# Knowledge Acquisition for Adaptive Game AI

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

[1]Maastricht University / MICC-IKAT; P.O. Box 616, NL-6200 MD Maastricht
[2]Department of Computer Science and Engineering;
Lehigh University; Bethlehem, PA 18015
[3]Navy Center for Applied Research in Artificial Intelligence;
Naval Research Laboratory (Code 5515); Washington, DC 20375
[1]{m.ponsen,p.spronck}@micc.unimaas.nl, [2]hem4@lehigh.edu, [3]david.aha@nrl.navy.mil

**Abstract:** Game artificial intelligence (AI) controls the decision-making process of computer-controlled opponents in computer games. Adaptive game AI (i.e., game AI that can automatically adapt the behaviour of the computer players to changes in the environment) can increase the entertainment value of computer games. Successful adaptive game AI is invariably based on the game's domain knowledge. We show that an offline evolutionary algorithm can learn important domain knowledge in the form of game tactics (i.e., a sequence of game actions) for dynamic scripting, an offline algorithm inspired by reinforcement learning approaches that we use to create adaptive game AI. We compare the performance of dynamic scripting under three conditions for defeating non-adaptive opponents in a real-time strategy game. In the first condition, we manually encode its tactics. In the second condition, we manually translate the tactics learned by the evolutionary algorithm, and use them for dynamic scripting. In the third condition, this translation is automated. We found that dynamic scripting performs best under the third condition, and both of the latter conditions outperform manual tactic encoding. We discuss the implications of these results, and the performance of dynamic scripting for adaptive game AI from the perspective of machine learning research and commercial game development.

**Keywords:** Computer games, artificial intelligence, real-time strategy, reinforcement learning, dynamic scripting, evolutionary algorithm, knowledge acquisition

## 1 Introduction

Today's gaming environments are becoming increasingly realistic, especially in terms of the graphical presentation of the virtual world. However, to further increase realism, the reasoning capabilities of characters 'living' inside these virtual worlds must be addressed (Laird & van Lent, 2001). People from both the game industry (Rabin, 2004) and academia (Laird & van Lent, 2001) predicted an increasing importance of artificial intelligence (AI) in computer games.

The term *game AI* is used differently by game developers and academic researchers (Gold, 2004). Academic researchers restrict the use of this term to refer to intelligent behaviours of game characters (Allen *et al*., 2001). In contrast, for game developers *game AI* is used in a broader sense to encompass techniques such as pathfinding, animation systems, level geometry, collision physics, vehicle dynamics, and even the generation of random numbers (Tomlinson, 2003). In this paper we use this term in the narrower, academic sense.

High-quality game AI will increase the game playing challenge (Nayerek, 2004) and is a potential selling point for a game. Development time for game AI is usually short; most game companies assign graphics and storytelling the highest priorities (for marketing reasons) and typically assign the implementation of game AI to the end of the development process (Nayerek, 2004), which complicates designing and testing strong game AI. That is why even in state-of-the-art games, game AI is generally of inferior quality (Schaeffer, 2001; Buro, 2004; Gold, 2004). Game AI can benefit from academic research into commercial games (Forbus and Laird, 2002).

Adaptive game AI, which concerns methods for adapting the behaviour of computer-controlled opponents, can potentially increase the quality of game AI. However, to ensure the reliability of adaptive game AI, it must incorporate a sufficient amount of correct prior domain

knowledge (Manslow, 2002). If the incorporated domain knowledge is incorrect or insufficient, adaptive game AI will not be reliable, and be unable to generate satisfying results.

*Dynamic scripting* is an offline reinforcement learning technique that can be used to implement adaptive AI (Spronck *et al.*, 2006). We implemented dynamic scripting in a real-time strategy (RTS) game called *Wargus*, an open-source clone of the popular Warcraft II™ game. Our machine learning mechanism in Wargus focuses on an ambitious performance task, namely winning RTS games. The quality of the knowledge base (i.e., the set of available actions) is essential for achieving good performance with dynamic scripting.

To generate knowledge bases for use by the adaptive game AI opponents, we envision three alternatives. The first alternative is to *manually* encode the knowledge bases. This may take a long time, which game developers generally don't have. Furthermore, there is a considerable risk that the knowledge bases are substantially sub-optimal due to analysis and encoding errors. Consequently, the adaptive game AI may not generate satisfying results.

For the second alternative, we investigated whether *semi-automatically* improving the knowledge bases can increase the performance of the adaptive game AI. The semi-automatic approach involves running machine learning experiments to discover strong tactics (i.e., action sequences) offline after which they are manually added to knowledge bases. We implemented an evolutionary algorithm in Wargus to search the space of effective tactics. Afterwards, we manually extracted tactics from among those discovered and added them to the knowledge bases. The improved adaptive game AI should be able to perform better versus strong players, and be more efficient in finding tactics of a desired effectiveness. This approach alleviates some of the difficulties with the manual approach, but manually modifying knowledge bases can still be cumbersome and time consuming.

The third alternative is to *automatically* generate the knowledge bases. As a first step, we again use an offline evolutionary algorithm. However, unlike the semi-automatic approach where we manually extracted the tactics from the evolved action sequences, the second step of this alternative automatically transfers the domain knowledge obtained in the first step to the knowledge bases.

We report empirical results, which have been previously discussed by Ponsen *et al.* (2006a), showing that the automatic approach outperforms the manual and semi-automatic approaches. Therefore, we conclude that, at least for Wargus, high-quality domain knowledge used by the adaptive AI opponents can be automatically generated.

This paper continues as follows. Section 2 discusses related work. Section 3 describes RTS games and the complexity of Wargus. Section 4 discusses how dynamic scripting was implemented in Wargus, while Section 5 introduces the evolutionary algorithm we used. Section 6 evaluates dynamic scripting's performance for the three competing knowledge acquisition approaches: manual, semi-automatic and automatic. Section 7 discusses the results, and Section 8 presents conclusions and future work.

## 2 Related Work

Although many studies exist on learning to win classical board games and other games with small search spaces, few studies exist on learning to win complex strategy games. In recent years, some AI researchers (Laird and Van Lent, 2001; Buro, 2004) have begun focusing on complex strategy games. Game agents require sophisticated representations and reasoning capabilities to perform competently in these environments, which are challenging to construct (Forbus *et al.*, 2001). For this reason, existing research efforts on complex strategy games often focus on simpler tasks. For example, Guestrin *et al.* (2003) applied relational Markov decision process models to some limited Wargus scenarios (e.g., 3×3 combat). Similarly, Cheng and Thawonmas (2004) proposed a case-based plan recognition approach for assisting Wargus players, but only for low-level management tasks. Unlike these experiments, we are focussing on the ambitious performance task of winning real-time strategy games by reducing the complexity of Wargus through (automatic) knowledge acquisition.

Knowledge acquisition approaches are being investigated by many AI researchers (cf., Blythe *et al.*, 2001, Ilghami *et al.*, 2002, Winner and Veloso, 2003). However, very little work has been done

on acquiring domain knowledge for game AI. We distinguish three classes of approaches: (1) manual, (2) semi-automatic, and (3) automatic.

**Manual knowledge acquisition:** Research on these approaches concentrates on providing tools to facilitate the knowledge acquisition process. Some games (e.g., Age of Empires™ and Command and Conquer Generals™) include tools to encode new domain knowledge used by the game AI.

**Semi-automatic knowledge acquisition:** Research on these approaches concentrates on developing tools that allow the improvement of manually created knowledge. For example, Street *et al.* (2001) report on tools using pattern recognition techniques developed to help balance the capabilities of RTS units. Typical RTS games implement the rock-scissors-paper principle. One unit may be well suited to destroy a particular kind of unit or game element. However, this unit itself is particularly vulnerable to attacks from other kinds of units. The problem is compounded by the fact that modern RTS games such as Age of Empires™ offer different playing sides (usually called *races*). Each race has unique units and properties. This makes it very difficult for game developers to find an adequate balance.

**Automatic knowledge acquisition:** Research on these approaches concentrates on applying them to classic board games. For example, Kirby (2003) was successful in applying neural networks to acquire domain knowledge for Backgammon, and partially successful in applying them to Go and Chess. The main difficulty in using such approaches for game AI is that they require training examples to be annotated with information describing how various transformations took place in the domain. This requirement can be difficult to fulfil in actual games. In our automated knowledge acquisition approach, we require as input only some pre-defined scripts, which RTS games typically provide.

## 3 Real-time Strategy Games

Real-Time Strategy (RTS) is a category of strategy games that usually focus on military combat. RTS games such as Warcraft™ and Empire Earth™ require the player to control armies (consisting of different types of units) and 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 distributing these resources over the various game 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 game actions that are executed sequentially (Tozour 2002). We define a *game action* as an atomic transformation in the game situation. Typical game 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 paper: *tactics* are action sequences consisting out of one or more atomic game actions, and *strategies* consists of a sequence of tactics that can be used to play a complete game.

### 3.1 Wargus

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 (illustrated in Figure 1) implements a clone of the popular RTS game Warcraft II™. In the context of Wargus, a complete script represents an opponent strategy, and a sub-collection of game actions in a script represents a tactic. A tactic can be as simple as one game action, i.e., “build a lumber mill”, or as complex as a sequence of actions, i.e., “build a lumber mill, then build a defensive army consisting of soldiers, then research new weaponry, and finally replace the town hall by a keep”. We had four opponent strategies at our disposal for running our machine learning experiments:


**Figure 1**: A screen shot of a Wargus game.

1. **Small Balanced Land Attack (SBLA):** This strategy keeps a balance between offensive actions, defensive actions, and research. It is effective against many different playing styles. The SBLA is applied on a small map.
2. **Large Balanced Land Attack (LBLA):** This is a similar strategy to the SBLA, but applied on a large map.
3. **Soldier's Rush (SR):** This attempts to overwhelm the opponent with cheap military units in an early state of the game. Since SR works best in fast games, we apply 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. Since KR works best in slower-paced games, we apply it on a large map.

### 3.2 Reducing the Complexity of Wargus

RTS games include a wide variety of possible tactics that can be selected at any point in the game. Typically, games such as Wargus are designed so that no single tactic dominates all others; they rather follow the rock-paper-scissors principle (i.e., some tactics are particularly well suited against other particular tactics but are themselves vulnerable against others). For example, solely focusing attention on training an army might cause a lag in research accomplishments, which prevents creating army units that are as strong as the neighbour's. In contrast, neglecting the army can lead to a crushing defeat at the hands of a strong neighbour. A continuous balance must be maintained among the potential tactics. Intelligent decisions should be based on the current game situation and the (predicted) decision model of the opponents. However, RTS games include only partially observable environments that contain adversaries who modify the game state asynchronously, and whose decision models are unknown, thereby making it infeasible obtain complete information on the current game situation. In addition, to successfully play an RTS game, players must make their decisions in real-time (i.e., under severe time constraints) and execute multiple orders simultaneously. We believe that these properties of RTS games make them a very complex and challenging test-bed for AI research.

RTS games contain a comparatively large action space, which is defined as the set of possible actions that can be executed at a particular moment. We roughly estimate the action space in Wargus to be $O(2^W(A \cdot P) + 2^T(D+S) + B(R+C))$, where $W$ is the current number of workers, $A$ is the number of assignments workers can perform (e.g., create a building, gather gold), $P$ is the average number of workplaces, $T$ is the number of troops (fighters plus workers), $D$ is the average number of directions that a unit can move, $S$ is the number of choices for a troop's stance (i.e., stand, patrol, attack), $B$ is the

number of buildings, *R* is the average number of choices for research objectives at a building, and *C* is the average number of choices for units to create at a building. For the simple early game scenario shown in Figure 1 (which includes some off-screen troops and an off-screen building), this estimate yields a decision complexity of $1.5 \times 10^3$, which is substantially higher than the average number of possible moves in many board games (e.g., for chess, this is approximately 30). While the judicious application of domain knowledge can reduce this high number to a few dozen sensible decisions, acquiring this background knowledge is challenging.

Reinforcement learning techniques, such as dynamic scripting, learn a policy that maps actions to specific game situations. In the case of Wargus, with an estimated action space of $1.5 \times 10^3$ and an even larger state space, learning becomes infeasible without abstractions. To reduce the complexity of Wargus, we created an abstraction of the *state space* by designing the building state lattice displayed in Figure 2. Consisting of 20 states, it defines sequences of building constructions that can occur during a Wargus game, where each state corresponds to the types of constructed buildings, which in turn determine the unit types and technologies that can be researched. Consequently, state changes are spawned by tactics that create new buildings. For example, starting with a town hall and barracks, the next building choices are a lumber mill, a blacksmith, or a keep (which replaces the town hall). Building one of these causes a transition from state 1 to states 2, 3, or 4, respectively.

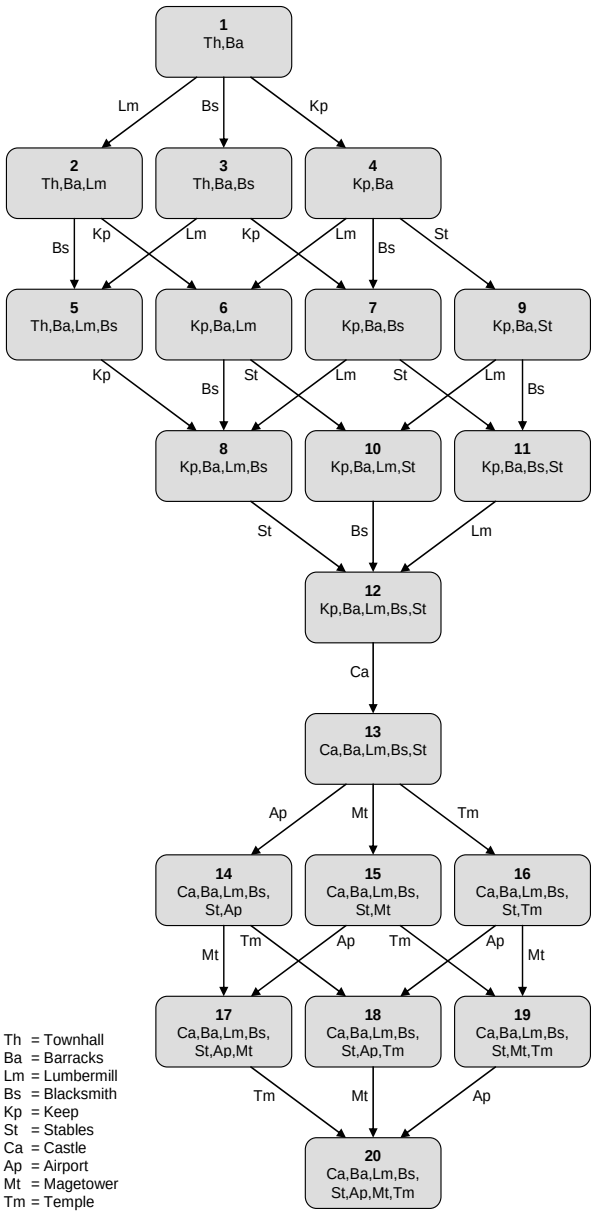



**Figure 2:** A building-specific state lattice for Wargus, where nodes represent states (defined by a set of completed buildings), and state transitions involve constructing a specific building.

We further reduced the Wargus complexity by constraining the *action space* using a high-level language for game actions. The high-level orders, listed in Table 1, interface with the available API provided by the Stratagus engine. This high-level API represents all possible game actions in the Wargus game on an abstract level. A typical high-level order is to construct a particular building. Deciding the best place to construct the building and deciding which worker will be assigned to the task is left to the engine, and will not take part in the search space for the machine learning algorithm. Another high-level order is to inform the AI to attack the opponent with an army. The training of individual soldiers and the exact details of the attack (e.g., planning an attack route, selecting a target) are also determined by the Stratagus engine.

Together, the building lattice and the list of high-level orders constrain the search space of useful strategies to a manageable size.

**Table 1:** Description of the available high-level actions in Wargus.

| **Wargus Game AI Actions** | |
|---|---|
| AiForce (ForceID, {force})<br>*e.g., AiForce(1,{"unit-grunt", 3})* | Define a force: determine the unit types and number of units that belong to it. |
| AiCheckForce (ForceID)<br>*e.g., AiCheckForce(1)* | Check if a force is complete and ready for combat. |
| AiAttackWithForce (forceID)<br>*e.g., AiAttackWithForce(1)* | Command the AI to attack an enemy with all units belonging to a predefined force. |
| AiForceRole (forceID, role)<br>*e.g.,AiForceRole(1,"defend")* | Define the role of a force: Assign it either a defensive or offensive role. |
| AiNeed (unitType)<br>*e.g., AiNeed("unit-footmen")* | Command the AI to train or build a unit of a specific unit type (e.g., request the training of a soldier). |
| AiResearch (researchType)<br>*e.g., AiResearch("upgrade-sword")* | Command the AI to pursue a specific research advancement. |
| AiUpgradeTo (unitType)<br>*e.g.,AiUpgradeTo("upgrade-ranger")* | Command the AI to upgrade a specific unit. |

## 4 Dynamic Scripting for Wargus

Game AI for complex games, such as Wargus, is mostly defined in scripts. Because scripts tend to be long and complex (Brockington and Darrah, 2002), they are likely to contain weaknesses, which human players can exploit. Spronck *et al.* (2006) introduced a technique, called *dynamic scripting*, which can be used to generate AI opponent scripts that have the ability to adapt to a human player's behaviour. Dynamic scripting generates scripts for each computer-controlled opponent at the start of an encounter (i.e., a fight between opposing teams), by randomly selecting a number of tactics from a specific knowledge base. The tactics are designed using domain-specific knowledge. The probability that a tactic is selected for a script is an increasing function of its associated weight value.

The learning mechanism in the dynamic scripting technique is based on reinforcement learning techniques (Sutton and Barto, 1998). In dynamic scripting, learning proceeds as follows. Upon completion of an encounter, the weights of the tactics employed during the encounter are adapted depending on their contribution to the outcome. Tactics that lead to success are rewarded with a weight increase, whereas tactics that lead to failure are punished with a weight decrease. The size of the weight changes is determined by a weight-updating function. The increment or decrement of each weight is compensated for by decreasing or increasing all remaining weights so as to keep the summed total of weights in a knowledge base constant. Through the process of punishments and rewards, dynamic scripting adapts to the human player in only a few trials.

Dynamic scripting can be applied to any form of game AI that meets three requirements: (1) the game AI can be scripted, (2) domain knowledge on the characteristics of a successful script can be collected, and (3) an evaluation function can be designed to assess the success of the function's execution. Such functions are not only found in games, but also in application areas, such as multi-agent systems. Dynamic scripting has proven to be fast, effective, robust, and efficient (Spronck *et al.*, 2006).

The next subsections discuss the dynamic scripting implementation in Wargus. In Subsection 4.1 we discuss how tactics are extracted from a knowledge base to generate a dynamic script. In Subsection 4.2 we describe the process for adapting the knowledge base.

### 4.1 Knowledge bases and Game States in Wargus

Typically, players in a RTS game such as Wargus 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 a RTS game mainly depend on the availability of different unit types and technologies.

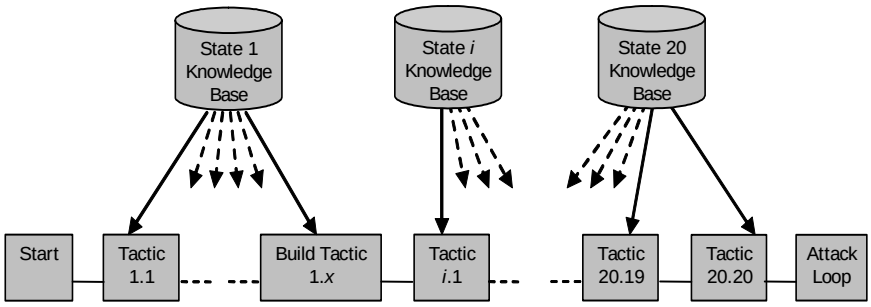



**Figure 3:** Schematic representation of dynamic script generation in Wargus

For a dynamic scripting implementation in Wargus, we must constrain the adaptive AI's tactics selection process. Therefore, we divided the game into a small number of distinct game states. Each state corresponds to a unique knowledge base whose tactics can be selected by the dynamic scripting technique when the game is in that particular state. We distinguish Wargus game 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. Note that not all tactics include build actions.

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 behaviour, each tactic is restricted to be selected only once per state. Tactic selection continues until either a total of $N$ tactics are selected ($N$=100 was used for the experiments) or until a final state was reached. The value for $N$ was determined during initial experiments: it was set sufficiently high so that the adaptive game AI almost always reached the final state in which it possessed all relevant buildings. For this final state, another $M$ tactics are 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 opponent civilizations. The dynamic script generation process is illustrated in Figure 3.

### 4.2 Weight Adaptation in Wargus

Weight updates in Wargus are based on both an evaluation of the performance of the adaptive AI during the whole game (called the *overall fitness*), and between state changes (called the *state fitness*). As such, the weight-updating function is based on a combination of state fitness and overall fitness. Using both evaluations for weight updating increases the learning mechanism's efficiency (Manslow, 2004). 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:

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

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:

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

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 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:

$$S_x = 0.7M_x + 0.3B_x \tag{3}$$

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). We prioritize military points because experiences indicate that these are a better indication for the success of tactics than building points.

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

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

In Equation 4, W is the new weight value, $W_{org}$ is the current weight value before the 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 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. 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 Evolutionary Algorithm in Wargus

For complex games such as Wargus, it is likely that the AI designers who are responsible for encoding the knowledge bases overlook certain interactions between game actions. Consequently, the incorporated domain knowledge may be sub-optimal, resulting in a weak or easily defeatable game AI. Testing every combination of game actions is an impossible task for an AI designer, especially given the short amount of time available for AI tuning. An offline learning mechanism can test out many more AI variations than an individual developer can (Chan *et al.*, 2004). In this section, we explain the process of evolving domain knowledge for RTS games using an evolutionary algorithm (EA). The goal of the EA in Wargus is to use offline learning to discover tactics that can be used to defeat static (i.e., non-adaptive) opponent strategies. In the following subsections, we describe the encoding of the chromosome (5.1), the fitness function (5.2), and the genetic operators (5.3).

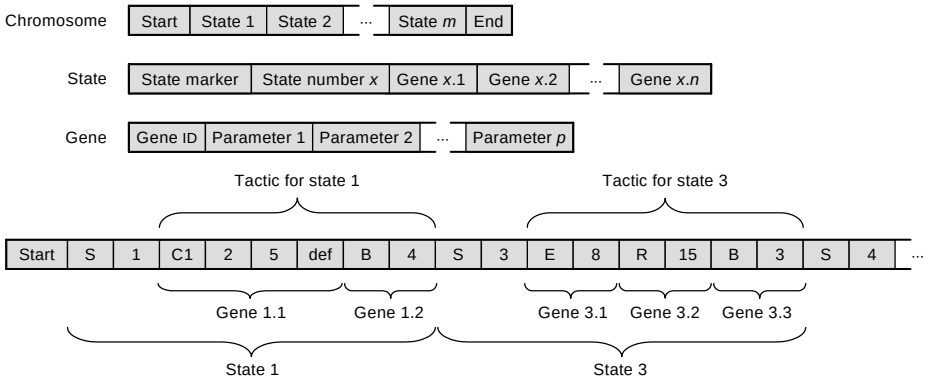



**Figure 4:** Design of a chromosome to store game AI for Wargus.

### 5.1 Encoding

EA works with a population of chromosomes (in our experiments we use a population of size 50, which proved sufficient during preliminary experiments to rapidly discover a variety of strong tactics), 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.

### 5.2 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 chromosome), which yields a value in the range [0,1]:

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

In Equation 5, $M_d$ represents the military points for the dynamic player, $M_o$ the military points for the dynamic player's opponent, and $b$ is the break-even point. $C_t$ represents the game cycle (i.e., the time it took before the game is lost by one of the players, or the game was aborted because time ran out). $C_{max}$ represents the maximum game cycle (i.e., the longest time a game is allowed to continue). When a game reaches the end cycle and neither army has been defeated, scores at that time are measured and the game is aborted. The factor $C_t$ / $C_{max}$ ensures losing chromosomes that play a long game are awarded higher fitness scores than losing chromosomes that play 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. We set the target value to 0.70, which represents a clear victory for the dynamic player controlled by the evolved strategy. Since there is no guarantee that a chromosome exceeding the target value will be found, the evolution process also ends after it has generated a maximum number of chromosomes. 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.

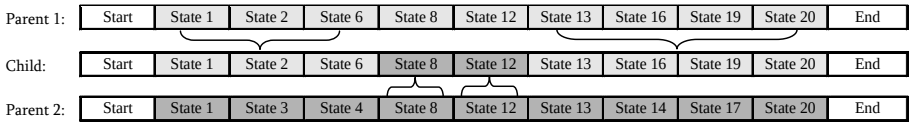



**Figure 5:** Example of a state crossover

### 5.3 Genetic Operators

Relatively successful chromosomes (as determined by a 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). To breed new chromosomes, we implemented four genetic operators. By design, all four ensure that a child chromosome always represents legal game AI. The four genetic operators take into account *activated* genes, which are genes representing 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. The four genetic operators are the following:

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, i.e., the player always passes through these 4 states (unless the game ends prematurely). 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. The process is illustrated in Figure 5. In the figure, parent switches occur at state 8 and state 13.

2. **Gene Replace Mutation** selects one parent, and replaces economy, research or combat genes with a 25 percent probability. It is allowed to replace a gene of a certain type with a different gene type (e.g., it is allowed to replace a research gene with a combat gene). Building genes are excluded, both for and as replacement, because these could spawn a state change and thus could possibly corrupt the chromosome.

3. **Gene Biased Mutation** selects one parent and mutates parameters for existing economy or combat genes with a 50 percent 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 percent chance of being selected during evolution. The other genetic operators had a 30 percent chance. With our randomization genetic operator and relatively high mutation and crossover rates, we stimulate diversity in the population in order to broadly search the enormous space of possible strategies in the Wargus game, in effect hoping to find a wide variety of different tactics.

## 6 Performance Evaluation of Dynamic Scripting in Wargus

We evaluated the performance of dynamic scripting under three conditions: using knowledge bases that were (1) manually acquired, (2) semi-automatically acquired, and (3) automatically acquired. We evaluated the performance of these three knowledge acquisition approaches 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 ended when of the players was defeated, or when a certain period of time had elapsed. If the game ended due to the time restriction, the player with the highest score (calculated using Equation 3) was considered to have won. After the game, the knowledge bases were adapted and used in the next game. A sequence of 100 games constituted one test. We tested four strategies for the static player, namely the SBLA, LBLA, SR, and KR (introduced in Section 3.1).

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 a 90 percent probability in favour of the dynamic player. RTP is the number of the first game in which the dynamic player statistically outperforms the static player. A low RTP value indicates good efficiency for dynamic scripting.

In the following sub-sections, we will describe the process for encoding the knowledge bases and the results for dynamic scripting using these knowledge bases.

### 6.1 Evaluation of the Manual Approach

For the first approach, we manually encoded the knowledge bases from scratch. The manually encoded (ME) knowledge bases consisted of 50 higher-level tactics, each consisting of a single atomic game action (e.g., constructing a blacksmith). Tactics can be classified into four basic categories, (1) build tactics for constructing buildings (12 tactics), (2) research tactics for acquiring new technologies (9 tactics), (3) economy tactics for stimulating resource gathering (4 tactics), and (4) combat tactics for offensive and defensive military operations (25 tactics). The tactics were designed based on the domain knowledge found in strategy guides for Warcraft II™ and our 'common sense' of RTS games. A typical tactic in the knowledge bases allows the dynamic player to launch an attack at his opponent. The domain knowledge here lies in the fact that this tactic automatically trains the most advanced units available. Most strategy guides indicated that in Warcraft II™ it is advisable to always attack with the most advanced units available, e.g., a knight can slaughter a group of soldiers. Another form of built-in domain knowledge is incorporated in the building tactics. According to most strategy guides, it is important to build more than one barrack. On the other hand, it doesn't really make sense to build more than one blacksmith, so we prevent the AI from doing this. We expected it to be crucial to regularly launch firm attacks and to have a steady defensive line at all times. For that reason half the tactics inserted in the knowledge bases were military tactics.

We set $P$ to 175, $R$ to 200, $W_{max}$ to 1250, $W_{min}$ to 25 and $b$ to 0.5. The results of the evaluation of dynamic scripting using the ME knowledge bases in Wargus are displayed in Table 2. The columns of the table represent, from left to right: (1) the strategy used by the static player, (2) the number of tests, (3) the average RTP calculated over all tests, (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.

The results for the ME knowledge bases show relatively low values for the average RTPs for both the SBLA and the LBLA. Therefore, we conclude that the dynamic player efficiently adapts to these two opponent strategies. However, the dynamic player was unable to adapt to the SR and the KR within 100 games. The dynamic player only won on average 1 out of 100 games against the SR, and 1 out of 50 games against the KR.

**Table 2:** Evaluation of dynamic scripting in Wargus using the ME knowledge bases.

| ME knowledge bases | | | | |
|---|---|---|---|---|
| Strategy | Tests | RTP | >100 | Won |
| SBLA | 31 | 50 | 0 | 60 |
| LBLA | 21 | 49 | 0 | 60 |
| SR | 10 | - | 10 | 1 |
| KR | 10 | - | 10 | 2 |

We believe that the reason for the inferior performance of the dynamic player against the two rush strategies can be ascribed to the fact that these strategies are optimized and can only be defeated by very specific counter-tactics, with little room for variation. It is therefore very hard to design adaptive game AI that can defeat these rush strategies consistently. Another issue may be that the knowledge bases do not contain the appropriate knowledge to easily design game AI that can beat the rush strategies. For the next approach we investigated whether improving the domain knowledge improves the performance of dynamic scripting against the rush strategies.

### 6.2 Evaluation of the Semi-Automatic Approach

For our second approach we manually improved the ME knowledge bases based on offline evolved domain knowledge. We will refer to these knowledge bases as the manually improved (MI) knowledge bases.

#### Evolving Domain Knowledge

We employed the evolutionary algorithm described in Section 5 to discover strong tactics offline that can perform well against the manually designed strategies that the adaptive game AI was unable to beat using the ME knowledge bases, namely the SR and KR strategies (see Table 2). The results of ten tests against each of the two strong scripts are shown in Table 3. From left to right, the columns show (1) the strategy used by the static player, (2) the number of tests, (3) the average fitness value, and (4) the number of tests that ended because of the run-stop criterion. Based on the reported average fitness scores in Table 3, we conclude that the evolutionary algorithm was successful in discovering static strategies able to defeat the SR and KR scripts. All but two solutions had a fitness score higher than our desired target fitness, which represents a clear victory.

**Table 3:** Evolutionary algorithm results.

| Strategy | Tests | Avg. | >250 |
|---|---|---|---|
| SR | 10 | 0.78 | 2 |
| KR | 10 | 0.75 | 0 |

#### Observations on the Evolved Chromosomes

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

We made the following observations about the chromosomes evolved against the KR. First, the KR is used on a large map, which frequently resulted in longer games. As an indication of this, on average for each chromosome five or six states were activated. Against the KR, all chromosomes 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 chromosomes evolved against the KR worked towards the goal of quickly creating advanced military units, in particular knights. Seven out of ten chromosomes achieved this goal by employing a specific building order, namely a blacksmith, followed by a lumber mill, followed by a keep, followed by stables. Two out of ten chromosomes followed 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 chromosomes against the KR, 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 is that, with their high damaging abilities and large range, they are particularly effective against tightly packed armies, such as groups of knights.

**Manually Improving the Knowledge bases**

We manually extracted strong tactics from the evolved chromosomes and incorporated these into the MI knowledge bases. Based on our observations we decided to create four new tactics for the knowledge bases, and to (slightly) change the parameters for several existing combat tactics.

The first new tactic was designed to be able to deal with the SR. The tactic contained the pattern that was observed in most of the evolved chromosomes against the SR, 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 tactic was designed to be able to deal with the KR. Against the KR, almost all evolved chromosomes aimed at creating advanced military units quickly. The new tactic checks whether it is possible to reach a state that allows the creation of advanced military units, by constructing one new building. If this is possible, the tactic constructs that building, and creates an offensive force consisting of the advanced military units.

The third tactic was aimed at boosting the economy by exploiting additional resource sites. In the evolved chromosomes, we discovered that exploitation of additional resource sites only occurred after a defensive force was built. The new tactic mimics this by preparing the exploitation of additional resource sites with the building of a defensive army.

The fourth tactic was a straightforward translation of the best chromosomes found against the KR. Simply all activated genes for each state were translated and combined in one tactic, and stored in the knowledge base corresponding to the state.

Besides the creation of the four new tactics, small changes were made to some of the existing combat tactics by changing the parameters to increase the number of units of types clearly preferred by the chromosomes, and to decrease the number of units of types avoided by the chromosomes. Through these changes, the use of catapults was encouraged.

**Results**

To empirically validate whether the changes to the ME knowledge bases resulted in an improved performance for dynamic scripting, we repeated the experiments, now using the MI knowledge bases. Table 4 summarizes the results. We set the values of the maximum reward and maximum penalty to 400, to allow dynamic scripting to reach the boundaries of the weight values faster. The columns represent the same variables as used in Table 2.

**Table 4:** Evaluation of dynamic scripting in Wargus using the MI knowledge bases.

| MI knowledge bases | | | | |
|---|---|---|---|---|
| Strategy | Tests | RTP | >100 | Won |
| SBLA | 11 | 19 | 0 | 72 |
| LBLA | 11 | 24 | 0 | 66 |
| SR | 10 | - | 10 | 27 |
| KR | 10 | - | 10 | 10 |

A comparison of the results with the ME and MI knowledge bases show that the performance of dynamic scripting is considerably improved against all opponent strategies. Against the two balanced strategies, SBLA and LBLA, the average RTP is reduced by more than 50 percent. Against the two optimized strategies, the SR and the KR, the number of games won out of 100 has increased considerably. We conclude that the manual changes (based on evolved chromosomes against the KR and SR) to the ME knowledge bases improved performance, in particular because during the experiments we observed that dynamic scripting assigned the new tactics large weights. However, despite these improvements, dynamic scripting still cannot statistically outperform the two rush strategies.

### 6.3 Evaluation of the Automatic Approach

Both the manual and semi-automatic approaches can be very costly in terms of time. For our third approach, we completely automated the process of generating knowledge bases. These knowledge bases are henceforth called the automatically evolved (AE) knowledge bases. The steps for automatically generating knowledge bases are schematically illustrated in Figure 6.

#### Evolving Domain Knowledge

The first step in the automatic approach (EA) involves using the evolutionary algorithm described in Section 5 to search for counter-strategies that defeat clearly distinguishable opponent strategies. The opponent strategies (i.e., game scripts) are provided to EA as a training set, the only manual input required. This training set contains (manually designed) static scripts and (automatically generated) evolutionary scripts. Static scripts are the default scripted opponents typically provided with early versions of the game engine to record the strategies often employed by human players while testing alpha and/or beta versions of the game engine. In contrast, an evolutionary script is a previously evolved chromosome that will be used as an opponent strategy to evolve new chromosomes. Static scripts have the advantage that they are usually of high quality (since they are recorded from human player strategies). In contrast, evolutionary scripts have the advantage that they can be generated completely automatically. Our training set includes the default-scripted opponents provided with the Stratagus engine (strategies 1 to 4 introduced in Section 3.1), and the evolutionary scripts (36 strategies). The output of the evolutionary algorithm consists of a set counter-strategies for defeating the scripts in the training set.

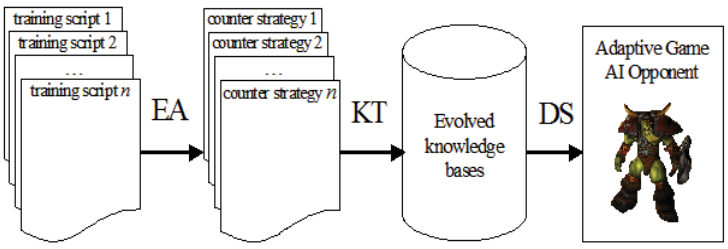



**Figure 6:** Schematic representation of the automatic knowledge acquisition process

### Automatically Improving the Knowledge Bases

The second step (KT) automatically transfers the domain knowledge stored in the evolved chromosomes to the knowledge bases that are used by the adaptive AI mechanism, in this case dynamic scripting. Unlike the semi-automatic approach discussed in Subsection 6.2, here we automatically recognize and extract tactics from the evolved chromosomes. The applicability of 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. Therefore, we can distinguish tactics based on the game states for Wargus as illustrated in Figure 2. All genes (i.e., the sequence of all game actions) grouped in an activated state (an activated state includes at least one activated gene) in a chromosome are considered to be a single tactic. The example chromosome in Figure 4 illustrates two potential tactics. The first tactic for state 1 includes genes 1.1 and 1.2. This tactic will be inserted into the knowledge base for state 1. Because gene 1.2 spawns a state change, the next genes will contribute to a different tactic for a different state.

### Results

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 (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 AE knowledge bases.

We repeated the experiments with dynamic scripting using the AE knowledge bases. The experimental parameters for dynamic scripting were unchanged. Table 5 summarizes the results. Dynamic scripting with the AE knowledge bases outperforms both balanced strategies before any learning occurs (e.g., before weights are adapted). In previous tests against the SR (using the ME and MI knowledge bases), dynamic scripting was unable to find an RTP. In contrast, dynamic scripting using the AE knowledge bases recorded an average RTP of 51 against the SR.

**Table 5:** Evaluation of dynamic scripting in Wargus using the AE knowledge bases.

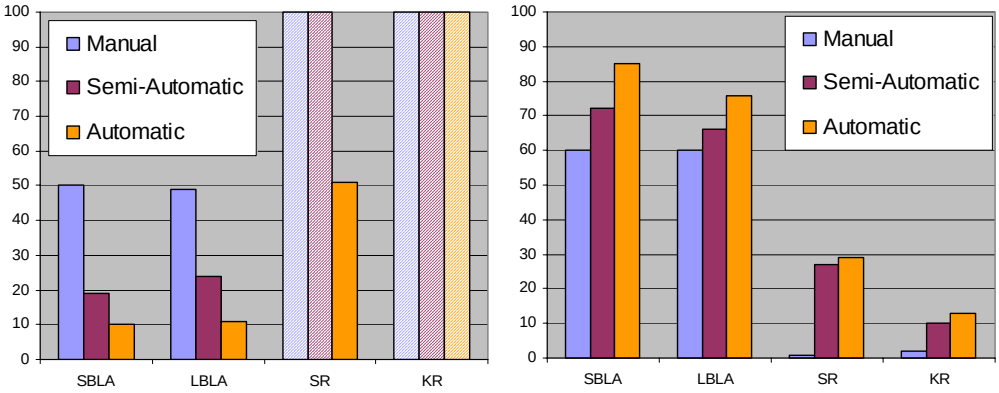
| AE knowledge bases | | | | |
|---|---|---|---|---|
| Strategy | Tests | RTP | >100 | Won |
| SBLA | 11 | 10 | 0 | 85 |
| LBLA | 11 | 11 | 0 | 76 |
| SR | 21 | 51 | 0 | 29 |
| KR | 10 | - | 10 | 13 |

## 7 Discussion

Based on a comparison of the results for the three competing approaches, illustrated in Figures 7 and 8, we may conclude that the fully automated approach obtained the best results for dynamic scripting. With the AE knowledge bases, RTP values against all strategies except KR have substantially decreased, and on average more games are won. We believe that this increased performance, compared to the other two knowledge acquisition approaches, occurred for at least three reasons.

The first reason is that the AE knowledge bases were not restricted to the (potentially poor) domain knowledge provided by the designer. We were responsible for manually encoding and manually improving the domain knowledge, and we hardly consider ourselves domain experts. In particular, we expect performance for the manual approach to increase when allowing an expert to encode the domain knowledge. However, manually encoding knowledge bases for complex domains such as Wargus is very challenging. A domain expert alone might not generate satisfying results. Being an expert in a particular domain does not imply that this person is also qualified to formalize the

**Figures 7 and 8:** Figure 7 (on the left) and Figure 8 (on the right) illustrate respectively the recorded average RTP values and the average number of games won out of a 100 for the three competing approaches (for each group of three the left represents the manual approach, the middle the semi-automatic approach and the right bar the automatic approach). The x-axis lists the opponent strategies. The y-axis in Figure 7 represents the average RTP value. A low RTP value indicates good efficiency for dynamic scripting. The five bars that reached 100 represent runs where no RTP was found (e.g., dynamic scripting was unable to statistically out-perform the specified opponent). The y-axis in Figure 8 represents the average number of games won out of a 100 by dynamic scripting.




**Average RTP** **Average Number of Games Won**

knowledge in a way that a program can successfully use it. This task is typically tackled by a knowledge engineer.

The second reason is that the AE knowledge bases include mostly tactics consisting of multiple game actions, whereas the tactics in both the ME and MI knowledge bases mostly consisted of single atomic actions. In the latter part of Section 3.2, we explained that we constrained the action space by employing a high-level control over the AI in the form of scripted game actions. Knowledge bases consisting of compound tactics (i.e., an effective combination of fine-tuned game actions) further reduce the search complexity in Wargus by providing an even higher-level control over the AI, allowing dynamic scripting to achieve relatively fast adaptation against many static opponents.

The third reason is that the automated knowledge acquisition approach is the only one that actually receives feedback from the game engine. An analogy can be made with wrapper and filter models for feature selection (Kohavi & John, 1997). Wrapper models select features by testing tentative selections on the targeted prediction algorithm (e.g., a classifier), while filter models do not (e.g., a filter model might select features according to their mutual information gain). Thus, while filter models have lower computational complexity, they employ a different bias for feature selection than the target algorithm itself. Similarly, our AE approach uses expensive game play to acquire tactics. While the manual and semi-automatic approaches have lower computational costs, there is no guarantee that their tactic selection biases match the biases of the game engine itself.

**The Issue of Generalization**

The automatic approach produced the best results with dynamic scripting. However, it is possible that the resulting knowledge bases from the AE process were tailored against specific game AI strategies (i.e., the ones received as input for the AE process). In particular, scripts 1 to 4 (SBLA, LBLA, SR, and KR) were both in the training and test set. We decided to run 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 (i.e., one starting location for a player was superior over the other starting locations). Therefore, we tested the student scripts on the small map.

Dynamic scripting using the AE knowledge bases was played against the new student scripts. The experimental parameters for dynamic scripting were unchanged. Table 6 summarizes the results against the student scripts. These results were encouraging. Only the champion script puts up a good fight; the others were already defeated from the start. We may conclude that the AE knowledge bases include generalized tactics, since dynamic scripting performs well against four independently created scripts that were not involved in the training set for the evolutionary algorithm.

**Table 6:** Evaluation of dynamic scripting using the AE knowledge bases against the student scripts.

| AE knowledge bases | | | | |
|---|---|---|---|---|
| Strategy | Tests | RTP | >100 | Won |
| SC1 | 10 | 83 | 5 | 27 |
| SC2 | 10 | 19 | 0 | 61 |
| SC3 | 10 | 12 | 0 | 84 |
| SC4 | 10 | 20 | 0 | 73 |

## 8 Conclusions and Future Work

We detailed three alternatives for acquiring high-quality domain knowledge used by adaptive game AI: manual, semi-automatic, and automatic. We first introduced our test environment Wargus, a faithful clone of the Warcraft II™ game, whose characteristics are typical of RTS games. We then discussed dynamic scripting, an adaptive game AI technique. We explained that domain knowledge is a crucial factor to the performance of dynamic scripting. We showed that, in our experiments, for the task of winning RTS games dynamic scripting's performance is best when using the automatic knowledge acquisition approach. The automatic approach requires as input a collection of pre-defined scripts. These are readily available in typical commercial games from various sources, including scripts used for testing the game engine (e.g., scripts that record human playing strategies). We also showed that the automatically generated knowledge bases included strong, generalized tactics that can perform well against many different opponent 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.

Our future work extends the discussed research in several directions. In dynamic scripting we are exploring ways not only to automatically generate knowledge bases, but also ways to automatically discover orderings and relationships between different knowledge elements. In the area of knowledge transfer, in the TIELT (2006) project, we are investigating ways to reuse previously discovered knowledge in new situations. For example, in preliminary research, Ponsen *et al.* (2006b) learned a navigation policy for a worker unit in a RTS game that generalized to unseen situations. Finally, since the ultimate goal of most games is to entertain human players, we are looking ways to create game AI that adapts to the entertainment value of game AI, rather than its effectiveness.

## Acknowledgements

The first and third author 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 US Government. The second author is funded by a grant from the Netherlands Organization for Scientific Research (NWO grant No 612.066.406).

**References**


Allen, M., H. Suliman, Z. Wen, N. Gough, and Q. Mehdi (2001). Directions for Future Game Development. In: Q. Mehdi, N. Gough, and D. Al-Dabass (eds.): *Proceedings of the Second International Conference on Intelligent Games and Simulation*. Ghent, Belgium, pp. 22–32, SCS Europe Bvba.

Blythe, J., Kim, J., Ramachandran, S., and Gil, Y. (2001). An Integrated Environment for Knowledge Acquisition. In: *Proceedings of the International Conference on Intelligent User Interfaces 2001*.

Brockington, M., and Darrah, M. (2002). How *Not* to Implement a Basic Scripting Language. *AI Game Programming Wisdom*, Charles River Media, Hingham, MA, pp. 548–554.

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

Buro, M. (2004). Call for AI Research in RTS Games. In: *Proceedings of the AAAI-04 Workshop on Challenges in Game AI*. Menlo Park, CA, pp. 139–142, AAAI press.

Chan, B., Denzinger, J., Gates, D., Loose K., and Buchanan J. (2004), Evolutionary Behavior Testing of Commercial Computer Games. In: *Proceedings of the 2004 IEEE Congress on Evolutionary Computation*, Piscataway, NJ, pp. 125-132

Cheng, D.C., and Thawonmas, R. (2004). Case-based plan recognition for real-time strategy games. *Proceedings of the Fifth Game-On International Conference*, pp. 36-40.

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

Forbus, K. and Laird, J. (2002). AI and the Entertainment Industry. *IEEE Intelligent Systems* 17(4), pp. 15–16.

Forbus, K., Mahoney, J., and Dill, K. (2001). How qualitative spatial reasoning can improve strategy game AIs. In: J. Laird & M. van Lent (Eds.) *Artificial Intelligence and Interactive Entertainment: Papers from the AAAI Spring Symposium (Technical Report SS-01-02)*. Stanford, CA: AAAI Press.

Gold, J. (2004). *Object-oriented Game Development*. Harrow, UK: Addison-Wesley

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.

Ilghami, O., Nau, D.S., Muñoz-Avila, H., and Aha, D.W. (2002). CaMeL: Learning Methods for HTN Planning. In: *Proceedings of AIPS'02.*

Kirby, N. (2003). Getting Around the Limits of Machine Learning. In: *AI Game Programming Wisdom 2*. Charles River Media.

Kohavi, R., and John, G. (1997). Wrappers for feature subset selection. *Artificial Intelligence*, 97, 273-324.

Laird, J., and Van Lent M. (2001). Human-Level's AI Killer Application: Interactive Computer Games. *Artificial Intelligence Magazine* 22(2), 15–26.

Manslow, J. (2002). Learning and Adaptation. *AI Game Programming Wisdom*, Charles River Media, Hingham, MA, pp. 557–566.

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., Muñoz-Avila, H., Spronck P., & Aha D.W (2006a). Automatically Generating Game Tactics through Evolutionary Learning. To appear in AI Magazine.

Ponsen, M., Spronck P., & Tuyls. K., (2006b). Hierarchical Reinforcement Learning with Deictic Representation in a Computer Game. To be published in the *Proceedings of the BNAIC 2006*, Namur, Belgium

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., Ponsen, M., Sprinkhuizen-Kuyper, I., and Postma, E. (2006). Adaptive Game AI with Dynamic Scripting. *Machine Learning*, Vol. 63, No. 3, pp. 217-248.

Street, G., Petersen, S., Kidd, M. (2001). How to Balance a Real Strategy Game: Lessons from the Age of Empire Series. In: *Proceedings of the2001 Game Developers Conference*.

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

TIELT (2006). Testbed for integrating and evaluating learning techniques. [http://www.tielt.org]

Tomlinson, S. L. (2003). Working at Thinking about Playing or A Year in the Life of a Games AI Programmer. In: *Proceedings of the 4th International Conference on Intelligent Games and Simulation (GAME-ON 2003)*. (eds. Q. Mehdi, N. Gough and S. Natkin), EUROSIS, Ghent, Belgium, pp. 5–12.

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

Winner, E., and Veloso, M.M. (2003). DISTILL: Learning Domain-Specific Planners by Example. In: *Proceedings of ICML-2003*. Washington, DC.