

# IMPROVING ADAPTIVE GAME AI WITH EVOLUTIONARY LEARNING

Marc Ponsen  
Lehigh University / Computer Science & Engineering  
19 Memorial Drive West  
Bethlehem, PA 18015-3084 USA  
mjp304@lehigh.edu

Pieter Spronck  
Maastricht University / IKAT  
P.O. Box 616, NL-6200 MD Maastricht,  
The Netherlands  
p.spronck@cs.unimaas.nl

## KEYWORDS

Games, artificial intelligence, real-time strategy, dynamic scripting, evolutionary algorithms.

## ABSTRACT

Game AI is defined as the decision-making process of computer-controlled opponents in computer games. Adaptive game AI can improve the entertainment provided by computer games, by allowing the computer-controlled opponents to fix automatically weaknesses in the game AI, and to respond to changes in human-player tactics online, i.e., during gameplay. Successful adaptive game AI is based invariably on domain knowledge of the game it is used in. Dynamic scripting is an algorithm that implements adaptive game AI. The domain knowledge used by dynamic scripting is stored in a rulebase with manually designed rules. In this paper we propose the use of an offline evolutionary algorithm to enhance the performance of adaptive game AI, by evolving new domain knowledge. We empirically validate our proposal, using dynamic scripting as adaptive game AI in a real-time-strategy (RTS) game, in three steps: (1) we implement and test dynamic scripting in an RTS game; (2) we use an offline evolutionary algorithm to evolve new tactics that are able to deal with optimised tactics, which dynamic scripting cannot defeat using its original rulebase; (3) we translate the evolved tactics to rules in the rulebase, and test dynamic scripting with the revised rulebase. The empirical validation shows that the revised rulebase yields a significantly improved performance of dynamic scripting compared to the original rulebase. We therefore conclude that offline evolutionary learning can be used to improve the performance of adaptive game AI.

## INTRODUCTION

Traditionally, commercial game developers spend most of their resources on improving a game's graphics. However, in recent years, game developers have begun to compete with each other by providing a more challenging gaming experience (Rabin 2004). For most games, challenging gameplay is equivalent to having high-quality game AI (Laird 2000). Game AI is defined as the decision-making process of computer-controlled opponents. Even in state-of-the-art games, game AI is, in general, of inferior quality (Schaeffer 2001, Laird 2001, Gold 2004). It tends to be predictable, and often contains weaknesses that human players can exploit.

Adaptive game AI, which implies the online (i.e., during gameplay) adaptation of the behaviour of computer-controlled opponents, has the potential to increase the quality of game AI. It has been widely disregarded by game developers, because online learning tends to be slow, and can lead to undesired behaviour (Manslow 2002). However, academic game AI researchers have shown that successful adaptive game AI is feasible (Demasi and Cruz 2002,

Johnson 2004, Spronck, Sprinkhuizen-Kuyper and Postma 2004a).

To ensure the efficiency and reliability of adaptive game AI, it must incorporate a great amount of prior domain knowledge (Manslow 2002, Spronck, Sprinkhuizen-Kuyper and Postma 2004b). However, if the incorporated domain knowledge is incorrect or insufficient, adaptive game AI will not be able to generate satisfying results. In this paper we propose an evolutionary algorithm to improve the quality of the domain knowledge used by adaptive game AI. We empirically validate our proposal by testing it on an adaptive game AI technique called "dynamic scripting", used in a real-time strategy (RTS) game.

The outline of the remainder of the paper is as follows. First, we discuss RTS games, and the game environment selected for the experiments. Then, we discuss the implementation of dynamic scripting for RTS games, followed by a discussion of the implementation of an evolutionary algorithm that generates successful tactics for RTS games. The achieved results are used to show how the tactics discovered with an evolutionary algorithm can be employed to improve the original dynamic scripting implementation. Finally, we draw conclusions and indicate future work.

## REAL-TIME-STRATEGY GAMES

RTS games are simple military simulations (war games) that require the player to control armies (consisting of different types of units), and defeat all opposing forces. In most RTS games, the key to winning lies in efficiently collecting and managing resources, and appropriately distributing these resources over the various game elements. Typical game elements in RTS games include the construction of buildings, the research of new technologies, and combat.

Game AI in RTS games determines the tactics of the armies controlled by the computer, including the management of resources. Game AI in RTS games is particularly challenging for game developers, because of two reasons: (1) RTS games are complex, i.e., a wide variety of tactics can be employed, and (2) decisions have to be made in real-time, i.e., under severe time constraints. RTS games have been called "an ideal test-bed for real-time AI research" (Buro 2003).

For our experiments, we selected the RTS game WARGUS, with STRATAGUS as its underlying engine. STRATAGUS is an open-source engine for building RTS games. WARGUS implements a clone of the highly popular RTS game WARCRAFT II. While the graphics of WARGUS are not up-to-date with today's standards, its gameplay can still be considered state-of-the-art. Figure 1 illustrates WARGUS. The figure shows a battle between an army of "orcs", which

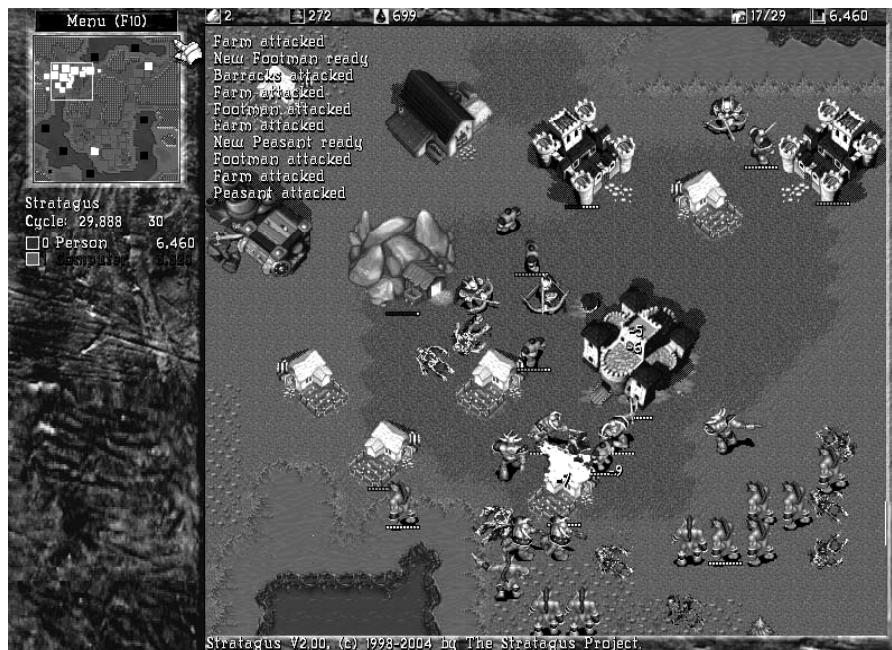


Figure 1: Screenshot of a battle in WARGUS.

approach from the bottom right, and an army of “humans”, which attempt to defend a base consisting of several buildings.

## ADAPTIVE GAME AI IN RTS GAMES

Game AI for complex games, such as RTS games, is mostly defined in scripts, i.e., lists of rules that are executed sequentially (Tozour 2002). Because the scripts tend to be long and complex (Brockington and Darrah 2002), they are likely to contain weaknesses, which the human player can exploit. Because scripts are static they cannot adapt to overcome these exploits. Spronck *et al.* (2004a) designed a novel technique called “dynamic scripting” that realises the online adaptation of scripted opponent AI. Experiments have shown that the dynamic scripting technique can be successfully incorporated in commercial Computer RolePlaying Games (CRPGs) (Spronck *et al.* 2004a, 2004b).

Because the game AI for WARGUS is defined in scripts, dynamic scripting should also be applicable to WARGUS. However, because of the differences between RTS games and CRPGs, the original dynamic scripting implementation cannot be transferred to RTS games unchanged. In this section a dynamic scripting implementation for the game AI in RTS games is designed and evaluated. The basics of dynamic scripting are explained first. Then, we highlight the changes made to dynamic scripting to apply it to RTS games, and discuss the implementation of dynamic scripting in WARGUS. The implementation is evaluated, and the evaluation results are discussed.

### Dynamic Scripting

Dynamic scripting is an online learning technique for commercial computer games, inspired by reinforcement learning (Russel and Norvig 1995). Dynamic scripting generates scripted opponents on the fly by extracting rules from an adaptive rulebase. The rules in the rulebase are manually designed using domain-specific knowledge. The

probability that a rule is selected for a script is proportional to a weight value that is associated with each rule, i.e., rules with larger weights have a higher probability of being selected. After every encounter between opponents, the weights of rules employed during gameplay are increased when having a positive contribution to the outcome, and decreased when having a negative contribution. The size of the weight changes is determined by a weight-update function. To keep the sum of all weight values in a rulebase constant, weight changes are executed through a redistribution of all weights in the rulebase. Through the process of punishments and rewards, dynamic scripting gradually adapts to the human player. For CRPGs, it has been shown that dynamic scripting is fast, effective, robust and efficient (Spronck *et al.* 2004a).

### Dynamic Scripting for RTS games

Our design of dynamic scripting for RTS games has two differences with dynamic scripting for CRPGs. The first difference is that, while dynamic scripting for CRPGs employs different rulebases for different opponent types in the game (Spronck *et al.* 2004a), our RTS implementation of dynamic scripting employs different rulebases for the different states of the game. The reason for this deviation from the CRPG implementation of dynamic scripting is that, in contrast with CRPGs, the tactics that can be used in an RTS game mainly depend on the availability of different unit types and technologies. For instance, attacking with weak units might be the only viable choice in early game states, while in later game states, when strong units are available, usually weak units will have become useless.

The second difference is that, while dynamic scripting for CRPGs executes weight updates based on an evaluation of a fight, our RTS implementation of dynamic scripting executes weight updates based on both an evaluation of the performance of the game AI during the whole game (called the “overall fitness”), and on an evaluation of the performance of the game AI between state changes (called

the “state fitness”). As such, the weight-update function is based on the state fitness, combined with the overall fitness. The use of both evaluations for the weight-updates increases the efficiency of the learning mechanism (Manslow 2004).

### Dynamic Scripting in WARGUS

We implemented the dynamic scripting process in WARGUS as follows. Dynamic scripting starts by randomly selecting rules for the first state. When a rule is selected that spawns a state change, from that point on rules will be selected for the new state. To avoid monotone behaviour, we restricted each rule to be selected only once for each state. At the end of the scripts, a loop is implemented that initiates continuous attacks against the enemy.

Because in WARGUS the available buildings determine the unit types that can be built and technologies that can be researched, we decided to distinguish game states according to the type of buildings possessed. Consequently, state

changes are spawned by rules that comprise the creation of new buildings. The twenty states for WARGUS, and the possible state changes, are illustrated in Figure 2.

We allowed a maximum of 100 rules per script. The rulebases for each of the states contained between 21 and 42 rules. The rules can be divided in four basic categories: (1) build rules (for constructing buildings), (2) research rules (for acquiring new technologies), (3) economy rules (for gathering resources), and (4) combat rules (for military activities). To design the rules, we incorporated domain knowledge acquired from strategy guides for WARCRAFT II.

The ‘overall fitness’ function  $F$  for player  $d$  controlled by dynamic scripting (henceforth called the “dynamic player”) yields a value in the range  $[0,1]$ . It is defined as:

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

In equation (1),  $S_d$  represents the score for the dynamic player,  $S_o$  represents the score for the dynamic player’s opponent, and  $b \in [0,1]$  is the break-even point. At the break-even point, weights remain unchanged.

For the dynamic player, the state fitness  $F_i$  for state  $i$  is defined as:

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

In equation (2),  $S_{d,x}$  represents the score of the dynamic player after state  $x$ , and  $S_{o,x}$  represents the score of the dynamic player’s opponent after state  $x$ .

The score function is domain-dependent, and should successfully reflect the relative strength of the two opposing players in the game. 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 destruction of 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 rules employed are updated. The weight-update function translates the fitness functions into weight adaptations for the rules in the script. The weight-update function  $W$  for the dynamic player is defined as:

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

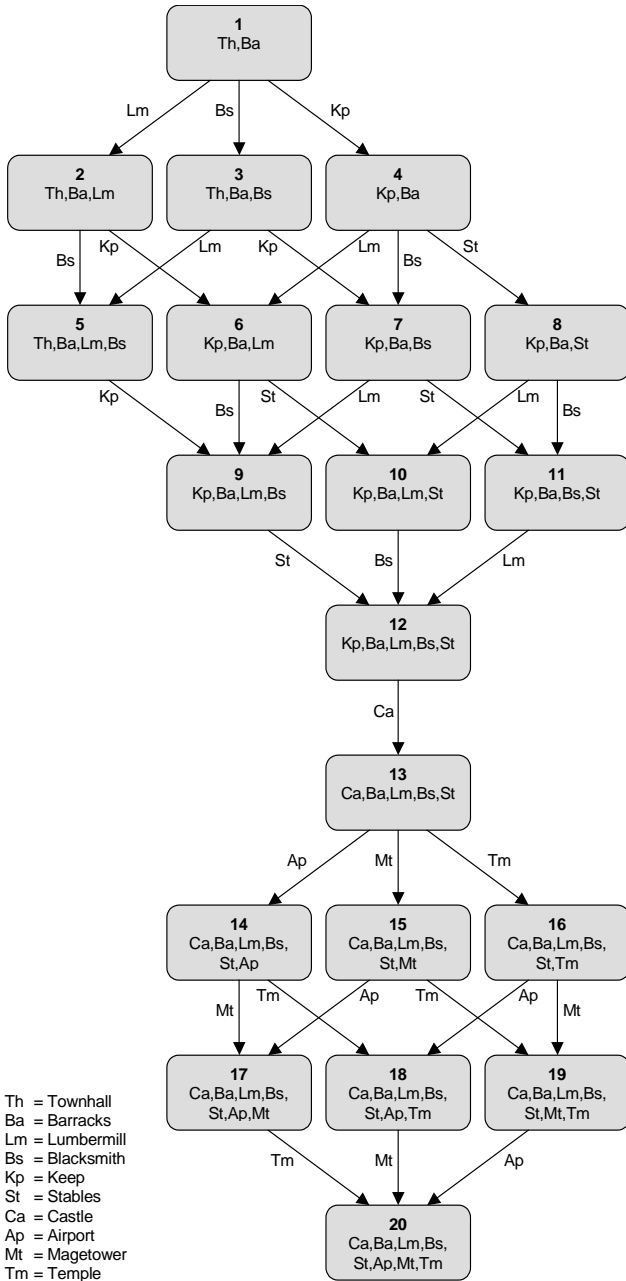


Figure 2: Game states in WARGUS.

In equation (4),  $W$  is the new weight value,  $W_{org}$  is the original weight value,  $P$  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 the overall fitness of the dynamic player,  $F_i$  is the state fitness for the dynamic player in state  $i$ , and  $b$  is the break-even point. The equation indicates that we prioritise state performance over overall performance. The reason is that, even if a game is lost, we wish to prevent rules from being punished (too much) in states where performance is successful. In our simulation we set  $P$  to 175,  $R$  to 200,  $W_{max}$  to 1250,  $W_{min}$  to 25 and  $b$  to 0.5.

### Evaluating Dynamic Scripting in WARGUS

We evaluated the performance of dynamic scripting for RTS games 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), and 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 (which was rarely the case), the player with the highest score was considered to have won. After the game, the rulebases were adapted, and the next game was started, using the adapted rulebases. A sequence of 100 games constituted one test. We tested four different tactics for the static player:

1. **Small Balanced Land Attack (SBLA):** The SBLA is a tactic that focuses on land combat, keeping a balance between offensive actions, defensive actions, and research. The SBLA is applied on a small map. Games on a small map are usually decided swiftly, with fierce battles between weak armies.
2. **Large Balanced Land Attack (LBLA):** The LBLA is similar to the SBLA, but applied on a large map. A large map allows for a slower-paced game, with long-lasting battles between strong armies.
3. **Soldier’s Rush (SR):** The soldier’s rush aims at overwhelming the opponent with cheap offensive units in an early state of the game. Since the soldier’s rush works best in fast games, we tested it on a small map.
4. **Knight’s Rush (KR):** The knight’s rush aims at quick technological advancement, launching large offences as soon as strong units are available. Since the knight’s rush works best in slower-paced games, we tested it on a large map.

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

If the player controlled by dynamic scripting is unable to statistically outperform the static player within 100 games,

the test is aborted. For the SBLA we ran 31 tests. For the LBLA we ran 21 tests. For both the SR and KR, we ran 10 tests.

### Results

The results of the evaluation of dynamic scripting in WARGUS are displayed in Table 1. From left to right, the table displays (1) the tactic used by the static player, (2) the number of tests, (3) the lowest RTP found, (4) the highest RTP found, (5) the average RTP, (6) the median RTP, (7) the number of tests that did not find an RTP within 100 games, and (8) the average number of games won out of 100.

Tactic	Tests	Low	High	Avg	Med	>100	Won
SBLA	31	18	99	50	39	0	59.3
LBLA	21	19	79	49	47	0	60.2
SR	10					10	1.2
KR	10					10	2.3

Table 1: Evaluation results of dynamic scripting in RTS games.

From the low values for the RTPs for both the SBLA and the LBLA, we can conclude that the dynamic player efficiently adapts to these two tactics. Therefore, we conclude that dynamic scripting in our implementation can be applied successfully to RTS games.

However, the dynamic player was unable to adapt to the soldier’s rush and the knight’s rush within 100 games. As the rightmost column in Table 1 shows, the dynamic player only won approximately 1 out of 100 games against the soldier’s rush, and 1 out of 50 games against the knight’s rush. The reason for the inferior performance of the dynamic player against the two rush tactics is twofold, namely (1) the rush tactics are optimised, in the sense that it is very hard to design game AI that is able to deal with them, and (2) the rulebase does not contain the appropriate knowledge to easily design game AI that is able to deal with the rush tactics.

The remainder of this paper investigates how offline evolutionary learning can be used to improve the rulebase to deal with optimised tactics.

## EVOLUTIONARY TACTICS

In this section we empirically investigate to what extent an evolutionary algorithm can be used to search for effective tactics for RTS games. Our goal is to use offline evolutionary learning to design tactics that can be used to defeat the two optimised tactics described in the previous section, the soldier’s rush and the knight’s rush. In the following subsections we describe the procedure used, the encoding of the chromosome, the fitness function, the genetic operators, and the achieved results.

### Experimental Procedure

We designed an evolutionary algorithm that evolves new tactics to be used in WARGUS against a static player using the soldier’s rush and the knight’s rush tactics. The evolutionary algorithm uses a population of size 50, representing sample solutions (i.e., game AI scripts). Relatively successful solutions (as determined by a fitness

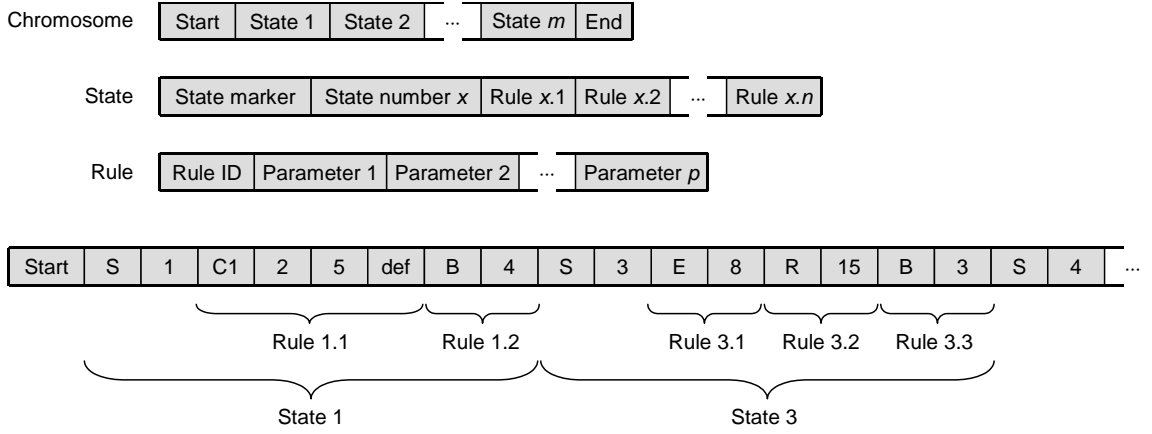


Figure 3: Design of a chromosome to store game AI for WARGUS.

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 solutions 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 0.75 against the soldier's rush, and to 0.7 against the knight's rush. Since there is no guarantee that a solution exceeding the target value will be found, the evolution process also ends after it has generated a maximum number of solutions. This is the run-stop criterion. We set the maximum number of solutions to 250. The choices for the fitness-stop and run-stop criteria were determined during preliminary experiments.

### Encoding

The evolutionary algorithm works with a population of chromosomes. In the present context, a chromosome represents a game-AI script. To encode a game-AI script for WARGUS, each gene in the chromosome represents one rule. Four different gene types are distinguished, corresponding to the four basic rule categories mentioned in the previous section, namely (1) build genes, (2) research genes, (3) economy genes, and (4) combat genes. Each gene consists of a rule ID that indicates the type of gene (B, R, E and C, respectively), followed by values for the parameters needed by the gene. Of the combat gene, there are actually twenty variations, one for each possible state, each with its own parameters. The genes are grouped by states. A separate marker (S), followed by the state number, indicates the start of a state.

The chromosome design is illustrated in Figure 3. A schematic representation of the chromosome, divided into states, is shown at the top. Below it, a schematic representation of one state in the chromosome is shown, consisting of a state marker and a series of rule genes. Rule genes are identified by the number of the state for which they occur, followed by a period, followed by a sequence number. Below the state representation, a schematic representation of one rule is shown. At the bottom, part of an example chromosome is shown. Chromosomes for the initial population are generated randomly.

By taking into account state changes spawned by build genes, it is ensured that only legal game AI scripts are created. A more detailed description of the chromosome design can be found in (Ponsen 2004).

### Fitness Function

To measure the success of a game AI script represented by a chromosome, the following fitness function  $F$  for the dynamic player  $d$ , yielding a value in the range [0,1], is defined:

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

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

### Genetic Operators

To breed new chromosomes, four genetic operators were implemented. By design, all four genetic operators ensure that a child chromosome always represents a legal game-AI script. Parent chromosomes are selected with a chance corresponding to their fitness values.

The genetic operators take into account "activated" genes. An activated gene is a gene that represents a rule that was executed during the fitness determination. Non-activated genes can be considered irrelevant to the game-AI script the chromosome represents. 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.

1. **State Crossover** selects two parents, and copies states from either parent to the child chromosome. State crossover is controlled by “matching states”. A matching state is a state that exists in both parent chromosomes. Figure 2 makes evident that, for WARGUS, there are always at least four matching states, namely state 1, state 12, state 13, and state 20. State crossover will only be used when there are least three matching states with activated genes. A child chromosome is created as follows. States are copied from the first parent chromosome to the child chromosome, starting at state 1 and working down the chromosome. When there is a state change to a matching state, there is a 50% probability that from that point on, the role of the two parents is switched, and states are copied from the second parent. When the next state change to a matching state is encountered, again a switch between the parents can occur. This continues until the last state has been copied.
2. **Rule Replace Mutation** selects one parent, and replaces economy, research or combat rules with a 25% probability. Building rules are excluded, both for and as replacement, because these could spawn a state change and thus could possibly corrupt the chromosome.
3. **Rule Biased Mutation** selects one parent and mutates parameters for existing economy or combat rules with a 50% chance. The mutations are executed by adding a random integer value in the range [-5,5].
4. **Randomization** generates a random new chromosome.

Randomization had a 10% chance of being selected during evolution. The other genetic operators had a 30% chance.

## Results

The results of ten tests of the evolutionary algorithm against each of the two optimised tactics are shown in Table 2. From left to right, the columns show (1) the tactic used by the static player, (2) the number of tests, (3) the lowest fitness value found, (4) the highest fitness value found, (5) the average fitness value, and (6) the number of tests that ended because of the run-stop criterion.

Tactic	Tests	Low	High	Avg	>250
SR	10	0.73	0.85	0.78	2
KR	10	0.71	0.84	0.75	0

Table 2: Evolutionary algorithm results.

The table shows surprisingly high average, highest, and even lowest solution-fitness values. Therefore, it may be concluded that offline adaptive game AI was successful in rapidly discovering game-AI scripts able to defeat both rush tactics used by the static player.

## IMPROVING ADAPTIVE AI

In the first experiment, we discovered that our original implementation of dynamic scripting did not achieve satisfying results against the two rush tactics. In the previous section we evolved new tactics that were able to defeat the two rush tactics. In the present section we discuss how the evolved tactics can be used to improve the rulebases employed by dynamic scripting, to enable it to deal with the

rush tactics with more success. First, we discuss observations on the evolved tactics. Then, we discuss the translation of the evolved tactics to rulebase improvements. Finally, we evaluate of the new rulebases by repeating the first experiment with the new rulebases.

### Observations on the Evolved Tactics

About the solutions evolved against the soldier’s rush, the following observations were made. The soldier’s rush is used on a small map. As is usual for a small map, the game played by the solutions was always short. Most solutions included only two states with activated genes. Basically, we found that all ten solutions counter the soldier’s rush with a soldier’s rush of their own. In eight out of ten solutions, the solutions included building a “blacksmith” very early in the game, which allows the research of weapon and armour upgrades. Then, the solutions selected at least two out of the three possible research advancements, after which large attack forces were created. These eight solutions succeeded because they ensure their soldiers are quickly upgraded to be very effective, before they attack. The remaining two solutions overwhelmed the static player with sheer numbers.

About the solutions evolved against the knight’s rush, the following observations were made. The knight’s rush is used on a large map, which enticed longer games. On average, for each solution five or six states were activated. Against the knight’s rush, all solutions included training a large number of “workers” to be able to expand quickly. They also included boosting the economy by exploiting additional resource sites after setting up defences.

Almost all solutions evolved against the knight’s rush worked towards the goal of quickly creating advanced military units, in particular “knights”. Seven out of ten solutions achieved this goal by employing a specific building order, namely a “blacksmith”, followed by a “lumbermill”, followed by a “keep”, followed by “stables”. Two out of ten solutions preferred a building order that reached state 11 as quickly as possible (see Figure 2). State 11 is the first state that allows the building of knights.

Surprisingly, in several solutions against the knight’s rush, the game AI employed many “catapults”. WARCRAFT II strategy guides generally consider catapults to be inferior military units, because of their high costs and considerable vulnerability. A possible explanation for the successful use of catapults by the evolutionary game AI is that, with their high damaging abilities and large range, they are particularly effective against tightly packed armies, such as groups of knights.

### Improving the Rulebase for Dynamic Scripting

Based on our observations we decided to create four new rules for the rulebases, and to (slightly) change the parameters for several existing combat rules.

The first new rule was designed to be able to deal with the soldier’s rush. The rule contained the pattern that was observed in most of the tactics evolved against the soldier’s rush, namely a combination of the building of a “blacksmith”, followed by the research of several upgrades, followed by the creation of a large offensive force.

The second rule was designed to be able to deal with the knight’s rush. Against the knight’s rush, almost all evolved

solutions aimed at creating advanced military units quickly. The new rule checks whether it is possible to reach a state that allows the creation of advanced military units, by constructing one new building. If such is possible, the rule constructs that building, and creates an offensive force consisting of the advanced military units.

The third rule was aimed at boosting the economy by exploiting additional resource sites. The original rulebases contained a rule to this end, but this rule was invariably assigned a low weight. In the evolved solutions we discovered that exploitation of additional resource sites only occurred after a defensive force was built. The new rule acknowledged this by preparing the exploitation of additional resource sites with the building of a defensive army.

The fourth rule was a straightforward translation of the best solution found against the knight’s rush. Simply all activated genes for each state were translated and combined in one rule, and stored in the rulebase corresponding to the state.

To keep the total number of rules constant, the new rules replaced existing rules, namely rules that always ended up with low weights. Besides the creation of the four new rules, small changes were made to some of the existing combat rules, by changing the parameters to increase the number of units of types clearly preferred by the solutions, and to decrease the number of units of types avoided by the solutions. Through these changes, the use of “catapults” was encouraged. More details on the original and revised rulebases can be found in (Ponsen 2004).

### Evaluating the Improved Rule-base in Wargus

We repeated the first experiment, but with dynamic scripting using the new rulebases, and with the values of the maximum reward and maximum penalty both set to 400, to allow dynamic scripting to reach the boundaries of the weight values faster. Table 3 summarises the achieved results. The columns in Table 3 are equal to those in Table 1.

Tactic	Tests	Low	High	Avg	Med	>100	Won
SBLA	11	10	34	19	14	0	72.5
LBLA	11	10	61	24	26	0	66.4
SR	10					10	27.5
KR	10					10	10.1

Table 3: Evaluation results of dynamic scripting in RTS games using improved rulebases.

A comparison of Table 1 and Table 3 shows that the performance of dynamic scripting is considerably improved with the new rulebases. Against the two balanced tactics, SBLA and LBLA, the average RTP is reduced by more than 50%. Against the two optimised tactics, the soldier’s rush and the knight’s rush, the number of games won out of 100 has increased enormously. Since we observed that dynamic scripting assigned the new rules large weights, the improved performance can be attributed to the new rules.

Note that, despite the improvements, dynamic scripting is still unable to statistically outperform the two rush tactics. The explanation is as follows. The two rush tactics are ‘super-tactics’, that can only be defeated by very specific counter-tactics, with little room for variation. By design,

dynamic scripting generates a variety of tactics at all times, thus it is unlikely to make the appropriate choices enough times in a row to reach the RTP. A possible solution to this shortcoming of adaptive game AI, is to allow it to recognise that an optimised tactic is used, and then oppose it with a pre-programmed “answer” without activating a learning mechanism. Note, however, that since the existence of super-tactics can be considered a weakness of game design, a better solution would be to change the game design before the release of the game, to make super-tactics impossible.

## CONCLUSIONS

We set out to show that offline evolutionary learning can be used to improve the performance of adaptive game AI, by improving the domain knowledge that is used by the adaptive game AI. We implemented an adaptive game AI technique called “dynamic scripting”, which uses domain knowledge stored in rulebases, in the RTS game WARGUS. We tested the implementation against four manually designed tactics. We observed that, while dynamic scripting was successful in defeating balanced tactics, it did not do well against two optimised rush tactics. We then used evolutionary learning to design tactics able to defeat the rush tactics. Finally, we used the evolved tactics to improve the rulebases of dynamic scripting. From our empirical results we were able to conclude that the new rulebases resulted in significantly improved performance of dynamic scripting against all four tactics.

We draw three conclusions from our experiments. (1) Dynamic scripting can be successfully implemented in RTS games. (2) Offline evolutionary learning can be used to successfully design counter-tactics against strong tactics used in an RTS game. (3) Tactics designed by offline evolutionary learning can be used to improve the domain knowledge used by adaptive game AI, and thus to improved performance of adaptive game AI.

### Future Work

It can be argued that a game is entertaining when the game AI attempts matching the playing strength of the human player, instead of defeating the human player. In parallel research, techniques have been investigated that allow dynamic scripting to scale the difficulty level of the game AI to match the human player’s skill, instead of optimise it (Spronck, Sprinkhuizen-Kuyper and Postma 2004c). In future work we will add difficulty-scaling enhancements to dynamic scripting in RTS games. We will also test dynamic scripting in RTS games played against humans, to determine if adaptive game AI actually increases the entertainment value of a game.

In the present research, the translation of the evolved solutions to improvements in domain knowledge was done manually. Because the translation requires understanding and interpretation of the evolved solutions, it is difficult to perform the translation automatically. Nevertheless, in future work we will attempt to design an automated mechanism that translates tactics evolved by offline evolutionary learning into an improved rulebase for dynamic scripting. The addition of such a mechanism would enable us to completely automate the process of designing successful rulebases for dynamic scripting.

## REFERENCES

- Brockington, M and M. Darrah. 2002. "How *Not* to Implement a Basic Scripting Language." *AI Game Programming Wisdom* (ed. S. Rabin), Charles River Media, Hingham, MA, pp. 548–554.
- Buckland, M. 2004. "Building better Genetic Algorithms." *AI Game Programming Wisdom 2* (ed. S. Rabin), Charles River Media, Hingham, MA, pp. 649–660.
- Buro, M. 2003. "RTS Games as Test-Bed for Real-Time AI Research". *Proceedings of the 7th Joint Conference on Information Science (JCIS 2003)* (eds. K. Chen *et al.*), pp. 481–484.
- 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* (eds. Q. Medhi, N. Gough and M. Cavazza), SCS Europe Bvba, pp. 113–120.
- Gold, J. 2004. *Object-Oriented Game Development*, Addison-Wesley, harrow, UK.
- Goldberg, D.E. 1989. *Genetic Algorithms in Search, Optimization & Machine Learning*, Addison-Wesley, Reading, UK.
- Johnson, S. 2004. "Adaptive AI: A Practical Example." *AI Game Programming Wisdom 2* (ed. S. Rabin), Charles River Media, Hingham, MA, pp. 639–647.
- 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*, Technical Report FS-00-03. AAAI Press 2000, pp. 80–87.
- Laird, J.E. 2001. "It Knows What You're Going To Do: Adding Anticipation to a Quakebot." *Proceedings of the Fifth International Conference on Autonomous Agents* (eds. J.P. Müller *et al.*), ACM Press, Montreal, Canada, pp. 385–392.
- Manslow, J. 2002. "Learning and Adaptation." *AI Game Programming Wisdom* (ed. S. Rabin), Charles River Media, Hingham, MA, pp. 557–566.
- Manslow, J. 2004. "Using reinforcement learning to Solve AI Control Problems." *AI Game Programming Wisdom 2* (ed. S. Rabin), Charles River Media, Hingham, MA, pp. 591–601.
- Ponsen, M. 2004. *Improving Adaptive AI with Evolutionary Learning*. MSc Thesis, Delft University of Technology.
- Rabin, S. 2004. *AI Game Programming Wisdom 2*. Charles River Media, Hingham, MA.
- Russel, S. and J. Norvig. 1995. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Pearson Education, Upper Saddle River, NJ.
- Schaeffer, J. 2001. "A Gamut of Games." *AI Magazine*, Vol. 22, No. 3, pp. 29–46.
- Spronck, P., I. Sprinkhuizen-Kuyper, and E. Postma. 2004a. "Online Adaptation of Game Opponent AI with Dynamic Scripting." *International Journal of Intelligent Games and Simulation* (eds. N.E. Gough and Q.H. Mehdi), Vol. 3, No. 1, University of Wolverhampton and EUROSIS, pp. 45–53.
- Spronck, P., I. Sprinkhuizen-Kuyper, and E. Postma. 2004b. "Enhancing the Performance of Dynamic Scripting in Computer Games." *Entertainment Computing – ICEC 2004* (ed. M. Rauterberg), LNCS 3166, Springer-Verlag, pp. 273–282.
- Spronck, P., I. Sprinkhuizen-Kuyper, and E. Postma. 2004c. "Difficulty Scaling of Game AI." *Proceedings of the GAME-ON 2004 Conference*. (To be published)
- Tozour, P. 2002. "The Perils of AI Scripting." *AI Game Programming Wisdom* (ed. S. Rabin), Charles River Media, Hingham, MA, pp. 541–547.