

Automatically Generating Game Tactics through Evolutionary Learning

Marc Ponsen, Héctor Muñoz-Avila,
Pieter Spronck, and David W. Aha

■ The decision-making process of computer-controlled opponents in video games is called *game AI*. Adaptive game AI can improve the entertainment value of games by allowing computer-controlled opponents to fix weaknesses automatically in the game AI and to respond to changes in human-player tactics. Dynamic scripting is a reinforcement learning approach to adaptive game AI that learns, during gameplay, which game tactics an opponent should select to play effectively. In previous work, the tactics used by dynamic scripting were designed manually. We introduce the evolutionary state-based tactics generator (ESTG), which uses an evolutionary algorithm to generate tactics automatically. Experimental results show that ESTG improves dynamic scripting's performance in a real-time strategy game. We conclude that high-quality domain knowledge can be automatically generated for strong adaptive game AI opponents. Game developers can benefit from applying ESTG, as it considerably reduces the time and effort needed to create adaptive game AI.

Today's video games are becoming increasingly realistic, especially in terms of the graphical presentation of the virtual world in which the game is situated. To further increase realism, characters "living" in these virtual worlds must be able to reason effectively. The term *game AI* refers to the decision-making process of computer controlled opponents. Both game industry practitioners (Rabin 2004)

and academics (Laird and van Lent 2000) predict an increasing importance of game AI. High-quality game AI will increase the game playing challenge (Nareyek, 2004) and is a potential selling point of a game. However, the time allocated to develop game AI is typically short; most game companies assign graphics and storytelling the highest priorities and do not implement the game AI until the end of the development process (Nareyek 2004). This complicates designing and testing strong game AI (that is, game AI that is effective in winning the game). Thus, even in state-of-the-art games, game AI is generally of inferior quality (Schaeffer 2001).

Adaptive game AI, which concerns methods for automatically adapting the behavior of computer-controlled opponents, can potentially increase the quality of game AI. *Dynamic scripting* is a reinforcement learning technique for implementing adaptive game AI (Spronck, Sprinkhuizen-Kuyper, and Postma 2004). We apply dynamic scripting to learn a policy for the complex real-time strategy (RTS) game *War-gus*. Dynamic scripting employs extensive domain knowledge in the form of knowledge bases containing tactics (that is, sequences of primitive actions). Manually designing these knowledge bases may be time intensive, and risks errors in analysis and encoding. We introduce a novel methodology, implemented in the evolutionary state-based tactics generator

(ESTG), which uses an evolutionary algorithm to generate tactics to be used by dynamic scripting automatically. Our empirical results show that dynamic scripting equipped with the evolved tactics can successfully adapt (that is, learn a winning policy) to static opponents.

In this article, we first describe related work. We then introduce RTS games and the game environment selected for the experiments. Next, we discuss our RTS implementation for dynamic scripting and the ESTG method for automatically generating the dynamic scripting knowledge bases. Finally, we describe our experimental results and draw conclusions.

Related Work

AI researchers have shown that successful adaptive game AI is feasible under the condition that it is applied to a limited game scenario or that appropriate abstractions and generalizations are assumed.

Demasi and Cruz (2002) used an evolutionary algorithm to adapt the behavior of opponent agents in an action game. They reported fast conversion to successful behavior, but their agents were limited to recognizing three ternary state parameters and making a choice out of only four different actions. Guestrin et al. (2003) applied relational Markov decision process models for some limited RTS game scenarios, for example, three on three combat. Cheng and Thawonmas (2004) proposed a case-based plan-recognition approach for assisting RTS players but only for low-level management tasks. In contrast, we focus on the highly complex learning task of winning complete RTS games.

Spronck, Sprinkhuizen-Kuyper, and Postma (2004) and Ponsen and Spronck (2004) implemented a reinforcement learning (RL) technique tailored for video games called *dynamic scripting*. They report good learning performances on the challenging task of winning video games. However, dynamic scripting requires a considerably reduced state and action space to be able to adapt sufficiently fast. Ponsen and Spronck (2004) evolved high-quality domain knowledge in the domain of RTS games with an evolutionary algorithm and used this to manually design game tactics (stored in knowledge bases). In contrast, in the present work we generate the tactics for the knowledge bases fully automatically. Aha, Molineaux, and Ponsen (2005) build on the work of Ponsen and Spronck (2004) by using a case-based reasoning technique that learns which evolved tactics are appropriate given the state and opponent. Marthi, Russell, and Latham

(2005) applied hierarchical reinforcement learning in a limited RTS domain. Their action space consisted of partial programs, essentially high-level preprogrammed behaviors with a number of choice points that can be learned using Q-learning. Our tactics bear a strong resemblance to their partial programs: both are preprogrammed, temporally extended actions that can be invoked on a higher level.

Real-Time Strategy Games

RTS is a category of strategy games that focus on military combat. For our experiments, we selected the RTS game Wargus, which is built on Stratagus, an open-source engine for RTS games. Wargus (illustrated in figure 1) is a clone of the popular game Warcraft II. RTS games such as Warcraft II require the player to control armies (consisting of different types of units) to defeat all opposing forces that are situated in a virtual battlefield (often called a map) in real time.

In most RTS games, the key to winning lies in efficiently collecting and managing resources and appropriately allocating these resources over the various action elements. Typically, the game AI in RTS games, which determines all decisions for a computer opponent over the course of the whole game, is encoded in the form of *scripts*, which are lists of actions that are executed sequentially. We define an *action* as an atomic transformation in the game situation. Typical actions in RTS games include constructing buildings, researching new technologies, and combat.

Both human and computer players can use these actions to form their game strategy and tactics. We will employ the following definitions in this article: a *tactic* is a sequence consisting of one or more primitive actions (for example, constructing a blacksmith shop and acquiring all related technologies for that building), and a *strategy* is a sequence of tactics that can be used to play a complete game. Designing strong RTS strategies is a challenging task. RTS games include only partially observable environments, which contain adversaries that modify the state asynchronously, and whose decision models are unknown, thereby making it infeasible to obtain complete information on the current situation. In addition, RTS games include an enormous number of possible actions that can be executed at any given time, and some of their effects on the state are uncertain. Also, to successfully play an RTS game, players must make their decisions under time constraints due to the real-time game flow. These properties of RTS games make them a challenging domain for AI research.



Figure 1. Screenshot of a Battle in the RTS Game Wargus.

Reinforcement Learning with Dynamic Scripting

In reinforcement learning problems, an adaptive agent interacts with its environment and iteratively learns a policy, that is, it learns *what* to do *when* to achieve a certain goal, based on a scalar reward signal it receives from the environment (Sutton and Barto 1998; Kaelbling, Littman, and Moore 1996). Policies can be represented in a tabular format, where each cell includes a state or state-action value representing, respectively, the desirability of being in a state or the desirability of choosing an action in a state. Several approaches have been defined for learning optimal policies, such as dynamic programming, Monte Carlo methods, and temporal-difference (TD) learning methods (Sutton and Barto 1998).

Dynamic scripting (Spronck et al. 2004) is a reinforcement learning technique designed for

creating adaptive video game agents. It employs on-policy value iteration to optimize state-action values based solely on a scalar reward signal. Consequently, it is concerned only with maximizing immediate rewards. Action selection is implemented with a softmax method (Sutton and Barto 1998). The reward in the dynamic scripting framework is typically designed with prior knowledge of how to achieve a certain goal and causes high discrepancies in the state-action values. Consequently, this will lead to faster exploitation; that is, the chance that the greedy action is selected increases.

Dynamic scripting has been designed so that adaptive agents start exploiting knowledge only in a few trials. It allows balancing exploitation and exploration by maintaining a minimum and maximum selection probability for all actions. Elementary solution methods such as TD learning or Monte-Carlo learning update state-action values only after they are executed

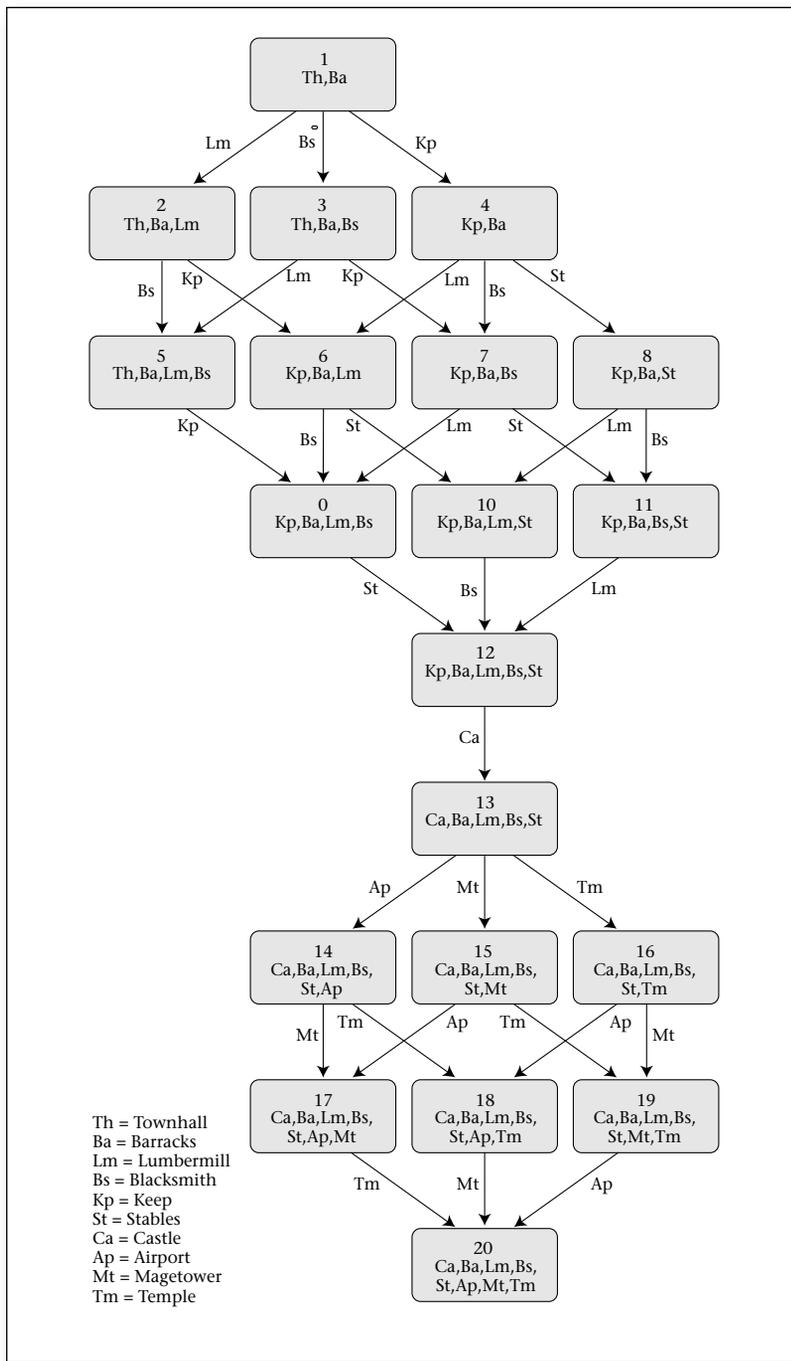


Figure 2. A Building-Specific State Lattice for Wargus.

Nodes represent states (defined by a set of completed buildings), and state transitions involve constructing a specific building.

(Sutton and Barto 1998). In contrast, dynamic scripting updates all state-action values in a specific state through a redistribution process (Spronck et al. 2004), so that the sum of the state-action values remains constant.

Because of these properties, dynamic scripting cannot guarantee convergence. This actu-

ally is essential for its successful use in video games. The learning task in a game constantly changes (for example, an opponent player may choose to switch tactics), thus aiming for an optimal policy may result in overfitting to a specific strategy. Dynamic scripting is capable of generating a variety of behaviors and responding quickly to changing game dynamics.

Dynamic Scripting in Wargus

In this section we will detail our dynamic scripting implementation in the RTS game Wargus. In Wargus, we play an agent controlled by dynamic scripting, henceforth called the *adaptive agent*, against a *static agent*. Both agents start with a town hall, barracks, and several units. The static agent executes a static script (representing a strategy), while the adaptive agent generates scripts on the fly based on its current policy. We will next describe our representation of the state space in Wargus and detail the policy update process.

States and their Knowledge Bases

Typically, players in an RTS game such as Wargus start with few admissible actions available to them. As players progress, they acquire a larger arsenal of weapons, units, and buildings. The tactics that can be used in an RTS game mainly depend on the availability of different unit types and technologies.

We divided the Wargus game into a small number of abstract states. Each state corresponds to a unique knowledge base whose tactics can be selected by dynamic scripting when the game is in that particular state. We distinguish states according to types of available buildings (see figure 2), which in turn determine the unit types that can be built and the technologies that can be researched. Consequently, state changes are spawned by tactics that create new buildings.

Dynamic scripting starts by selecting tactics for the first state. When a tactic is selected that spawns a state change, tactics will then be selected for the new state. To avoid monotonous behavior, each tactic is restricted to be selected only once per state. Tactic selection continues until either a total of N tactics is selected ($N = 100$ was used for the experiments) or until final state 20 (see figure 2) is reached. For this state in which the player possesses all relevant buildings, a maximum of M tactics must be selected ($M = 20$ was used for the experiments) before the script moves into a repeating cycle (called the *attack loop*), which continuously initiates attacks on the opponents.

Weight Value Adaptation

For each tactic in a state-specific knowledge base, dynamic scripting maintains an associated weight value that indicates the desirability of choosing that tactic in the specific state. At the start of our experiments weight values of all tactics are initialized to 100. After each game, the weight values of all tactics employed are updated. The magnitude of the weight adjustments in a state is uniformly distributed over the nonselected tactics for that state. The size of weight value updates is determined mainly by a *state reward*, that is, an evaluation of the performance of the adaptive agent during a certain state. To recognize the importance of winning or losing the game, weight value updates also take into account a *global reward*, that is, an evaluation of the performance of the adaptive agent for the game as a whole.

The state reward function R_i for state i , $i \in \mathbb{N}_0$, for the adaptive agent a yields a value in the range $[0, 1]$ and is defined in equation 1.

$$R_i = \frac{(S_{a,i} - S_{a,i-1})}{(S_{a,i} - S_{a,i-1}) + (S_{s,i} - S_{s,i-1})} \quad (1)$$

In equation 1, $S_{a,x}$ represents the score of the adaptive agent a after state x , $S_{s,x}$ represents the score of the static agent s after state x , $S_{a,0} = 0$, and $S_{s,0} = 0$. The score is a value that measures the success of an agent up to the moment the score is calculated. The score never decreases during game play.

The global reward function R_∞ for the adaptive agent a yields a value in the range $[0, 1]$, and it is defined in equation 2.

$$R_\infty = \begin{cases} \min\left(\frac{S_{a,L}}{S_{a,L} + S_{s,L}}, b\right) & \text{if } a \text{ lost,} \\ \max\left(\frac{S_{a,L}}{S_{a,L} + S_{s,L}}, b\right) & \text{if } a \text{ won.} \end{cases} \quad (2)$$

In equation 2, $S_{a,x}$ and $S_{s,x}$ are as in equation 1, L is the number of the state in which the game ended, and $b \in (0, 1)$ is the break-even point. At this point the weight values remain unchanged. The score function is domain dependent and should reflect the relative strength of the two opposing agents in the game. For Wargus, the score $S_{x,y}$ for agent x after state y is defined in equation 3.

$$S_{x,y} = CM_{x,y} + (1 - C)B_{x,y} \quad (3)$$

In equation 3, for agent x after state y , $M_{x,y}$ represents the *military points* scored, that is, the number of points awarded for killing units and destroying buildings, and $B_{x,y}$ represents the *building points* scored, that is, the number of points awarded for conscripting units and con-

structing buildings. The constant $C \in [0, 1]$ represents the weight given to the military points in the score function. Since experience indicates that military points are a better indication for the success of a tactic than building points, C is set to 0.7. Weight values are bounded by a range $[W_{min}, W_{max}]$. A new weight value is calculated as $W + \Delta W$, where W is the original weight value, and the weight adjustment ΔW is expressed by equation 4.

$$\Delta W = \begin{cases} -P_{max} \left(C_{end} \frac{b-R_\infty}{b} + (1 - C_{end}) \frac{b-R_i}{b} \right) \\ R_{max} \left(C_{end} \frac{R_\infty-b}{1-b} + (1 - C_{end}) \frac{R_i-b}{1-b} \right) \end{cases} \quad (4)$$

In equation 4, $R_{max} \in \mathbb{N}$ and $P_{max} \in \mathbb{N}$ are the maximum reward and maximum penalty, respectively, R_∞ is the global reward, R_i is the state reward for the state corresponding to the knowledge base containing the weight value, and b is the break-even point. For the experiments in this article, we set P_{max} to 400, R_{max} to 400, W_{max} to 4000, W_{min} to 25, and b to 0.5. The constant $C_{end} \in [0, 1]$ represents the fraction of the weight value adjustment that is determined by the global reward. It is desirable that, even if a game is lost, knowledge bases for states where performance was successful are not punished (too much). Therefore, C_{end} was set to 0.3, that is, the contribution of the state reward R_i to the weight adjustment is larger than the contribution of the global reward R_∞ .

Automatically Generating Tactics

The evolutionary state-based tactics generator (ESTG) method automatically generates knowledge bases for use by dynamic scripting. The ESTG process is illustrated in figure 3.

The first step (called EA, for *evolutionary algorithm*) uses an evolutionary algorithm to search for strategies that defeat specific opponent strategies. This step of the process is similar to experiments described by Ponsen and Spronck (2004). The opponent strategies are provided to EA as a training set, which is the only manual input ESTG requires. In our experiments, the training set contains 40 different strategies. Four of these are static scripts that were designed by the Wargus developers. Static scripts are usually of high quality because they are recorded from human player strategies. The remaining 36 strategies in our training set are evolutionary scripts, that is, previously evolved strategies that we will use as an opponent strategy. The output of EA is a set of counterstrategies.

The second step (called KT, for *knowledge transfer*) involves a state-based knowledge transfer from evolved strategies to tactics. Finally, we empirically validate the effectiveness

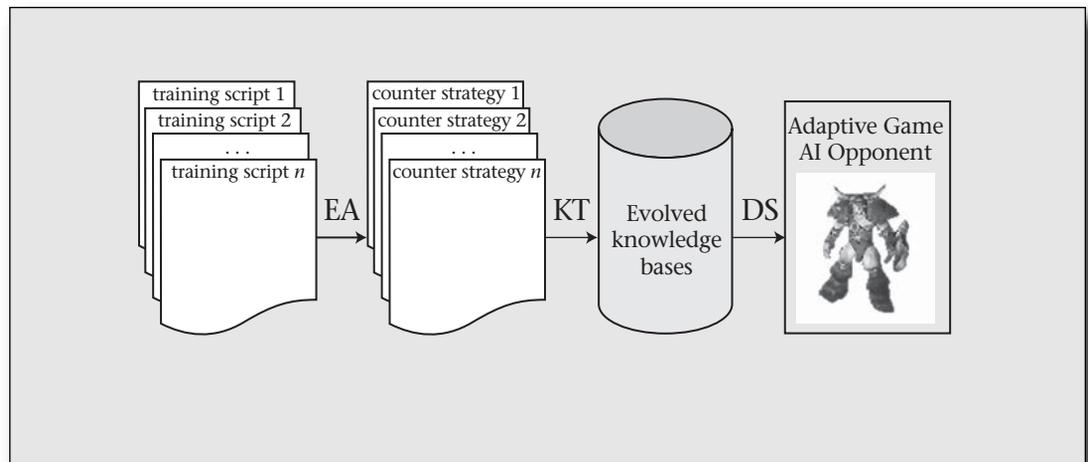


Figure 3. Schematic Representation of ESTG.

of the evolved tactics by testing them with dynamic scripting (DS). The evaluation with dynamic scripting is not a necessary part of the ESTG process, because other machine-learning techniques may also be used; for example, the case-based reasoning algorithm in Aha, Moliniaux, and Ponsen (2005) also used tactics evolved with ESTG.

EA: Evolving Domain Knowledge

To specify the evolutionary algorithm used in the EA step, we will discuss the chromosome encoding, the fitness function, and the genetic operators.

Chromosome Encoding

EA works with a population of chromosomes (in our experiments we use a population of size 50), each of which represents a static strategy. Figure 4 shows the chromosome's design. The chromosome is divided into the 20 states as defined earlier (see figure 2). States include a state marker followed by the state number and a series of genes. Each gene in the chromosome represents a game action. Four different gene types exist, corresponding to the available actions in Wargus, namely (1) build genes, (2) research genes, (3) economy genes, and (4) combat genes. Each gene consists of a gene ID that indicates the gene's type (B, R, E, and C, respectively), followed by values for the parameters needed by the gene. Chromosomes for the initial population are generated randomly. A partial example chromosome is shown at the bottom of figure 4.

Fitness Function

To determine the fitness of a chromosome, the chromosome is translated to a game AI script and played against a script in the training set. A fitness function measures the relative success

of the game AI script represented by the chromosome. The fitness function F for the adaptive agent a (controlled by the evolved game script) yields a value in the range $[0, 1]$ and is defined in equation 5.

$$F = \begin{cases} \min\left(\frac{C_T}{C_{max}} \cdot \frac{M_a}{M_a + M_s}, b\right) & \text{if } a \text{ lost,} \\ \max\left(\frac{M_a}{M_a + M_s}, b\right) & \text{if } a \text{ won.} \end{cases} \quad (5)$$

In equation 5, C_T represents the time step at which the game was finished (that is, lost by one of the agents or aborted because time ran out), C_{max} represents the maximum time step the game is allowed to continue to, M_a represents the military points for the adaptive agent, M_s represents the military points for the adaptive agent's opponent, and b is the break-even point. The factor C_T/C_{max} ensures that a game AI script that loses after a long game is awarded a higher fitness than a game AI script that loses after a short game. Our goal is to generate a chromosome with a fitness exceeding a target value. When such a chromosome is found, the evolution process ends. This is the fitness stop criterion. For our experiments, we set the target value to 0.7. Because there is no guarantee that a chromosome exceeding the target value will be found, evolution also ends after it has generated a maximum number of chromosomes. This is the run-stop criterion. We set the maximum number of chromosomes to 250. The choices for the fitness-stop and run-stop criteria were determined during preliminary experiments.

Genetic Operators

Relatively successful chromosomes (as determined by equation 5) are allowed to breed. To

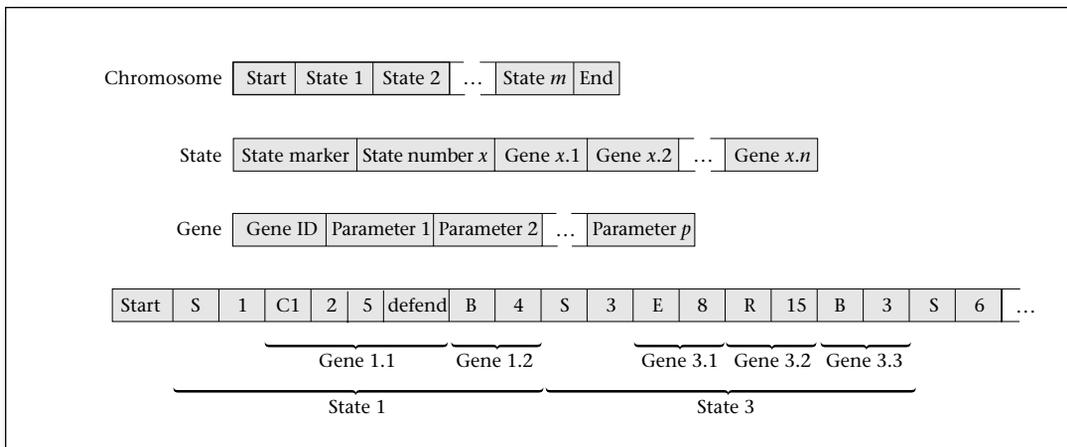


Figure 4. Design of a Chromosome to Store a Game AI Script in Wargus.

select parent chromosomes for breeding, we use size-3 tournament selection. This method prevents early convergence, and it is computationally fast. Newly generated chromosomes replace existing chromosomes in the population, using size-3 crowding (Goldberg 1989). To breed new chromosomes, we implemented four genetic operators: (1) *state crossover*, which selects two parents and copies states from either parent to the child chromosome, (2) *gene replace mutation*, which selects one parent, and replaces economy, research, or combat genes with a 25 percent probability, (3) *gene biased mutation*, which selects one parent and mutates parameters for existing economy or combat genes with a 50 percent probability, and (4) *randomization*, which randomly generates a new chromosome. Randomization has a 10 percent chance of being selected during an evolution. The other genetic operators have a 30 percent chance. By design, all four ensure that a child chromosome always represents a legal game AI.

The genetic operators take into account *activated* genes, which represent actions that were executed when fitness was assessed. Nonactivated genes are irrelevant to the chromosome. If a genetic operator produces a child chromosome that is equal to a parent chromosome for all activated genes, then this child is rejected and a new child is generated.

KT: State-Based Knowledge Transfer

ESTG automatically recognizes and extracts tactics from the evolved chromosomes and inserts these into state-specific knowledge bases. The possible tactics during a game mainly depend on the available units and technology, which in RTS games typically depend on the

buildings that the player possesses. Thus, we distinguish tactics using the Wargus states displayed in figure 2. All genes grouped in an activated state (which includes at least one activated gene) in the chromosomes are considered to be a single tactic. The example chromosome in figure 4 displays two tactics. The first tactic for state 1 includes genes 1.1 (a combat gene that trains a defensive army) and 1.2 (a build gene that constructs a blacksmith shop). This tactic will be inserted into the knowledge base for state 1. Because gene 1.2 spawns a state change, the next genes will be part of a tactic for state 3 (that is, constructing a blacksmith causes a transition to state 3, as indicated by the state marker in the example chromosome).

Experimental Evaluation

Through the EA and KT steps, ESTG generates knowledge bases. The quality of these knowledge bases is evaluated with dynamic scripting (DS).

Crafting the Evolved Knowledge Bases

We evolved 40 chromosomes against the strategies provided in the training set. The EA was able to find a strong counterstrategy against each strategy in the training set. All chromosomes had a fitness score higher than 0.7 (as calculated with equation 5), which represents a clear victory.

In the KT step, the 40 evolved chromosomes produced 164 tactics that were added to the evolved knowledge bases for their corresponding state. We noticed that no tactics were found for some of the later states. All games in the evolution process ended before the adaptive agent constructed all buildings, which explains

why these later states were not included. By design, the AI controlled by dynamic scripting will only visit states in which tactics are available and will ignore other states.

Performance of Dynamic Scripting

We evaluated the performance of the adaptive agent (controlled by dynamic scripting using the evolved knowledge bases) in Wargus by playing it against a static agent. Each game lasted until one of the agents was defeated or until a certain period of time had elapsed. If the game ended due to the time restriction, the agent with the highest score was considered to have won. After each game, the adaptive agent's policy was adapted. A sequence of 100 games constituted one experiment. We ran 10 experiments each against four different strategies for the static agent:

Strategies 1 and 2. Small / Large Balanced Land Attack (SBLA / LBLA). These two strategies focus on land combat, maintaining a balance between offensive actions, defensive actions, and research. SBLA is applied on a small map (64 by 64 cells) and LBLA is applied on a large map (128 by 128 cells).

Strategy 3. Soldier's Rush (SR): This strategy attempts to overwhelm the opponent with cheap offensive units in an early state. Because SR works best in fast games, we tested it on a small map.

Strategy 4. Knight's Rush (KR): This strategy attempts to quickly advance technologically, launching large offenses as soon as powerful units are available. Because KR works best in slower-paced games, we tested it on a large map.

To quantify the relative performance of the adaptive agent against the static agent, we used the randomization turning point (RTP), which is measured as follows. After each game, a randomization test (Cohen 1996) was performed using the global reward values over the last 10 games, with the null hypothesis that both agents are equally strong. The adaptive agent was said to outperform the static agent if the randomization test concluded that the null hypothesis can be rejected with 90 percent probability in favor of the adaptive agent. RTP is the number of the first game in which the adaptive agent outperforms the static agent. A low RTP value indicates good efficiency for dynamic scripting. Ponsen and Spronck (2004) manually improved existing knowledge bases (referred to as the *semiautomatic* approach) from counterstrategies that were evolved offline and tested dynamic scripting against SBLA, LBLA, SR, and KR.

We ran new experiments with dynamic scripting against SBLA, LBLA, SR, and KR, now

using the automatically evolved knowledge bases found with the ESTG method (referred to as the *automatic* approach). The results for dynamic scripting with the two competing approaches are shown in figure 5. From the figure, we conclude that the performance of dynamic scripting improved with the evolved knowledge bases against all previously tested scripts, except for KR; RTP values against these scripts have substantially decreased. Dynamic scripting with the evolved knowledge bases outperforms both balanced scripts before any learning occurs (for example, before weight values are adapted). In previous experiments against the SR, dynamic scripting was unable to find an RTP. In contrast, dynamic scripting using the evolved knowledge bases recorded an average RTP of 51 against SR.

We believe that dynamic scripting's increased performance, compared to our earlier experiments (Ponsen and Spronck 2004), occurred for two reasons. First, the evolved knowledge bases were not restricted to the (potentially poor) domain knowledge provided by the designer (in earlier experiments, the knowledge bases were manually designed and manually "improved"). Second, the automatically generated knowledge bases include tactics that consist of multiple primitive actions, whereas the knowledge bases used in earlier experiments mostly include tactics that consist of a single primitive action. Knowledge bases consisting of compound tactics (that is, an effective combination of fine-tuned actions) reduce the search complexity in Wargus allowing dynamic scripting to achieve fast adaptation against many static opponents.

The Issue of Generalization

The automatic approach produced the best results with dynamic scripting. However, it is possible that the resulting knowledge bases were tailored for specific game AI strategies (that is, the ones received as input for the ESTG method). In particular, scripts 1 to 4 (SBLA, LBLA, SR, and KR) were both in the training and test sets. We ran additional experiments against scripts that were not in the training set. As part of a game programming class at Lehigh University, students were asked to create Wargus game scripts for a tournament. To qualify for the tournament, students needed to generate scripts that defeat scripts 1 to 4 in a predefined map. The top four competitors in the tournament (SC1–SC4) were used for testing against dynamic scripting. During the tournament, we learned that the large map was unbalanced (that is, one starting location was superior over the other). Therefore, we tested the student scripts on the

small map. Dynamic scripting using the evolved knowledge bases was played against the new student scripts. The experimental parameters for dynamic scripting were unchanged. Figure 6 illustrates the results. From the figure it can be concluded that dynamic scripting is able to generalize against strong strategies that were not in the training set. Only the champion script puts up a good fight; the others are already defeated from the start.

Conclusions

In this article, we proposed a methodology (implemented as ESTG) that can automatically evolve knowledge bases of state-based tactics (that is, temporally extended actions) for dynamic scripting, a reinforcement learning method that scales to computer game complexity. We applied it to the creation of an adaptive opponent for *Wargus*, a clone of the popular *Warcraft II* game. From our empirical results we showed that the automatically evolved knowledge bases improved the performance of dynamic scripting against the four static opponents that were used in previous experiments (Ponsen and Spronck 2004). We also tested it against four new opponents that were manually designed. The results demonstrated that dynamic scripting using the ESTG evolved knowledge bases can adapt to many different static strategies, even to previously unseen ones. We therefore conclude that ESTG evolves high-quality tactics that can be used to generate strong adaptive AI opponents in RTS games.

Acknowledgments

Marc Ponsen and Héctor Muñoz-Avila were sponsored by DARPA and managed by NRL under grant N00173-06-1-G005. The views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of DARPA, NRL, or the U.S. government. Pieter Spronck is funded by a grant from the Netherlands Organization for Scientific Research (NWO grant No 612.066.406).

References

- Aha, D.; Molineaux, M.; and Ponsen, M. 2005. Learning to Win: Case-Based Plan Selection in a Real-Time Strategy Game. In *Proceedings of 6th International Conference on Case-Based Reasoning (ICCB-05)*, Lecture Notes in Computer Science 3620, 5–20. Berlin: Springer-Verlag.
- Cheng, D., and Thawonmas, R. (2004). Case-Based Plan Recognition for Real-Time Strategy Games. In *Proceedings of the 5th International Conference on Intelligent Games and Simulation (GAME-ON-04)*, 36–40.

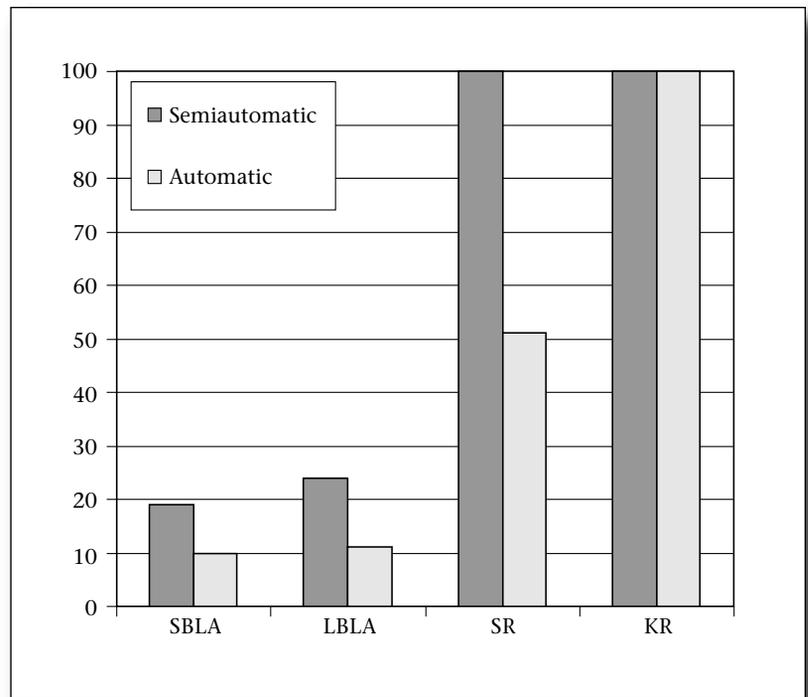


Figure 5. Recorded Average RTP Values for Two Competing Approaches.

Values represent 10 experiments for the two competing approaches. The x-axis lists the opponent strategies. The y-axis represents the average RTP value. A low RTP value indicates good efficiency for dynamic scripting. The three bars that reached 100 represent runs where no RTP was found (for example, dynamic scripting was unable to statistically outperform the specified opponent).

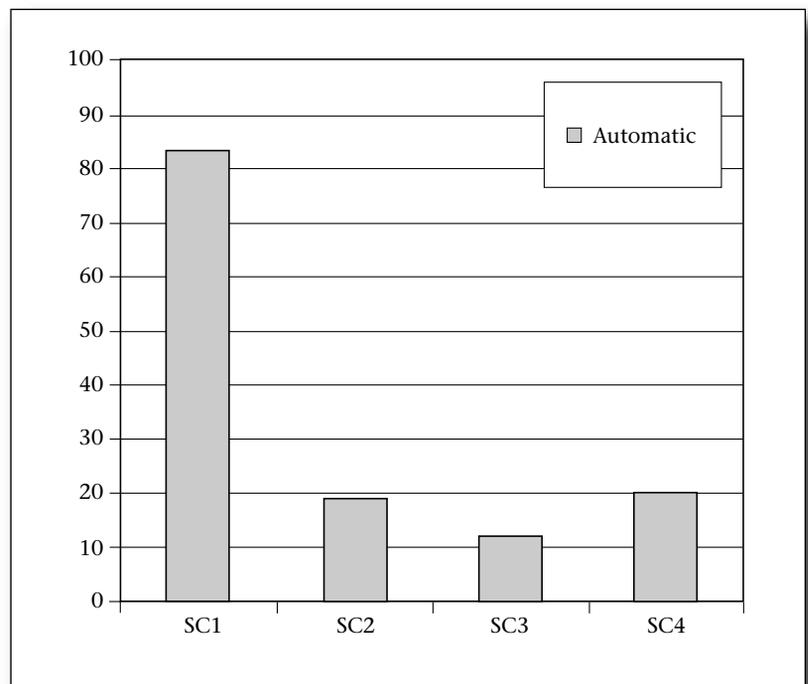


Figure 6. Recorded Average RTP Value for Dynamic Scripting.

Values represent 10 experiments for dynamic scripting with the automatically evolved knowledge bases against the student scripts. The x-axis lists the opponent strategies. The y-axis represents the average RTP value.

Zwijnaarde, Belgium: European Multidisciplinary Society for Modelling and Simulation Technology.

Cohen, P. (1996). Empirical Methods for Artificial Intelligence. *IEEE Expert: Intelligent Systems and Their Applications* 11(6): 88.

Demasi, P., and Cruz, A. (2002). Online Coevolution for Action Games. Paper presented at the 3rd International Conference on Intelligent Games and Simulation (GAME-ON 2002), 113, 120, November 29–30.

Goldberg, D. 1989. *Genetic Algorithms in Search, Optimization and Machine Learning*. Reading, MA: Addison-Wesley.

Guestrin, C.; Koller, D.; Gearhart, C.; and Kanodia, N. 2003. Generalizing Plans to New Environments in Relational MDPs. In *Proceedings of Eighteenth International Joint Conference on Artificial Intelligence (IJCAI-03)*, 1003–1010. San Francisco: Morgan Kaufmann Publishers.

Kaelbling, L.; Littman, M.; and Moore, A. 1996. Reinforcement Learning: A Survey. *Journal of Artificial Intelligence Research* 4: 237–285.

Laird, J., and van Lent, M. 2000. Human-Level AI's Killer Application: Interactive Computer Games. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, 1171–1178. Menlo Park, CA: AAAI Press.

Marthi, B.; Russell, S.; and Latham, D. 2005. Writing Stratagus-Playing Agents in Concurrent ALisp. Paper presented at the Workshop on Reasoning, Representation and Learning in Computer Games, IJCAI-05, Edinburgh, Scotland, 31 July.

Nareyek, A. 2004. AI in Computer Games. *Queue* 1(10), 58–65.

Ponsen, M., and Spronck, P. (2004). Improving Adaptive Game AI with Evolutionary Learning. In *Proceedings of Computer Games: Artificial Intelligence, Design and Education (CGAIDE-04)*, 389–396. Manhasset, NY: CMP Media.

Rabin, S. 2004. *AI Game Programming Wisdom 2*. Hingham, MA: Charles River Media.

Schaeffer, J. 2001. A Gamut of Games. *AI Magazine*, 22 (3): 29–46.

Spronck, P.; Sprinkhuizen-Kuyper, I.; and Postma, E. 2004. Online Adaptation of Game Opponent AI with Dynamic Scripting. *International Journal of Intelligent Games and Simulation* 3(1): 45–53.

Sutton, R., and Barto, A. (1998). *Reinforcement Learning: An Introduction*. Cambridge, MA: The MIT Press.



Marc Ponsen is a computer science Ph.D. candidate at the Institute of Knowledge and Agent Technology (IKAT) of Maastricht University. Prior to joining Maastricht University, he worked as an artificial intelligence researcher at Lehigh University. His research interests include machine learning,

reinforcement learning, and multiagent systems. His current research focuses on scaling reinforcement learning algorithms to complex environments, such

as computer games. He coauthored several refereed conference, workshop, and international journal papers on this subject.



Héctor Muñoz-Avila is an assistant professor at the Department of Computer Science and Engineering at Lehigh University. Prior to joining Lehigh, Muñoz-Avila worked as a researcher at the Naval Research Laboratory and the University of Maryland at College Park. He received his Ph.D. from the University of Kaiserslautern (Germany). Muñoz-Avila has done extensive research on case-based reasoning, planning, and machine learning, having written more than 10 journal papers and more than 30 refereed conference or workshop papers on the subject. Two of these papers received awards. He is also interested in advancing game AI with AI techniques. He has been chair, program committee member, and a reviewer for various international scientific meetings. He was program cochair of the Sixth International Conference on Case-Based Reasoning (ICCB-05) that was held in Chicago, IL (USA).



Pieter Spronck is a researcher of artificial intelligence at the Institute of Knowledge and Agent Technology (IKAT) of Maastricht University, The Netherlands. He received his Ph.D. from Maastricht University with a dissertation discussing adaptive game AI. He has coauthored more than 40 articles on AI research in international journals and refereed conference proceedings, about half of which are on AI in games. His research interests include evolutionary systems, adaptive control, computer game AI, and multiagent systems.



David W. Aha (Ph.D., University of California, Irvine, 1990) leads the Intelligent Decision Aids Group at the U.S. Naval Research Laboratory. His group researches, develops, and modifies state-of-the-art decision-aiding tools. Recent example projects concern a testbed (named TIELT) for evaluating AI learning techniques in simulators, knowledge extraction from text documents, and a web service broker for integrating meteorological data. His research interests include case-based reasoning (with particular emphasis on mixed-initiative, conversational approaches), machine learning, planning, knowledge extraction from text, and intelligent lessons learned systems. He has organized 15 international meetings on these topics, served on the editorial boards for three AI journals, assisted on eight dissertation committees, and was recently elected to the AAAI executive council.