtree of that tree. In Figure 6 the entire tree is replaced by the selector subtree.

f) Switch Positions of Random Sibling Nodes: This mutation switches the position of two random sibling nodes, including both leaf and composite nodes. In Figure 7 the *Condition 1* and *Action 2* nodes have switched places.

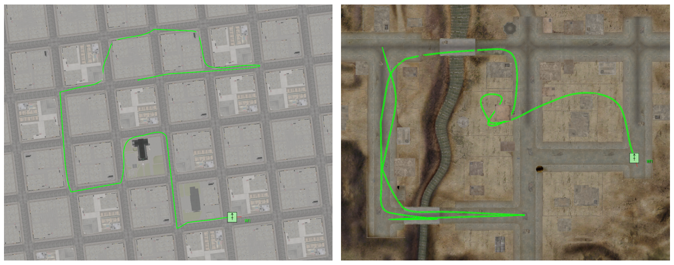
IV. Experiments and Results

A. Experiments

The experiment revolves around a wanderer and a follower. The objective for the experiment is to have the follower agent follow the wanderer. The wanderer is pre-programmed to follow a specific plan, which involves going to different locations and waiting a given amount of time at certain positions. The example file used to for training was recorded from a manually created behaviour tree, shown in Figure 11. This was an easy way to generate training data, and it also made it possible to compare the learned model with the "true" model.

A behaviour tree for a follower unit can have five different types of leaf nodes: *Move to target*, which makes the follower move toward its target for one tick. *Turn to target*, which makes the follower turn towards its target. *Wait*, which makes the follower stand still for a single tick. *Is within*, a variable condition node which checks whether the followers target is within a specified euclidean distance. *Is approaching*, a variable condition node which checks whether the angle between the movement vector of the target and the vector between the follower and the target is smaller than a specified threshold—effectively checking whether the followers target is moving towards the follower.

For this experiment, the training was done on two separate scenarios, using different terrains and wanderer paths. 2D overviews of the terrain and wanderer paths for both scenarios are shown in Figure 8. The first scenario simulates approximately 18 minutes of real-time over 1100 ticks, and the second approximately 12 minutes of real-time over 700 ticks. The training ran for 30 epochs with a population of 10 behaviour trees. The objective is to imitate the recorded behaviour. The fitness function therefore compares the behaviour produced by the behaviour tree with the recorded behaviour. The behaviour trees were evaluated by comparing the euclidean distance in each tick between follower and target position in the training data with the follower-target distance during behaviour tree simulation. All trees were tested on both scenarios. The equation for calculating the fitness value of a behaviour tree



(a) Scenario 1 (b) Scenario 2

Fig. 8: Scenario terrain and path overviews

for a single scenario is shown in Equation 1, where n is the number of ticks that were simulated.

$$Fitness = \frac{1}{n} \times \sum_{t=0}^{n} \Big(dist(example_t) - dist(btree_t) \Big)^2 \quad (1)$$

For each of the recorded ticks we find the follower-target distance from the training data and the simulated behaviour tree. The difference of these two distances is then squared so that a large difference over a few ticks is worse than a small difference over a large number of ticks. The squared differences in euclidean distance are summed and normalised over the number of ticks (n) to make the different scenario fitness values more comparable. The fitness values should be minimised. We chose this fitness function, as it is a simple formula that captures the similarity of the example and evaluated behaviour.

During training in this experiment, NSGA-II was set to simultaneously minimise three fitness values: the fitness value from running scenario 1, the fitness value from running scenario 2, and the number of nodes in the behaviour tree. By adding the tree size as a minimisation objective, we prevented the algorithm from creating bloated behaviour trees with unnecessary subtrees that have no significant effect on the behaviour.

All mutations were initially weighted the same, however with different weight factors, as described in Section III-B2. The mutations that change the behaviour trees drastically—*add random subtree* and *Replace tree with subtree*—were given a factor of less than 1, while *randomise variable of random variable node* was given a factor higher than 1. This way, the algorithm prioritises local search over larger changes at later epochs. The creation, crossover and mutation functions for behaviour trees were also restricted from producing behaviour trees with less than three and more than 12 nodes.

B. Results

Figure 9 shows the development of the fitness for the first scenario over 30 epochs. Figure 10a shows the fitness development over time for scenario 2, and Figure 10b shows a zoomed in view of the best-fitness development. The results show that the generated behaviour trees improve over time. For both scenarios, the trend is that the best and average score

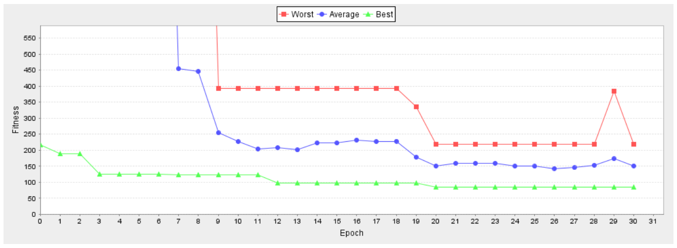
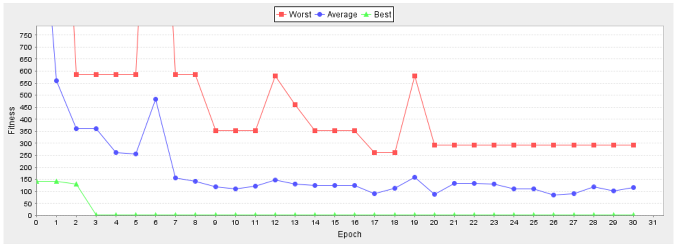
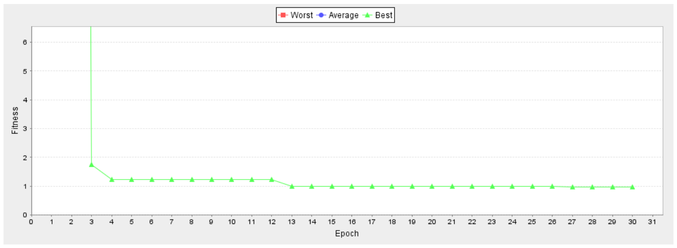


Fig. 9: Scenario 1 fitness development



(a) Complete



(b) Zoomed

Fig. 10: Scenario 2 fitness development

is continually decreasing as the algorithm is running. Short-term increases in average fitness values are due to the multi-objective selection of the NSGA-II algorithm.

We have chosen two of the non-dominated behaviour trees from the population at epoch 30, shown in Figure 12 with fitness values included in the sub figure captions. The behaviour tree in Figure 12a has the smallest possible size following the size restrictions, and has the lowest fitness on scenario 2 of all the trees of the same size. The larger one, shown in Figure 12b, performs better at scenario 1 and scenario 2, but has more than twice the number of nodes.

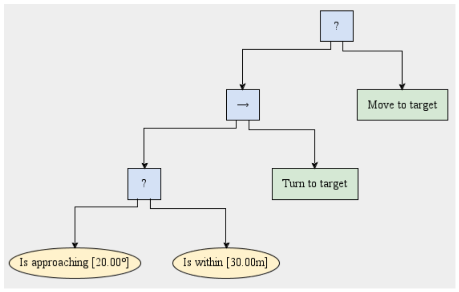
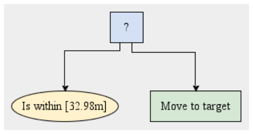
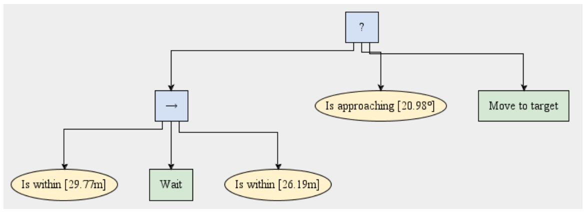


Fig. 11: Manually made behaviour tree



(a) Fitness: [size=3, scenario1=219, scenario2=114]



(b) Fitness: [size=7, scenario1=84, scenario2=5]

Fig. 12: Two of the resulting behaviour trees from epoch 30

V. Evaluation and Discussion

We used a manually created behaviour tree, shown in Figure 11, to record the example file used in the training phase. While running the experiment, we observed different outcomes of the simulations with identical inputs. E.g., running the multiple simulations with the same behaviour tree controlling the follower resulted in slightly different pathing, and therefore different fitness values. When simulating the same behaviour tree that was used for recording the example data, the fitness on scenario 1 varied between 0 and 40, and on scenario 2 the fitness varied between 0 and 9.

It appears that the simulation engine has internal inconsistencies on when received unit tasks are executed. We suspect this is caused by the internal path planning of the simulation engine taking different lengths of time to complete due to available processing power. It seems the simulation is not paused while the pathing is calculated, and so the units start to move at different ticks.

The consequence is that we have to account for stochastic evaluation of chromosomes, where a chromosome can be evaluated better or worse based on luck. A way of managing stochastic problems with GAs is to simulate each chromosome a large number of times, combining the results [21]. However, as it takes approximately 18 seconds to simulate each behaviour tree, running a large number of simulations per chromosome was not an option. The inconsistencies in fitness evaluation can also affect the selection process of NSGA-II negatively.

When comparing the two selected resulting behaviour trees with the manually made behaviour tree used for recording the example, we can see that there are significant similarities. For the tree in Figure 12a, it has managed to represent approximately what we consider the most important part of the example behaviour—moving only when more than 30 meters away from the target. In both example scenarios, checking for distance is more important than checking whether the target is approaching. This is because the target usually moves away from the follower, and that the target is standing still for a significant portion of the scenarios.

The bigger resulting tree, shown in Figure 12b, includes most of the behaviour of the manually made example tree. Before moving to the target, both distance and movement angle is checked with approximately the same distance and angle used in the example tree. However, due to the sequence of distance checks and wait node, the follower might move closer while it is between 29.77 and 26.19 meters away from the target. Then again, as the target is moving, the wait node between the distance checks will often cause the target to be further than 29.77 meters away for the next tick.

Theoretically, the system should be able to find behaviour trees that result in 0 fitness on both scenarios. However, we suspect that the stochastic simulation outcomes significantly limits the performance of the algorithm by causing it to keep trees that were lucky during evaluation over trees with statistically better performance that were unlucky during evaluation. Combined with the long time it takes to simulate a behaviour tree, finding optimal models will take a very long, even with simple experiments.

The bottleneck of the system is running simulations to evaluate the behaviour trees. Simulating a single behaviour tree on the two scenarios used for the previously described experiment takes approximately 18 seconds. This limits the population size we can use for NSGA-II, as evaluating a large number of chromosomes per epoch will be inefficient use of time, especially when multiple scenarios are used for evaluation.

Colledanchise et al. [4] and Lim et al. [12] also use GAs to generate behaviour trees, with the same crossover operator as we have used in our system. For mutation, however, Colledanchise et al. used a single mutation which replaces a single node in a behaviour tree with another node of the same type, while we use six different mutations that alter the behaviour tree in different ways. Another important difference is that Colledanchise et al. use reinforcement learning to generate behaviour trees designed to play Mario, while we generate behaviour trees that imitate example behaviour in complex simulation environments.

Lim et al. [12] use two mutations with similarities to two of our mutations—adding a random behaviour tree as a subtree, and changing the an inner variable of an existing node. They also had issues with long simulation times, which they handled by distributing their simulation over 20 computers, drastically speeding up each experiment. Lim et al. trains the behaviour using reinforcement learning while we use supervised learning.

The followers behaviour model is currently fed ground truth information about the target from the simulation system. This means that the follower always knows the current position of the target, even if the follower unit is unable to observe the targets position and velocity. For a more realistic experiment, the follower behaviour model should only have access to perceived truth, which is supported in VR-Forces through e.g. line of sight and radio communication. This would probably result in more human-like behaviour.

VI. Conclusion and Future Work

In this paper we used genetic algorithms to generate behaviour trees that control the behaviour of CGFs in a real military simulation system. The objective was to imitate a recorded behaviour. As mentioned in the previous section, other researchers have used GAs to generate behaviour trees, but we have not seen other work that has used GAs for learning from observation. Also, the other research has been done with simpler simulation systems.

We consider the experiment to be a success. The resulting behaviour trees reproduce the most essential parts of the behaviour used to record the example data, which shows that the system is able to replicate simple behaviour by generating and evolving behaviour trees.

We were surprised by the fact that the simulation system is not deterministic, which resulted in significant variation in the fitness of a behaviour tree when executed on the same scenario multiple times. Assuming that we are able to solve the issues with stochastic simulation outcomes, the system's ability to replicate more complex behaviour seems promising. Hence, solving this issue should has the highest priority going forward.

Increasing the complexity of the objective and scenarios is also very relevant. E.g. using a number of agents to perform a military action called bounding overwatch where multiple agents must work together to advance forward. See [8] for more information on bounding overwatch. The agents could use radio messages to communicate when the next agent should advance. Extending the experiment described in this paper with variable movement speeds and using perceived truth target information are other potential ways of increasing experiment complexity.

In addition to implementing a larger variety of action leaf nodes, we wish to introduce other types of composite nodes in future experiments, e.g., random and parallel composite nodes. We also wish to include the use of decorators, that alter the resulting status of a single node.

The example used in this experiment was recorded with a manually created behaviour tree used to control the follower unit. This was done to have a representation of optimal behaviour to compare the results with, making visual analysis easier. However, for future experiments, the example behaviour should be recorded with the unit controlled by a human, e.g. using VR-Engage to control the example unit with mouse and keyboard. It should be interesting to compare how the person controlling the unit would represent his/her own behaviour to how the computer ends up representing it.

Finally, we aim at investigating the possibilities of distributing the simulation to reduce the required run-time of simulating and evaluating behaviour trees. Lim [12] had a similar problem, where running the entire experiment would take approximately 41 days. They were able to distribute the computation to 20 computers, reducing the number of days to approximately 2 days per experiment. This is also an option for our simulation system and would allow for a larger population or a higher number of epochs in the experiment.

References

[1] S. Bruvoll, J. E. Hannay, G. K. Svendsen, M. L. Asprusten, K. M. Fauske, V. Kvernelv, R. A. Løvlid, and J. I. Hyndøy, "Simulation-supported wargaming for analysis of plans," in *NATO Modelling and Simulation Group Symphosium. M&S Support to Operational Tasks Including War Gaming, Logistics, Cyber Defence (MSG-133)*, 2015.

[2] M. R. Endsley, "Toward a theory of situation awareness in dynamic systems," *Human factors*, vol. 37, no. 1, pp. 32–64, 1995.

[3] M. R. Endsley and D. J. Garland, "Pilot situation awareness training in general aviation," in *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, vol. 44, no. 11. SAGE Publications Sage CA: Los Angeles, CA, 2000, pp. 357–360.

[4] M. Colledanchise, R. Parasuraman, and P. Ögren, "Learning of behavior trees for autonomous agents," *arXiv preprint arXiv:1504.05811*, 2015.

[5] 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*, 2015, pp. 2637–2647.

[6] M. G. Core, H. C. Lane, M. Van Lent, D. Gomboc, S. Solomon, and M. Rosenberg, "Building explainable artificial intelligence systems," in *AAAI*, 2006, pp. 1766–1773.

[7] J. J. Roessingh, A. Toubman, J. van Oijen, G. Poppinga, R. A. Løvlid, M. Hou, and L. Luotsinen, "Machine learning techniqus for autonomous agents in military simulations - multum in parvo," in *Proceedings of the 2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, 2017, pp. 3445–3450.

[8] F. Kamrani, L. J. Luotsinen, and R. A. Løvlid, "Learning objective agent behavior using a data-driven modeling approach," in *Systems, Man, and Cybernetics (SMC), 2016 IEEE International Conference on*. IEEE, 2016, pp. 002 175–002 181.

[9] Bohemia Interactive. (2016) VBS. [Online]. Available: https://bisimulations.com/virtual-battlespace-3

[10] MÄK. (2018) VR-Forces. [Online]. Available: https://www.mak.com/products/simulate/vr-forces

[11] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach*, 3rd ed. Pearson Education, 2014, pp. 129–132.

[12] C.-U. Lim, R. Baumgarten, and S. Colton, "Evolving behaviour trees for the commercial game defcon," in *European Conference on the Applications of Evolutionary Computation*. Springer, 2010, pp. 100–110.

[13] NATO NSA, *STANAG 4603 - Modelling and Simulation Architecture Standards for Technical Interoperability: High Level Architecture (HLA)*, 2nd ed., 2015.

[14] Simulation Interoperability Standards Organization (SISO), *Standard for Guidance, Rationale, and Interoperability Modalities (GRIM) for the Real-time Platform Reference Federation Object Model (RPR FOM), Version 2.0*, http://www.sisostds.org/DigitalLibrary.aspx?Command=Core_Download\&EntryId=30822, 2015, SISO-STD-001-2015.

[15] ——, *Standard for Real-time Platform Reference Federation Object Model (RPR FOM), Version 2.0*, http://www.sisostds.org/DigitalLibrary.aspx?Command=Core_Download\&EntryId=30823, 2015, SISO-STD-001.1-2015.

[16] A. Alstad, O. Mevassvik, M. Nielsen, R. Løvlid, H. Henderson, R. Jansen, and N. de Reus, "Low-level battle management language," in *Proceedings of the 2013 Spring Simulation Interoperability Workshop*, no. 13S-SIW-032, 2013.

[17] J. Ruiz, D. Dsert, A. Hubervic, P. Guillou, R. Jansen, N. de Reus, H. Henderson, K. Fauske, and L. Olsson, "BML and MSDL for multi-level simulations," in *Proceedings of the 2013 Fall Simulation Interoperability Workshop*, no. 13F-SIW-002, 2013.

[18] A. Alstad, R. A. Løvlid, S. Bruvoll, M. N. Nielsen, and O. M. Mevassvik, "Autonomous simulation of a battalion operation - seamless integration of command and control and simulation for planning and training," Forsvarets forskningsinstitutt, FFI-rapport 2013/01547, 2013.

[19] VT MAK. (2017) VR-Engage. [Online]. Available: https://www.mak.com/products/simulate/vr-engage

[20] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE transactions on evolutionary computation*, vol. 6, no. 2, pp. 182–197, 2002.

[21] R. Al-Aomar, "Incorporating robustness into genetic algorithm search of stochastic simulation outputs," *Simulation Modelling Practice and Theory*, vol. 14, no. 3, pp. 201–223, 2006.