Proceedings, The Eleventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE-15)

Capturing the Essence: Towards the Automated Generation of Transparent Behavior Models

Patrick Schwab and **Helmut Hlavacs**
Research Group Entertainment Computing, Faculty of Computer Science
University of Vienna, Vienna, Austria
a0927193@unet.univie.ac.at, helmut.hlavacs@univie.ac.at

Abstract

Hand-coded finite-state machines and behavior trees are the go-to techniques for artificial intelligence (AI) developers that want full control over their character's bearing. However, manually crafting behaviors for computer-controlled agents is a tedious and parameter-dependent task. From a high-level view, the process of designing agent AI by hand usually starts with the determination of a suitable set of action sequences. Once the AI developer has identified these sequences he merges them into a complete behavior by specifying appropriate transitions between them. Automated techniques, such as learning, tree search and planning, are on the other end of the AI toolset's spectrum. They do not require the manual definition of action sequences and adapt to parameter changes automatically. Yet AI developers are reluctant to incorporate them in games because of their performance footprint and lack of immediate designer control. We propose a method that, given the symbolic definition of a problem domain, can automatically extract a transparent behavior model from Goal-Oriented Action Planning (GOAP). The method first observes the behavior exhibited by GOAP in a Monte-Carlo simulation and then evolves a suitable behavior tree using a genetic algorithm. The generated behavior trees are comprehensible, refinable and as performant as hand-crafted ones.

Introduction

Despite the advent of computationally efficient, automated AI algorithms from the areas of planning, tree search and learning (Orkin 2006; Browne et al. 2012; Champandard 2003), manual authoring remains the predominant way of specifying agent behaviors in the gaming industry. The main behavior authoring techniques for AI designers are finite-state machines (FSMs) and behavior trees (BTs).

Still, hand-coding behaviors has several drawbacks over using automated processes of generating action sequences. The main drawback is the significant authoring effort that an AI designer has to invest in building behaviors. Another drawback is that hand-made state transitions depend on game-specific parameters. Any change in these parameters, for example due to a change in the game's mechanics, possibly requires remodelling of the created behavior and thus further time investment.

The slow adoption of automated approaches can in part be attributed to their high complexity compared to FSMs and BTs. However, another part of the reason why the industry is unwilling to incorporate these techniques into their games is that they have to surrender creative control over the produced behaviors to an algorithm. We acknowledge that designers are reluctant to give up creative control because it raises unwanted issues with reproducibility and quality control (Millington and Funge 2009).

This is precisely the problem we address with our approach. We propose a hybrid solution between automated and manual authoring that uses automation, not to replace manual authoring, but to enhance it. Instead of using opaque AI techniques to produce finalised behaviors at runtime, we use them offline to generate transparent behavior models in form of behavior trees. By doing so we retain the creative control as all generated behaviors are fully transparent, editable and deterministic. In essence, our approach extracts a static, compressed version of the behavior exhibited when using a dynamic, automated method, such as Goal-Oriented Action Planning (GOAP). As a welcome side effect the generated models are as performant as hand-crafted behaviors.

Our approach builds on the principle of identifying *characteristic* sequences. In the context of this paper, a sequence of actions is characteristic for an agent solving a given problem when it is one of the most likely (sub-)sequences that lead to the agent's goal. To find those sequences we observe the automated decision-making algorithm's behavior in a Monte-Carlo simulation. We then employ a genetic algorithm to evolve a behavior tree that outputs similar action sequences to those produced in simulation by the automated decision-making algorithm.

This work presents our first steps towards the ultimate goal of being able to apply this approach to every decision-making algorithm. In this paper we show that GOAP can be used to generate suitable simulation data for our approach. The motivation behind this work is to facilitate the adoption and understanding of automated, opaque AI algorithms by translating them to comprehensible behavior models.

The Problem

We first present a simple example to gain an understanding of the problem we have to solve to find a transparent representation of an opaque algorithm. We use this example to then formulate an abstract definition of the general case in the next section of this paper.

Consider the following (grossly simplified) scenario: We design the behavior of an agent that has just three courses of action at her disposal. Her first option is that she can throw paint balls at other agents until she holds no more paint balls. However, she may only throw paint balls at other agents when they are in her limited throwing range. To be able to keep throwing paint balls she also has the option of restocking paint balls up to the maximum amount of two paint balls. Furthermore, we assume that the stock of available paint balls is infinite and thus the agent may restock them unconditionally. Finally, the last action at her disposal is that she may simply do nothing and idle for one second. Furthermore, we define that success in our scenario means having hit as many agents as possible with paint balls.

We now put ourselves in an AI developer's shoes: How do we find an intelligent behavior model for this agent? The agent can not simply constantly throw paint balls, because her paint ball stock would deplete after two shots. Similarly, it makes no sense for her to throw paint balls when no other agents are in throwing range. Additionally, she certainly does not want to be idling when there are agents in range. These are the undesirable behaviors that we do not want our agent to exhibit.

The next step is to determine the behavior sequences we want to see instead. A straightforward approach to find these sequences is to consider how the situation would play out under different conditions and memorising only those that we deem successful. This constitutes a classic trial-and-error strategy.

We notice that our agent should shoot if an agent is close when considering the cases where the agent knows that there are opponents in range after coming up with just a few scenarios. Similarly, she should restock when her paint balls are depleted. In any other cases she can simply stay idle. Via this simple analysis we naturally discovered the most characteristic sequences of this scenario. Granted, due to the low number of actions it was not hard to come up with sufficiently many cases, but the process remains the same even for larger state spaces. We can now build an intelligent behavior for our agent by prioritising and combining each identified sequence considering its associated preconditions, effects and total cost.

We manually created solution sequences to better demonstrate the underlying thought process in this example. In principle, this is the same process our approach has to go through in order to automatically create behavior models. However, our goal is to have a computer use opaque decision-making algorithms to automatically compute solution sequences for us. The importance of providing automated support for this task becomes clear if we recall that the number of cases an AI designer has to consider grows exponentially with the number of actions of the agent.

Generalisation

In the general case, we can formalise our problem as follows: Firstly, we have a set of possible actions A at an agent's disposal. Secondly, we have a problem domain that we describe in terms of the set of distinct start states S and a designated goal state s_g. And, thirdly, we have a function $f(s)$ representing a decision-making algorithm that, for a start state s, returns an action sequence leading to the goal state. Lastly, we are looking for the *characteristic* action sequences that best capture the essence of algorithm $f(s)$ considering the whole set of start states S.

It follows that our problem θ can be formally modelled as:

$$\theta = (A, S, s_g, f(s))$$

where

$A = \{a_1, ..., a_n\}$, with n ... number of actions
$S = \{s_1, ..., s_m\}$, with m ... number of distinct start states
s_g ... goal state
a_i ... action i
s_i ... start state i
$f(s)$ ... function returning an action sequence to reach g

Furthermore, we characterise actions themselves as having a set of preconditions P, effects E and a cost function $c(s)$. Preconditions p_i describe the conditions imposed on the current agent state for an action to be applicable. Effects e_i describe the change to the current agent state when an action is applied. The cost function $c(s)$ computes the abstract cost associated with applying an action to a state. It follows that we can formalise an action a as:

$$a = (P, E, c(s))$$

where

$P = \{p_1, ..., p_u\}$, with u ... number of preconditions
$E = \{e_1, ..., e_v\}$, with v ... number of effects
p_i ... precondition i
e_i ... effect i
$c(s)$ ... function returning the cost of applying a to s

We define a precondition p as the requirement that a named key k in a given state is equal to value v. To keep things simple, equality is the only logical operation we consider in this paper. Therefore, we can express a precondition as:

$$p = (k, v)$$

where

k ... named key
v ... expected value of k

Finally, we describe an effect e as a pair itself consisting of a set of preconditions P, defining in which cases e is triggered, and a ramification r. Put in abstract terms this means:

$$e = (P, r)$$

where

r ... ramification; the change to the state when e is applied

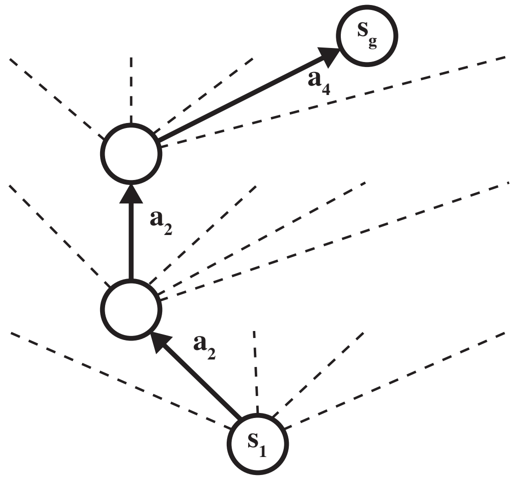

Figure 1: An illustration of the state graph spanning from start state s_1 to the goal state s_g. The bold lines indicate the action sequence $\{a_2, a_2, a_4\}$ returned by $f(s_1)$.

Exploring the State Graph

We can visualise every distinct combination of a start state s_i with the goal state s_g of a problem θ, as described in the previous section, as a state graph. In this graph, nodes represent world states, and directed edges denote actions. We can therefore picture an action sequence leading to the goal state s_g as a path on the state graph. Figure 1 shows an example of such a graph for the start state s_1. The bold arrows highlight the solution $\{a_2, a_2, a_4\}$ computed by evaluating the decision-making algorithm f for s_1.

The number of start states s_i that lead to distinct solutions when applying f has an upper limit under the constraint that the algorithm f and the actions' cost functions c do not depend on any parameters other than the ones present in the model. In this case, we can derive all possible start states directly from our problem's model, since we know which properties, identified by their named key k, of the state can influence the pre- and postconditions of actions. It follows that we have fully captured the behavior of $f(s)$ when applied to a problem θ if we store all the solutions returned for each start state s_i.

Now, two issues arise when we try to capture the complete behavior exhibited by f in this manner. Firstly, since the number of start states rises exponentially with the number of preconditions, computing every possible solution sequence is time-intensive and only feasible for small problems. Secondly, it is not a good idea to simply take all solutions and recombine them into a FSM or BT. Due to the sheer size of the resulting behavior, editing or understanding it would be almost impossible.

Thus, in order to avoid the latter issue, we try to find the sequences that are most likely part of a solution produced by f. Those are the sequences that f emits the most if we assume that each start state is equally likely to happen. We call those sequences characteristic for our problem θ.

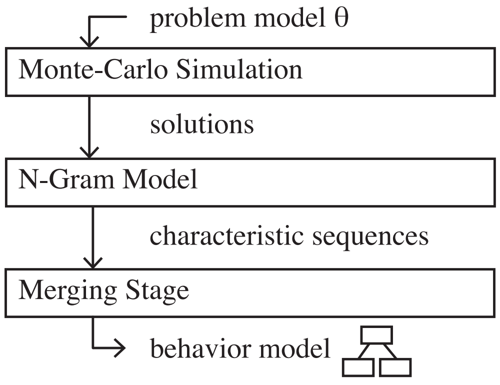

Figure 2: An overview of the presented approach. The boxes represent its three main stages. The text in between the stages denotes their inputs and outputs.

Monte-Carlo Simulation

Our approach is a multistage process and figure 2 depicts an overview of the process. The first stage is about observing the behavior of a decision-making algorithm f. We employ the Monte-Carlo method to choose simulation scenarios for observation. At every simulation step we randomly pick one of the distinct start states s_i of the problem θ and record the solution sequence produced by f. The reason for incorporating Monte-Carlo simulation into the approach is that it allows us to generate behavior models even for large state spaces due to being parametrisable in the number of performed simulations. Adjusting the number of generated solutions allows us to control the execution time of our approach. However, doing so also incurs an accuracy-time tradeoff that we further explore in the latter sections of this paper. However, it is not strictly necessary that you use the Monte-Carlo method. For problems with a low amount of distinct solutions it is possible to explore the whole solution space in acceptable time. A full search can be viable in those cases depending on the throughput of the applied algorithm f. Monte-Carlo simulation is the first stage of our approach and its output is a set of solutions produced by the algorithm f for different world states s_i of θ.

N-Gram Model

Next, we identify the most characteristic sequences in the generated solution set. We do so by building an n-gram model using the data obtained from the Monte-Carlo simulation stage. To generate the model we count every observed n-gram in the generated solution strings and return only those we encounter the most in our simulation data. This stage of the approach is parameterised as well: You choose which lengths of n-grams and how many of the most common ones you want to include in your n-gram model. However, we have not yet thoroughly analysed how this choice affects the outcome of the approach and what parameters

Algorithm 1 Extraction of characteristic n-grams

```
Input:
    $\theta$ ... problem
    $k$ ... the number of simulations,
    $n$ ... the n-grams' length,
    $l$ ... the number of returned top n-grams,
    all $counts$ must be initialised to 0.

Output:
    The extracted $l$ most common action n-grams
    of length $k$.

    $\Omega \leftarrow$ All preconditions P from actions and effects in $\theta$
    for $i \leftarrow 1$ to $k$ do
        $s_{start} \leftarrow$ next_random_permutation( $\Omega$ )
        $\Phi \leftarrow f(s_{start})$
        for each subsequence $\phi$ of length $n$ in $\Phi$ do
            $count_\phi \leftarrow count_\phi + 1$
        end for
    end for
    return the n-grams $\phi$ with the $l$ highest values in $count$
```

produce the best results. Algorithm 1 shows the process of simulating scenarios and extracting n-grams of a fixed length in detail. Building the n-gram model is the second stage of our approach. The model captures the probability distribution of sequences that are part of generated solutions. Thus, the n-grams with the highest amount of observations represent the characteristic sequences of our problem θ.

Merging Stage

The third, and last, stage of our approach is the merging stage. The merging stage combines the identified characteristic n-grams into a behavior tree. Conceptually, this stage focuses on *when* we want to see the characteristic behaviors exhibited, as opposed to the previous stage, that was about finding *which* behaviors are characteristic. Therefore, the question we face at this stage is: How do we determine suitable priorities and transitions between characteristic sequences in a way that is applicable to any problem θ?

Evolutionary Approach

There are no trivial general rules to merge the given characteristic sequences into a behavior tree, since the optimal tree structure depends on both the applied algorithm f and the problem structure of θ. Thus, we need an optimisation strategy that can automatically adapt to these parameters.

We chose to employ a genetic algorithm (Mitchell 1998) to generate behavior models from characteristic action sequences. An advantage of using evolutionary optimisation is that we can readily apply it to trees (Palmer and Kershenbaum 1994). Additionally, genetic algorithms are oblivious to the problem domain they are applied to. We only need to define a measurement of the fitness of a particular behavior tree and the genetic operations of crossover and mutation.

The fitness function in a genetic algorithm measures the performance of a single solution. It is used to direct the population towards better solutions. If we recall that we are trying to capture the essence of the decision-making algorithm a better solution is one that better resembles the behavior of the decision-making algorithm. Thus, we need a metric that describes how closely a behavior tree follows the behavior exhibited by the decision-making algorithm. For such a metric, however, we first need to be able to determine the similarity of two action sequences.

For the purpose of comparison we see two action sequences as strings where every symbol represents a distinct action. When comparing those action strings we want to penalise missing, superfluous and transposed actions equally. Thus, the Damerau-Levenshtein distance (Damerau 1964; Levenshtein 1966) is an apt choice of metric to compare the similarity of two action strings.

Now, we compare the behavior exhibited by a behavior tree with that of our decision-making algorithm by comparing their produced solutions for the same start states s_i. A higher dissimilarity on average means that the behavior tree has a worse coverage of the decision-making algorithm's behavior. The other details of the employed genetic algorithm are:

Greedy initialisation: To start off, we greedily initialise the population with behavior trees formed by randomly combining characteristic sequences. The greedy initialisation jump-starts the evolutionary process as it incorporates the information of our n-gram model into the construction of the initial population. The root node of every initial behavior tree is a selector.

Crossover: Our crossover operator picks a node in the first behavior tree at random and replaces it with a subtree from the second behavior tree.

Mutation: Our mutation operator performs two different types of changes: Firstly, by chance it mutates a random node in the behavior tree. If the random node is a composite node the mutation changes its type, e.g. from sequence to selector. If it is an action the mutation replaces it with another random action. If it is a conditional node the mutation either adds or removes a random precondition from the conditional node. Secondly, the mutation operator adds or removes conditional nodes from the behavior tree. We initialise conditional nodes that are added in this way with a random combination of preconditions from the problem model.

Elitism: The algorithm preserves a number of top behavior trees of a population for the next generation. We employ elitism to ensure the behavior trees always improve over generations.

Example

We use a refined version of our previously mentioned example to showcase the application of our approach. Our refined example involves the same agent we described previously. This time, however, we put her in a different environment: The environment is a typical Capture the Flag scenario consisting of two bases on opposing sides of a playing field. Each base holds a flag that agents strive to capture and carry to their respective home bases. A score point is awarded ev-

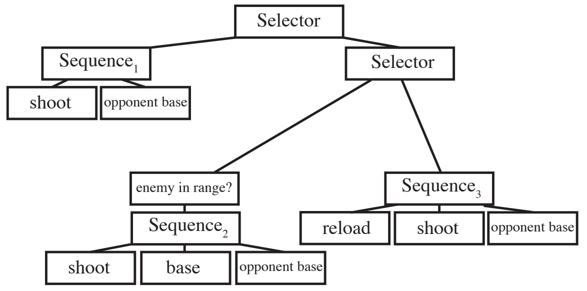

Figure 3: A sample behavior tree generated in 10 generations using 5 characteristic bigrams for the problem θ_{ctf}. Its average Damerau-Levenshtein distance from the solution strings generated by GOAP is 1.25.

ery time an agent carries the other team's flag to her own team's base.

Our agent has the ability to perform multiple different actions: Firstly, our agent can still shoot other agents with paint as long as she has the required ammunition. She also still has the option of restocking. Her new ability is that she can now move. However, to limit the size of the problem space the agent may only move to the enemy flag or her own base. We measure the success of an agent in the Capture the Flag scenario by scored points. Thus, our agent's new main goal is to capture the enemy flag. Furthermore, Goal-Oriented Action Planning (GOAP) is the algorithm f whose behavior we wish to capture in a transparent model.

Formal Model

Following our generalised model, we can formally express the problem from the previous section as:

$$\theta_{ctf} = (A, S, s_g, f(s))$$

where

$A = \{a_1, a_2, a_3, a_4\}$, with
a_1 .. shoot,
a_2 .. restock,
a_3 .. move to base,
a_4 .. move to opponent base
s_g ... state with points scored by our agent
$f(s)$ ... Goal-Oriented Action Planning

The preconditions of the shoot action a_1 are that an enemy is in range and that the agent has ammunition. Its first effect is that the agent's ammunition is spent. Its second effect is that the agent hits the enemy in range with paint. Agents that are hit with paint are removed from the playing field for a small amount of time. Thus, after the shoot action the shot enemy is not in throwing range anymore.

The restock action a_2 has no precondition. Its effect is that the agent restores her ammunition.

The movement actions a_3 and a_4 have no preconditions. Their effects are that the agent's position is changed to either the agent's own base (for a_3) or the opponent's base (for a_4).

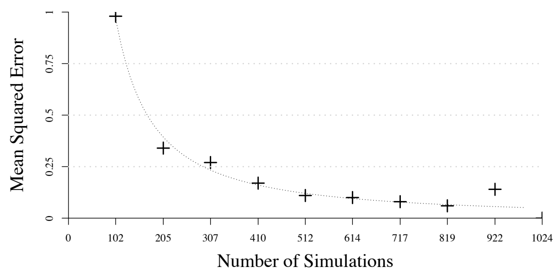

Figure 4: A graph showing the mean squared error (MSE) of bigram probabilities of a problem with 1024 distinct start states s_i for different numbers of simulations. The dotted trend line indicates the approximate time-accuracy tradeoff. We use the actual distribution to calculate the MSE.

The cost function returns a constant value corresponding to the time spent reloading and shooting for a_1 and a_2, respectively. For the movement actions a_3 and a_4 we use a dynamic cost function that increases the cost proportionately with the distance of the movement target from the agent's current position. However, the cost of moving the agent increases dramatically if the current state indicates that there is an opponent in her throwing range, because she is in danger of getting shot.

Qualitative Evaluation

To evaluate our approach qualitatively we implemented it in the PALAIS testbed environment (Schwab and Hlavacs 2015). Figure 3 shows a sample behavior tree that our approach generated for the problem θ_{ctf} using 5 characteristic bigrams. Note that we omit the conditional nodes preceding each action for brevity and that our agent reevaluates the root node indefinitely if it exits.

Our approach successfully identified and preserved the most obvious, characteristic sequences in this sample behavior tree. Namely those sequences are (restock, shoot) and (base, opponent base). An interesting point to note is that our approach output Sequence$_3$ (in the rightmost subtree) in the order reload first, then shoot, instead of the more natural order shoot first, then reload. This makes sense, however, if we consider that reloading first is more general because it also captures the states where the agent has no ammunition.

To our detriment, our approach also produced a superfluous conditional node above Sequence$_2$. The condition is unnecessary since performing the first action of the sequence is not possible if there is no enemy in range. The problem of superfluous elements in output behavior trees could be addressed by adding a post-processing stage to the approach or by penalising superfluous elements in the fitness function.

Quantitative Evaluation

To evaluate our approach quantitatively we generated a difficult problem with 10 distinct actions. On average each of these actions has 3.8 effects, 1.7 postconditions and 6.7 dependent variables. In total there are 10 unique preconditions

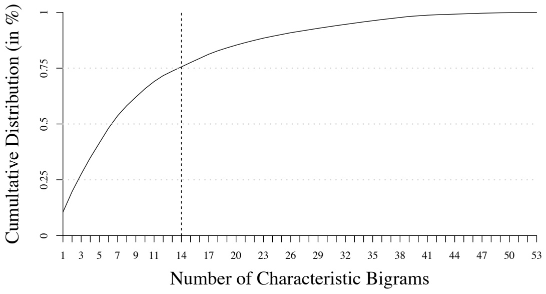

Figure 5: A graph showing the relative cumulative distribution of the top bigrams in all of the solutions generated during our simulation of a problem containing 1024 distinct start states s_1. The dashed line marks the top 14 bigrams at which point more than 75% of the solutions could theoretically be reproduced with optimal transitions between bigrams. The bigrams are sorted from most to least common.

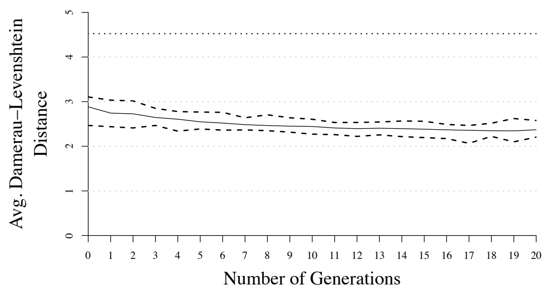

Figure 6: A graph illustrating the change in the average fitness of the best behavior trees of 30 runs when adjusting the number of generations. The two dashed lines show the minimum and maximum observed values. The dotted line indicates the average distance of a randomly generated set of solutions which represents the worst possible output.

in actions and effects, which results in $2^{10} = 1024$ distinct start states s_i. We use the top 25 bigrams to model the characteristic sequences of this problem. Moreover, we configured our genetic algorithm to use a population size of 300 behavior trees with a mutation rate of 15%, a crossover rate of 50% and 2% of the top behavior trees being preserved every generation as elitists. Additionally, 35% of the performed mutations add or remove conditional nodes. This section answers a question related to the performance of each stage of our approach:

- How does the number of Monte-Carlo simulations affect the accuracy of our generated n-gram model?
- How well does the the n-gram model capture the whole set of solutions generated by f?
- How does our merging stage perform with regards to the worst-case scenario?

Figure 4 shows the results of our analysis of the time-accuracy tradeoff incurred by increasing the number of simulations performed in the Monte-Carlo simulation. The raising MSE at 922 simulations can be attributed to the error introduced by sampling a single solution multiple times, because at this point the number of samples amounts to around 90% of the solution space's size. The graph indicates that the estimated bigram probabilities converge quickly towards the actual probabilities.

Figure 5 shows the, theoretically possible, optimal performance of our n-gram model using the most commonly observed bigrams. The graph tells us that we can produce very similar results to GOAP with just a small set of characteristic sequences, if we can deduce the optimal transitions.

Figure 6 shows how our merging stage performs with an increasing number of generations and 200 simulated solutions to work with. We include the dotted line on top, indicating the fitness of random action selection as a worst-case scenario, for reference. The graph shows a steady increase in the average fitness of the generated behavior trees. However, the performance gain slows down significantly after about 7 generations. Furthermore, the average fitness of around 3 at generation 0, before any evolutionary optimisation, indicates that our greedy initialisation step already constitutes a substantial improvement over the worst-case scenario.

Related Work

Evolutionary approaches have been applied to build decision trees in the context of classification (Barros et al. 2012). By nature, the induction of decision trees for classification is similar to the problem we present in this paper. However, the evolution of a behavior tree is more complicated because we evaluate their performance in dependant sequences as opposed to singular independent decisions. (Lim, Baumgarten, and Colton 2010; Perez et al. 2011) used evolutionary algorithms to evolve game-specific behavior trees with the goal of producing optimal behaviors. In contrast, our approach focuses on the extraction of comprehensible, transparent behavior trees from a decision-making algorithm.

Conclusion and Future Work

We presented a novel approach that extracts comprehensible behavior models from decision-making algorithms. We showed that GOAP is a decision-making algorithm that can be used with our approach. Furthermore, the evaluation of our approach demonstrated that it is able to deduce behavior trees that result in action sequences that are, to a good degree, similar to the ones exhibited GOAP. However, the data suggests that further improvement of our approach is possible. Future work will include the expansion of our approach to make it applicable to other decision-making algorithms.

References

Barros, R. C.; Basgalupp, M. P.; De Carvalho, A.; and Freitas, A. A. 2012. A survey of evolutionary algorithms for decision-tree induction. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on* 42(3):291–312.

Browne, C. B.; Powley, E.; Whitehouse, D.; Lucas, S. M.; Cowling, P. I.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S. 2012. A survey of Monte Carlo tree search methods. *Computational Intelligence and AI in Games, IEEE Transactions on* 4(1):1–43.

Champandard, A. J. 2003. *AI game development: Synthetic creatures with learning and reactive behaviors*. New Riders.

Damerau, F. J. 1964. A technique for computer detection and correction of spelling errors. *Commun. ACM* 7(3):171–176.

Levenshtein, V. I. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, 707–710.

Lim, C.-U.; Baumgarten, R.; and Colton, S. 2010. Evolving behaviour trees for the commercial game defcon. In *Applications of evolutionary computation*. Springer. 100–110.

Millington, I., and Funge, J. 2009. *Artificial intelligence for games*. CRC Press.

Mitchell, M. 1998. *An introduction to genetic algorithms*. MIT press.

Orkin, J. 2006. Three states and a plan: the AI of FEAR. In *Game Developers Conference*, volume 2006, 4.

Palmer, C. C., and Kershenbaum, A. 1994. Representing trees in genetic algorithms. In *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*, 379–384. IEEE.

Perez, D.; Nicolau, M.; ONeill, M.; and Brabazon, A. 2011. Evolving behaviour trees for the mario ai competition using grammatical evolution. In *Applications of Evolutionary Computation*. Springer. 123–132.

Schwab, P., and Hlavacs, H. 2015. Palais: A 3d simulation environment for artificial intelligence in games. In *Proceedings of the AISB Convention 2015*.