Automatically Acquiring Domain Knowledge For Adaptive Game AI Using Evolutionary Learning

Marc J.V. Ponsen[1], Héctor Muñoz-Avila[1], Pieter Spronck[2], David W. Aha[3]

[1]Dept. of Computer Science & Engineering; Lehigh University; Bethlehem, PA; USA; {mjp304,hem4}@lehigh.edu
[2]IKAT; Maastricht University; Maastricht; POB 616; NL-6200 MD; The Netherlands; p.spronck@cs.unimaas.nl
[3]Navy Center for Applied Research in AI; Naval Research Laboratory (Code 5515); Washington, DC; USA; aha@aic.nrl.navy.mil

Abstract

Game AI is the decision-making process of computer-controlled opponents in computer games. Adaptive game AI can improve the entertainment value of computer games. It allows computer-controlled opponents to automatically fix weaknesses in the game AI and respond to changes in human-player tactics. *Dynamic scripting* is a recently developed approach for adaptive game AI that learns which tactics (i.e., action sequences) an opponent should select to play effectively against the human player. In previous work, these tactics were manually generated. We introduce AKADS; it uses an evolutionary algorithm to automatically generate such tactics. Our experiments show that it improves dynamic scripting's performance on a real-time strategy (RTS) game. Therefore, we conclude that high-quality domain knowledge (i.e., tactics) can be automatically generated for strong adaptive AI opponents in RTS games. This reduces the time and effort required by game developers to create intelligent game AI, thus freeing them to focus on other important topics (e.g., storytelling, graphics).

1. Introduction

Today's gaming environments are becoming increasingly realistic, especially in terms of the graphical presentation of the virtual world. To further increase realism, characters 'living' inside these virtual worlds must be able to reason effectively (Laird & van Lent 2000). Both game industry practitioners (Rabin 2004) and academics (Laird & van Lent 2000) predicted an increasing importance of artificial intelligence (AI) in computer games. This *game AI* is the decision-making process of computer controlled opponents. High-quality game AI will increase the game playing challenge (Nayerek 2004) and is a potential selling point for a game. However, the development time for game AI is typically short; most game companies assign graphics and storytelling the highest priorities (for marketing reasons) and do not implement the game AI until the end of the development process (Nayerek 2004), which complicates designing and testing strong game AI. Thus, even in state-of-the-art games, game AI is generally of inferior quality (Schaeffer 2001).

Adaptive game AI, which concerns methods for adapting the behavior of computer-controlled opponents, can potentially increase the quality of game AI. *Dynamic scripting* is a recently developed technique for implementing adaptive AI (Spronck *et al*. 2004). We use dynamic scripting to learn a state-based decision policy for the complex real-time strategy (RTS) game WARGUS.

Dynamic scripting uses extensive domain knowledge, namely one knowledge base (a set of action sequences, or *tactics*) per state. Manually designing these knowledge bases can take a long time, which game developers generally don't have, and risks analysis and encoding errors. We introduce a novel methodology, implemented in *AKADS* (Automatic Knowledge Acquisition for Dynamic Scripting), that uses an evolutionary algorithm to automatically generate tactics used by dynamic scripting. Our empirical results show that AKADS can successfully adapt to several static opponent strategies.

We describe related work in the next section. We then introduce RTS games and the game environment selected for the experiments. Next, we discuss our RTS implementation for dynamic scripting, and AKADS' method for automatically generating strong adaptive AI opponents in a RTS game. Finally, we describe our experimental results, and conclude with future work.

2. Related Work

AI researchers have shown that successful adaptive game AI is feasible (Demasi and Cruz 2002, Spronck *et al.* 2004). Successful adaptive game AI is invariably based on access to and use of the game's domain knowledge.

Ponsen and Spronck (2004) used offline evolutionary learning to generate high-performing opponents, and *manually* extracted tactics from them to improve the performance of adaptive game AI. In this paper, we describe an algorithm that *automatically* generates this high-quality domain knowledge.

Few attempts have been made to automatically create game AI. For example, Madeira et al. (2004) used reinforcement learning to generate strong AI opponents. However, their approach produces a static opponent. In contrast, dynamic scripting adapts to a player's behavior.

To the best of our knowledge, most empirical studies with RTS games address only a subset of the game AI. For example, Madeira et al.'s (2004) learning mechanism focuses on high-level, strategic decisions and lets the default AI execute these orders. Guestrin et al. (2003) instead focus on low-level decision-making, controlling

only a small number of soldiers. Unlike these experiments, we do not constrain the number of units involved, and focus on both strategic decisions (high-level decisions that determine the general behavior of a population as a whole) and tactical decisions (intermediate-level decisions that determine the behavior of units in local situations).

3. Real-Time Strategy Games

RTS is a strategy game genre that usually focuses on military combat. RTS games such as WARCRAFT™ and COMMAND & CONQUER™ require players to control armies (consisting of different unit types) and defeat all opposing forces in real-time. In most RTS games, winning requires efficiently collecting and managing resources, and appropriately distributing them over the various game elements. Typical RTS game actions include constructing buildings, researching new technologies, and combat.

Both human and computer players can use these game actions to form their strategy and tactics. Typically, RTS games require competing against multiple game AI opponents encoded as *scripts*, which are lists of game actions that are executed sequentially (Tozour 2002). A complete script represents a strategy, while a sub-list of game actions (i.e., one or more atomic game actions) in a script represents a tactic (e.g., constructing a blacksmith and acquiring all related technologies for that building).

For our experiments, we selected the RTS game WARGUS, whose underlying engine is STRATAGUS – an open-source engine for building RTS games. WARGUS (Figure 1) is a clone of the popular game WARCRAFT II™.

4. Dynamic Scripting in WARGUS

Spronck et al. (2004) introduced a novel technique, called *dynamic scripting*, that can generate AI opponent scripts which can adapt to a player's behavior. Dynamic scripting in WARGUS generates scripts on the fly by selecting state-specific tactics from a knowledge base, where these tactics were created using domain-specific knowledge. Each tactic in a state-specific knowledge base is assigned a weight, and the probability of selecting a tactic for a script is an increasing function of its weight value. After each opponent encounter, the weights of tactics employed during gameplay are increased when their contribution yields positive outcomes, and are decreased otherwise. The size of the weight change is determined by a weight-update function. To keep the sum of all weight values in a knowledge base constant, weight changes are executed through a redistribution of all weights in the knowledge base. Through punishments and rewards, dynamic scripting gradually adapts to the player's behavior. Spronck et al. (2004) detail dynamic scripting, which is an effective, robust, and efficient method for generating adaptive game AI that has been used for computer role playing (Spronck et al. 2004) and RTS games (Ponsen & Spronck 2004).

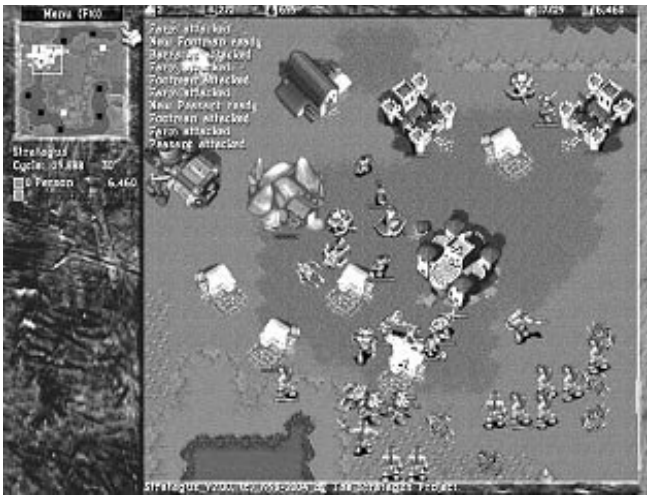

Figure 1: Screenshot of a battle in WARGUS.

4.1 Game States and their Knowledge Bases

Typically, RTS game players start with few game actions available to them. As players progress up the technology ladder, they acquire a larger arsenal of weapons, units, and buildings. The tactics that can be used in RTS games mainly depend on the availability of different unit types and technologies. When applying dynamic scripting to RTS games, we must constrain the adaptive AI's tactics selection process. Therefore, we divided the game into a small number of game states. Each state is paired with a unique knowledge base of tactics, and dynamic scripting can select these tactics when the game is in that state. Dynamic scripting starts by selecting tactics for the first state. When the execution of a selected tactic spawns a state change, a tactic is then selected for the new state.

We distinguish WARGUS game states according to the types of available buildings, which in turn determine the unit types that can be built and the technologies that can be researched. Consequently, state changes are spawned by executing tactics that create new buildings. For example, a player in the first state has a town hall and a barracks. The next building choices are a lumber mill, a blacksmith, and a keep. Building these cause transition from state 1 to states 2, 3, and 5, respectively. For a detailed description of the states see (Ponsen & Spronck 2004).

4.2 Weight Adaptation

Weight updates in WARGUS are based on both an evaluation of the performance of the adaptive AI during the entire game (*overall fitness*), and between state changes (*state fitness*). Using both evaluations for weight updating increases learning efficiency (Manslow 2004).

The overall fitness function F for dynamic player d controlled by dynamic scripting yields a value in [0,1]. It is defined as:

$$F = \begin{cases} \min\left(\frac{S_d}{S_d + S_o}, b\right) & \{d \quad lost\} \\ \max\left(b, \frac{S_d}{S_d + S_o}\right) & \{d \quad won\} \end{cases} \qquad (1)$$

In Equation 1, S_d represents the score for d, S_o represents the score for d's opponent, and $b \in [0,1]$ is the break-even point, at which the weights remain unchanged. For the dynamic player, the state fitness F_i for state i is defined as:

$$F_i = \begin{cases} \dfrac{S_{d,i}}{S_{d,i} + S_{o,i}} & \{i = 1\} \\ \dfrac{S_{d,i}}{S_{d,i} + S_{o,i}} - \dfrac{S_{d,i-1}}{S_{d,i-1} + S_{o,i-1}} & \{i > 1\} \end{cases} \quad (2)$$

In Equation 2, $S_{d,i}$ represents d's score after state i, and $S_{o,i}$ represents the score of d's opponent after state i.

This scoring function is domain-dependent, and should reflect the relative strength of the two opposing players. For WARGUS, we defined the score S_x for player x as:

$$S_x = 0.7M_x + 0.3B_x \quad (3)$$

In Equation 3, M_x represents the military points for player x (i.e., the number of points awarded for killing units and destroying buildings), and B_x represents the building points for player x (i.e., the number of points awarded for training armies and constructing buildings).

After each game, the weights of all the selected tactics are updated. The weight-update function translates the fitness functions into weight adaptations. The weight-update function W for dynamic player d is defined as:

$$W = \begin{cases} \max\left(W_{\min}, W_{org} - 0.3\dfrac{b-F}{b}P - 0.7\dfrac{b-F_i}{b}P\right) & \{F < b\} \\ \min\left(W_{org} + 0.3\dfrac{F-b}{1-b}R + 0.7\dfrac{F_i-b}{1-b}R, W_{\max}\right) & \{F \geq b\} \end{cases} \quad (4)$$

In Equation 4, W is the tactic's new weight value, W_{org} is its value before this update, P is the maximum penalty, R is the maximum reward, W_{max} is the maximum weight value, W_{min} is the minimum weight value, F is d's overall fitness, F_i is the fitness for d in state i, and b is the break-even point. This equation prioritizes state performance over overall performance because, even if a game is lost, we wish to prevent tactics from being punished too much in states where performance is successful.

5. Automatically Generating AI Opponents

AKADS' three steps for automatically generating adaptive AI opponents are listed in Table 1. This process is schematically illustrated in Figure 2.

Table 1: The three steps in AKADS

Name	Learning task
EA	Evolving domain knowledge (using an Evolutionary Algorithm)
KT	Within-domain Knowledge Transfer (of evolved strategies to tactics for the knowledge bases)
DS	State-based tactic selection (via Dynamic Scripting)

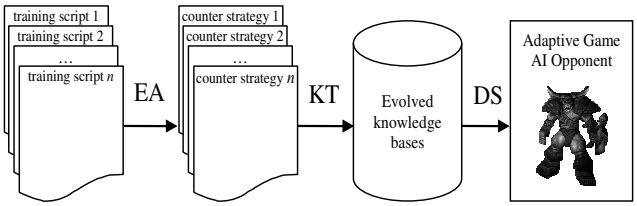

Figure 2: Schematic representation of AKADS's process

The first step in AKADS uses an evolutionary algorithm (EA) to search for strategies that defeat specific opponent strategies. These strategies are provided to EA as a training set, the only manual input AKADS requires. This training set contains (manually designed) *static scripts* and (automatically generated) *evolutionary scripts*. Static scripts are default scripted opponents that are typically provided with alpha versions of a game to record the strategies employed by human players during testing. In contrast, an evolutionary script is a previously evolved strategy that we will use as an opponent strategy to evolve new strategies. Static scripts are usually of high quality because they are recorded from human player strategies. In contrast, evolutionary scripts can be generated completely automatically. Our training set includes the four default-scripted opponents provided with STRATAGUS, and 36 evolutionary scripts. The output of EA is a set of *counter-strategies*, which are static strategies that can defeat the scripts in the training set.

The second step (KT) transfers the domain knowledge stored in the evolved strategies to the knowledge bases used by the adaptive AI algorithm (i.e., dynamic scripting).

The last step in AKADS (DS) empirically tests the performance of the adaptive AI. We will detail the first two steps in the following sections. In Section 6, we then report our evaluation of this methodology for automatically generating adaptive AI opponents.

5.1 EA: Evolving Domain Knowledge

Chromosome Encoding

EA works with a population of chromosomes, each of which represents a static strategy. To encode a strategy for WARGUS, each gene in the chromosome represents a game action. Four different gene types exist, corresponding to the available game 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.

Figure 3 shows the chromosome's design. The chromosome is divided into states (as described in section 4.1). Figure 3 shows that states include a state marker followed by the state number and a series of genes, whose representation is also shown in Figure 3. A partial example of a chromosome is shown at the bottom of Figure 3 (i.e., State 1 includes a combat gene and a build gene). Chromosomes for the initial population are generated randomly. By taking into account state changes spawned by executing build genes, only legal game AI is created.

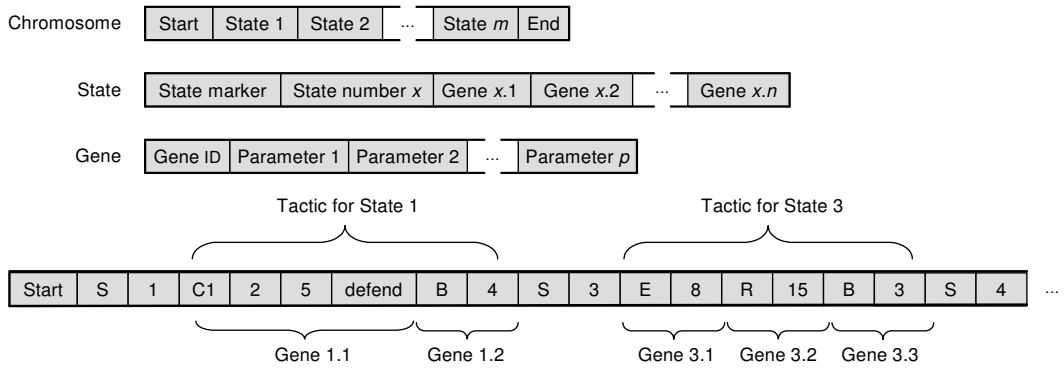

Figure 3: Design of a chromosome to store game AI for WARGUS.

Ponsen and Spronck (2004) provide a more detailed description of the chromosome encoding.

Fitness Function

To measure the success of a chromosome, we used the following fitness function F for the dynamic player d (controlled by an evolved strategy), which yields a value in the range [0,1]:

$$F = \begin{cases} \min\left(\frac{GC}{EC} \cdot \frac{M_d}{M_d + M_o}, b\right) & \{d \quad lost\} \\ \max\left(b, \frac{M_d}{M_d + M_o}\right) & \{d \quad won\} \end{cases} \qquad (5)$$

n Equation 5, M_d represents the military points for d, M_o the military points for d's opponent, b the break-even point, GC the game cycle (i.e., time it took before the game is lost by one of the players) and EC the end cycle (i.e., longest time a game is allowed to continue). When a game reaches the end cycle and neither army has been completely defeated, scores are measured and the game is aborted. The factor GC/EC ensures losing chromosomes that play a long game receive higher fitness scores than losing chromosomes that play a short game.

Genetic Operators

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% probability, (3) Gene Biased Mutation, which selects one parent and mutates parameters for existing economy or combat genes with a 50% probability, and (4) Randomization, which generates a random new chromosome. Randomization has a 10% chance of being selected during an evolution. The other genetic operators have a 30% chance. By design, all four ensure that a child chromosome always represents a legal game AI.

The four genetic operators take into account "activated" genes, which represent game actions that were executed when fitness was assessed. Non-activated 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, the child is rejected and a new child is generated. Ponsen and Spronck (2004) describe more detail for these genetic operators.

5.2 KT: Within-Domain Knowledge Transfer

AKADS 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 depends on the buildings that the player possesses. Thus, we distinguish tactics using the WARGUS game states described in Section 4.1.

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 3 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). 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 (i.e., constructing a blacksmith causes a transition to state 3, as indicated by the state marker in the example chromosome).

6. Experimental Evaluation

6.1 Crafting the Evolved Knowledge Bases

The evolutionary algorithm uses a population of size 50. Relatively successful chromosomes (as determined by the fitness function) are allowed to breed. To select parent chromosomes for breeding, we used size-3 tournament selection (Buckland 2004). This method prevents early convergence and is computationally fast. Newly generated chromosomes replace existing chromosomes in the population, using size-3 crowding (Goldberg 1989). 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.

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.

We evolved 40 chromosomes against the strategies provided in the training set. The EA was able to find a strong counter-strategy against each strategy in the training set. All chromosomes had a fitness score higher than 0.7, which represents a clear victory.

Using the KT algorithm that described in Section 5.2, the 40 evolved chromosomes produced 164 tactics that were added to the evolved knowledge bases for their corresponding state. The EA generally evolved solutions that end after a few states and did not find any tactics for some of the later game states. All games in the offline process ended before the evolutionary AI 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 ignore other states.

6.2 Performance of Dynamic Scripting

We evaluated the performance of dynamic scripting using the evolved knowledge bases in WARGUS by letting the computer play the game against itself. One of the two opposing players was controlled by dynamic scripting (the *dynamic player*), while the other was controlled by a static script (the *static player*). Each game lasted until one of the players was defeated, or until a certain period of time had elapsed. If the game ended due to the time restriction, the player with the highest score was considered to have won. After each game, the dynamic player's knowledge bases were adapted, and the next game was started using them. A sequence of 100 games constituted one test. We tested eight different strategies for the static player:

1-2. Small/Large Balanced Land Attack (SBLA/LBLA). These focus on land combat, maintaining a balance between offensive actions, defensive actions, and research. SBLA is applied on a small map (64x64 cells) and LBLA is applied on a large map (128x128 cells).

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

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

5-8. Student Scripts (SC): Scripts 1-4 were used as input for AKADS. We also tested dynamic scripting against four static scripts that were not part of the training set for AKADS. We asked students to independently create scripts that could defeat scripts 1-4. We then played the student scripts against one another. The top four competitors in the tournament were used for testing against dynamic scripting on a small map.

To quantify the relative performance of the dynamic player against the static player, we used the randomization turning point (RTP), which is measured as follows. After each game, a randomization test (Cohen 1995; pp. 168–170) was performed using the overall fitness values over the last ten games, with the null hypothesis that both players are equally strong. The dynamic player was said to outperform the static player if the randomization test concluded that the null hypothesis can be rejected with 90% probability in favor of the dynamic player. RTP is the number of the first game in which the dynamic player outperforms the static player. A low RTP value indicates good efficiency for dynamic scripting.

Table 2: Evaluation results of dynamic scripting in WARGUS RTS games using the improved and evolved knowledge bases

	Improved Knowledge Bases				**Evolved Knowledge Bases**			
Strategy	Tests	RTP	>100	Won	Tests	RTP	>100	Won
SBLA	11	19	0	72	11	10	0	85
LBLA	11	24	0	66	11	11	0	76
SR	10	-	10	27	21	51	0	29
KR	10	-	10	10	10	-	10	13
SC1	-	-	-	-	10	83	5	27
SC2	-	-	-	-	10	19	0	61
SC3	-	-	-	-	10	12	0	84
SC4	-	-	-	-	10	20	0	73

Ponsen and Spronck (2004) manually improved existing knowledge bases from counter-strategies that were evolved offline, and tested dynamic scripting against SBLA, LBLA, SR, and KR. The results for dynamic scripting using these knowledge bases in WARGUS are shown in the left half of Table 2. From left to right, this table displays (1) the strategy used by the static player, (2) the number of tests, (3) the average RTP, (4) the number of tests that did not find an RTP within 100 games, and (5) the average number of games won out of 100.

For our new experiments with dynamic scripting using the automatically evolved knowledge bases, we set P to 400, R to 400, W_{max} to 4000, W_{min} to 25, and b to 0.5. The columns on the right half of Table 2 show the results for dynamic scripting using these knowledge bases. As shown, performance improved against all previously tested scripts; RTP values against all scripts except KR have substantially decreased and on average more games are won. Dynamic scripting with the evolved knowledge bases outperforms both balanced scripts before any learning occurs (e.g., before weights are adapted). In previous tests against the SR, dynamic scripting was unable to find an RTP point. In contrast, dynamic scripting using the evolved knowledge bases recorded an average RTP of 51 against SR. The results against the student scripts are also encouraging. Only the champion script puts up a good fight; the others are already defeated from the start.

We believe that dynamic scripting's increased performance, compared to our earlier experiments (Ponsen & 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 atomic game actions, whereas the improved knowledge bases include tactics that consist of a single atomic game action. Knowledge bases consisting of compound tactics (i.e., an effective combination of fine-tuned game actions) reduce the search complexity in WARGUS allowing dynamic scripting to achieve fast adaptation against many static opponents.

7. Conclusions and Future Work

We set out to show that automatically generating strong adaptive AI opponents for RTS games is feasible. We first discussed the dynamic scripting technique and its application to WARGUS, a clone of the popular WARCRAFT II™ game. We explained that domain knowledge is a crucial factor for dynamic scripting's performance.

We proposed a methodology (implemented in AKADS) that can automatically generate high-quality domain knowledge for use by dynamic scripting. We tested AKADS against eight different static scripts. From our empirical results we concluded that the evolved knowledge bases improved the performance of dynamic scripting against the four static opponents that were used in previous experiments (Ponsen & Spronck 2004). We also tested dynamic scripting against four new strong scripts. The results were very encouraging (i.e., dynamic scripting almost always found an RTP), which shows that dynamic scripting can adapt to many different static strategies.

We therefore draw the following conclusion from our experiments: *It is possible to automatically generate high-quality domain knowledge that can be used to generate strong adaptive AI opponents in RTS games.* AKADS produces strong adaptive AI opponents with relatively little effort on the part of game developers, hence freeing time for them to focus on graphics and storytelling.

For future work we plan to investigate whether strategy recognition can further enhance performance. It seems the EA successfully located fast solutions (e.g., chromosomes that quickly defeated the static scripts). Consequently, mostly strong tactics for *early* states were added to the knowledge bases. The evolved knowledge bases may be overfitted with early rush tactics. Strategy recognition may solve this problem by preventing many similar tactics (e.g., for combating the rush strategies) from being repeatedly added to the knowledge bases. We will also investigate on-line applications of dynamic scripting, which require states to occur repeatedly within a game. Finally, we'll investigate the use of more expressive state representations for dynamic scripting. Currently, states only identify which buildings have been created. Aha et al. (2005) extend this state description with eight additional features to guide a case-based tactics selection algorithm for playing WARGUS, and similar approaches may improve dynamic scripting's performance.

Acknowledgements

This research was sponsored by a grant from DARPA and the Naval Research Laboratory.

References

Aha, D.W., Molineaux, M., & Ponsen, M. (2005). Learning to win: Case-based plan selection in a real-time strategy game. To appear in *Proceedings of the Sixth International Conference on Case-Based Reasoning.* Chicago, IL: Springer.

Buckland, M. 2004. "Building better Genetic Algorithms." *AI Game Programming Wisdom 2*, Charles River Media, Hingham, MA, pp. 649–660.

Cohen, R.C. (1995). *Empirical Methods for Artificial Intelligence*, MIT Press, Cambridge, MA.

Demasi, P. and A.J. de O. Cruz. 2002. "Online Coevolution for Action Games." GAME-ON 2002 *3rd International Conference on Intelligent Games and Simulation*, SCS Europe Bvba, pp. 113–120.

Goldberg, D.E. 1989. *Genetic Algorithms in Search, Optimization & Machine Learning*, Addison-Wesley, Reading, UK.

Guestrin, C., Koller, D., Gearhart C., and Kanodia N. (2003) "Generalizing Plans to New Environments in Relational MDPs". *International Joint Conference on Artificial Intelligence (IJCAI 2003)*, Acapulco, Mexico.

Laird, J.E. and M. van Lent. 2000. "Human-Level AI's Killer Application: Computer Game AI." *Proceedings of AAAI 2000 Fall Symposium on Simulating Human Agents*, AAAI Press 2000, pp. 80–87.

Madeira, C., Corruble, V. Ramalho, G. Ratich B. 2004. "Bootstrapping the Learning Process for the Semi-Automated Design of Challenging Game AI", *Proceedings of the AAAI-04 workshop on AI in games*, San Jose 2004

Manslow, J. 2004. "Using reinforcement learning to Solve AI Control Problems". *AI Game Programming Wisdom 2*, Charles River Media, Hingham, MA, pp. 591–601.

Nareyek, A. 2004. "AI in Computer Games". ACM Queue. 1(10). pp. 58-65

Ponsen, M. and Spronck, P. (2004). "Improving Adaptive Game AI with Evolutionary Learning." *Computer Games: Artificial Intelligence, Design and Education* (CGAIDE 2004). pp. 389-396. University of Wolverhampton.

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

Schaeffer, J. 2001. "A Gamut of Games." *AI Magazine,* Vol. 22, No. 3, pp. 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*, Vol. 3, No. 1, University of Wolverhampton and EUROSIS, pp. 45–53.

Tozour, P. 2002. "The Perils of AI Scripting." *AI Game Programming Wisdom*, Charles River Media, Hingham, MA, pp. 541–547.