Mach Learn (2006) 63:217–248
DOI 10.1007/s10994-006-6205-6

Adaptive game AI with dynamic scripting

Pieter Spronck · Marc Ponsen · Ida Sprinkhuizen-Kuyper · Eric Postma

Received: 11 February 2005 / Revised: 21 September 2005 / Accepted: 14 November 2005 /
Published online: 9 March 2006

Abstract Online learning in commercial computer games allows computer-controlled opponents to adapt to the way the game is being played. As such it provides a mechanism to deal with weaknesses in the game AI, and to respond to changes in human player tactics. We argue that online learning of game AI should meet four computational and four functional requirements. The computational requirements are speed, effectiveness, robustness and efficiency. The functional requirements are clarity, variety, consistency and scalability. This paper investigates a novel online learning technique for game AI called 'dynamic scripting', that uses an adaptive rulebase for the generation of game AI on the fly. The performance of dynamic scripting is evaluated in experiments in which adaptive agents are pitted against a collection of manually-designed tactics in a simulated computer roleplaying game. Experimental results indicate that dynamic scripting succeeds in endowing computer-controlled opponents with adaptive performance. To further improve the dynamic-scripting technique, an enhancement is investigated that allows scaling of the difficulty level of the game AI to the human player's skill level. With the enhancement, dynamic scripting meets all computational and functional requirements. The applicability of dynamic scripting in state-of-the-art commercial games is demonstrated by implementing the technique in the game NEVERWINTER NIGHTS. We conclude that dynamic scripting can be successfully applied to the online adaptation of game AI in commercial computer games.

Editors: Michael Bowling, Johannes Fürnkranz, Thore Graepel, and Ron Musick

P. Spronck (✉) · M. Ponsen · I. Sprinkhuizen-Kuyper · E. Postma
Institute for Knowledge and Agent Technology, Universiteit Maastricht,
Maastricht, NL-6200 MD, P.O. Box 616, The Netherlands
e-mail: p.spronck@cs.unimaas.nl

M. Ponsen
e-mail: m.ponsen@cs.unimaas.nl

I. Sprinkhuizen-Kuyper
e-mail: kuyper@cs.unimaas.nl

E. Postma
e-mail: postma@cs.unimaas.nl

Keywords Computer game · Reinforcement learning · Dynamic scripting

1. Introduction

The quality of commercial computer games (henceforth called 'games') is directly related to their entertainment value (Tozour, 2002a). The general dissatisfaction of game players with the current level of the artificial intelligence of computer-controlled opponents (so-called 'game AI') makes them prefer human-controlled opponents (Schaeffer, 2001). Improving the quality of game AI (while preserving the characteristics associated with high entertainment value; Scott, 2002) is desired in case human-controlled opponents are not available.

For complex games, such as Computer Role-Playing Games (CRPGs) and strategy games, where the number of choices at each turn ranges from hundreds to even thousands, the incorporation of advanced AI is hard. To implement game AI for complex games, most developers resort to scripts, i.e., lists of rules that are executed sequentially. Scripts are the technique of choice in the game industry to implement game AI, because they are understandable, predictable, adaptable to specific circumstances, easy to implement, easily extendable, and useable by non-programmers (Tozour, 2002b).

Scripts are generally static and tend to be quite long and complex (Brockington & Darrah, 2002), which gives rise to two problems. The first problem is that, due to their complexity, scripts are likely to contain weaknesses (Nareyek, 2002), which can be exploited by human players to defeat supposedly tough opponents with ease. The second problem is that, due to their static nature, scripts cannot deal with unforeseen human-player tactics, and cannot scale the difficulty level exhibited by the game AI to cater to both novice and experienced human players. These two problems, which are common to state-of-the-art game AI (Buro, 2003; Spronck et al., 2003), hamper the entertainment value of games.

The application of online machine learning techniques to game AI has the capability to deal with both these problems. We designed a novel machine learning technique called 'dynamic scripting' that improves scripted game AI by online learning, in particular in complex games. The present paper reports on experiments performed in both a simulated and an actual commercial CRPG to assess the adaptive performance obtained with the technique.

The outline of the remainder of the paper is as follows. Section 2 discusses adaptive game AI, and the requirements it must meet to be practically applicable. Section 3 describes the dynamic-scripting technique. The experiments performed for evaluating the adaptive performance of dynamic scripting are described in Sections 4 to 6. Section 4 describes the experimental procedure, and investigates the performance of dynamic scripting in a simulated CRPG. Section 5 investigates enhancements to dynamic scripting to allow scaling of the difficulty level of the game AI to the skill level of the human player. In Section 6, the results achieved in the simulation CRPG are validated in an actual state-of-the-art CRPG. Section 7 presents our conclusions and describes future work.

2. Adaptive game AI

Adaptive game AI is game AI that has the ability to adapt successfully to changing circumstances. This section discusses the implementation of adaptive game AI by the application of online machine learning techniques. It provides a short introduction to game AI (2.1), discusses the application of online learning techniques to game AI (2.2), and provides four computational and four functional requirements adaptive game AI must meet to be applicable in practice (2.3).

2.1. Game AI

Traditionally, game-development companies competed by creating superior graphics for their games. Nowadays they attempt to compete by offering a better game-play experience (Tozour, 2002a). Game AI is an essential element of game-play, which has become an important selling point of games (Laird & Van Lent, 2001; Forbus & Laird, 2002). However, even in state-of-the-art games the game AI is, in general, of low quality (Schaeffer, 2001; Buro, 2004; Gold, 2004). Game AI can benefit from academic research into commercial games (Forbus & Laird, 2002), although this research is still in its infancy (Laird & Van Lent, 2001).

It should be noted that the term 'game AI' is used differently by game developers and academic researchers (Gold, 2004). Academic researchers restrict the use of the term 'game AI' to refer to intelligent behaviours of game characters (Allen et al., 2001). In contrast, for game developers the term '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 the term 'game AI' will be used in the narrow, academic sense.

2.2. The state of the art in adaptive game AI

Through the application of online machine-learning techniques game AI can become adaptive. There are two distinct ways of implementing online machine learning of game AI, namely (i) human-controlled, and (ii) computer-controlled.

Human-controlled online learning implements changes to game AI by processing immediate feedback given by the human player on any decision the game AI makes. The feedback indicates whether a decision is desired or undesired. Human-controlled online learning has been sporadically used in commercial games to adapt game AI as part of the game itself (Adamatzky, 2000; Evans, 2002). While these examples are considered to be among the most advanced in current game AI, for the present research they are mostly irrelevant, because we aim at having the game AI adapt automatically.

Computer-controlled online learning (henceforth called 'online learning') implements automatic changes to game AI by processing observations on the effect of the game AI during the game directly. It is widely disregarded by commercial game developers (Woodcock, 2000; Rabin, 2004a), even though it has been shown to be feasible for simple games (Demasi, 2002; Johnson, 2004). An important reason for the disinterest on the part of game developers is that game publishers are reluctant to release games with online learning capabilities. The publishers' main fear is that the game AI learns to exhibit inferior behaviour. Therefore, the few games that contain online learning, do so in a severely limited sense, adjusting only a few game parameters, to run as little risk as possible (Charles & Livingstone, 2004).

2.3. Requirements

After a literature survey, personal communication with game developers, and applying our own insights to the subject matter, we arrived at a list of four computational and four functional requirements, which adaptive game AI must meet to be applicable in practice.

The computational requirements are necessities: if adaptive game AI does not meet the computational requirements, it is useless in practice. The functional requirements are not so much necessities, as strong preferences by game developers. Failure of adaptive game AI to meet the functional requirements means that many game developers will be unwilling to

include the technique in their games, even if the technique yields good results and meets all computational requirements.

The four computational requirements are the following.

Speed: Adaptive game AI must be computationally fast, since learning takes place during game-play (Laird & Van Lent, 2001; Nareyek, 2002; Charles & Livingstone, 2004).

Effectiveness: Adaptive game AI must be effective during the whole learning process, to avoid it becoming inferior to manually-designed game AI (Charles & Livingstone, 2004). When it is effective, adaptive game AI produces reasonably successful behaviour at all times. By meeting this requirement, the main fear of game publishers, that opponents will learn inferior behaviour, is resolved.

Robustness: Adaptive game AI has to be robust with respect to the randomness inherent in most games.

Efficiency: Adaptive game AI must be efficient with respect to the number of learning opportunities needed to be successful, since in a single game, a player experiences only a limited number of encounters with similar situations.

The four functional requirements are the following.

Clarity: Adaptive game AI must produce easily interpretable results, because game developers distrust learning techniques of which the results are hard to understand.

Variety: Adaptive game AI must produce a variety of different behaviours, because agents that exhibit predictable behaviour are less entertaining than agents that exhibit unpredictable behaviour.

Consistency: The average number of learning opportunities needed for adaptive game AI to produce successful results should have a high consistency, i.e., a low variance, to ensure that their achievement is independent both from the behaviour of the human player, and from random fluctuations in the learning process.

Scalability: Adaptive game AI must be able to scale the difficulty level of its results to the skill level of the human player (Lidén, 2004).

To meet the requirements, especially the four computational ones, an online learning algorithm must be of 'high performance'. According to Michalewicz and Fogel (2000), the two main factors of importance when attempting to achieve high performance for a learning mechanism are the exclusion of randomness and the addition of domain-specific knowledge. Since randomness is inherent in most games, it cannot be excluded. Therefore, it is imperative that the learning process is based on domain-specific knowledge.

The remainder of this paper discusses the 'dynamic scripting' technique, that has been developed to meet the four computational and four functional requirements.

3. Dynamic scripting

This section describes the dynamic-scripting technique (3.1), compares dynamic scripting to reinforcement learning (3.2), presents pseudo-code for the main parts of the algorithm (3.3), and explains how it meets the computational and functional requirements for adaptive game AI (3.4).

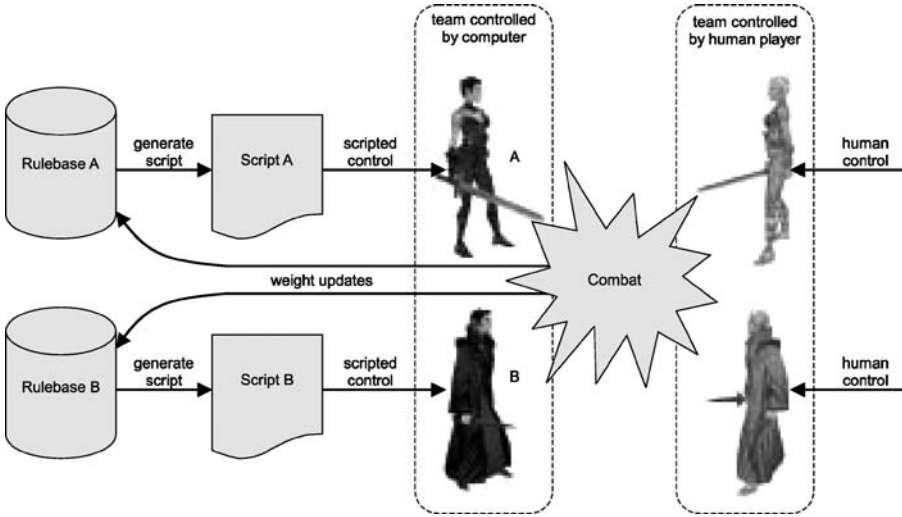

Fig. 1 Dynamic scripting.

3.1. Description of dynamic scripting

Dynamic scripting is an online competitive machine-learning technique for game AI, that can be characterised as stochastic optimisation. Dynamic scripting maintains several rulebases, one for each agent class in the game. Every time a new instance of an agent is generated, the rulebases are used to create a new script that controls the agent's behaviour. The rules that comprise a script controlling a particular agent are extracted from the rulebase associated with the agent's class. The probability that a rule is selected for a script is influenced by a weight value that is attached to each rule. The goal of dynamic scripting is to adapt the weights in the rulebases so that the expected fitness of the behaviour defined by its generated scripts is increased rapidly, even in changing environments. The fitness is commonly defined as the probability that the team to which the agent belongs wins the game. Adaptation of the rulebase proceeds by changing the weight values to reflect the success or failure rate of the corresponding rules in scripts. The weight changes are determined by a weight-update function.

The dynamic-scripting technique is illustrated in Fig. 1 in the context of a commercial game. In the figure, the team dressed in grey is controlled by a human player, while the computer controls the team dressed in black. The rulebase associated with each computer-controlled agent (named 'A' and 'B' in Fig. 1) contains manually-designed rules derived from domain-specific knowledge. It is imperative that the majority of the rules in the rulebase define effective, or at least sensible, agent behaviour.

At the start of an encounter (i.e., a fight between two opposing teams), a new script is generated for each computer-controlled agent, by randomly selecting a specific number of rules from its associated rulebase. Dynamic scripting uses a softmax selection mechanism: there is a linear relationship between the probability that a rule is selected and its associated weight. The order in which the rules are placed in the script depends on the application domain. A priority mechanism can be used to let certain rules take precedence over other rules. Such a priority mechanism is only required if a general ordering of rules and actions

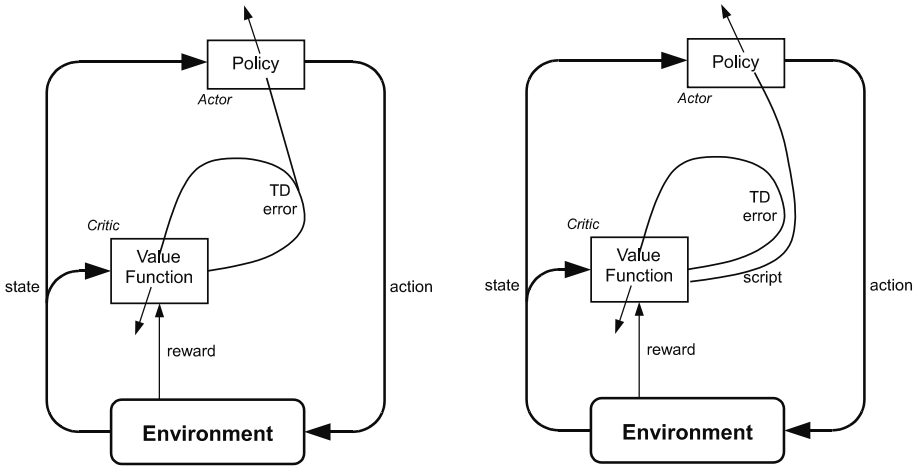

Fig. 2 Comparison between an actor-critic architecture (left; Sutton & Barto, 1998), and dynamic scripting (right).

is prescribed by the domain knowledge. More specific action groupings, such as two actions which must always be executed in a specific order, should be combined in one rule.

In the dynamic-scripting approach, learning proceeds as follows. Upon completion of an encounter (combat), the weights of the rules employed during the encounter are adapted depending on their contribution to the outcome. Rules that lead to success are rewarded with a weight increase, whereas rules that lead to failure are punished with a weight decrease. The increment or decrement of each weight is compensated for by decreasing or increasing all remaining weights as to keep the weight total constant. The redistribution of weights is a key feature of dynamic scripting, since it makes all rules in the rulebases learn at every update.

Dynamic scripting can be applied to any form of game AI that meets three requirements: (i) the game AI can be scripted, (ii) domain knowledge on the characteristics of a successful script can be collected, and (iii) an evaluation function can be designed to assess the success of the script.

3.2. Dynamic scripting and reinforcement learning

Dynamic scripting is based on reinforcement learning (Sutton & Barto, 1998). In fact, the architecture of dynamic scripting is quite similar to an actor-critic architecture, as illustrated in Fig. 2. Sutton and Barto (1998) provide good reasons tot assume that actor-critic methods work well for adaptive game AI, as they state: "[Actor-critic methods] require minimal computation in order to select actions . . . They can learn an explicitly stochastic policy; that is, they can learn the optimal probabilities of selecting various actions. This ability turns out to be useful in competitive and non-Markov cases."

Similar to actor-critic methods, dynamic scripting is an on-policy control method that maintains separate datastructures for the actor, namely the script, and for the critic, namely the rulebases with the associated weight-adjustment mechanism. In dynamic scripting, the size of the weight adjustment is determined by a fitness function that incorporates the agent's state, and a reward (or penalty) for the performance of the agent's team in the environment.

Dynamic scripting and actor-critic methods differ in two main ways: (i) dynamic scripting updates the policy by extracting rules from a rulebase, whereas actor-critic methods update the policy directly using a TD error, and (ii) dynamic scripting updates the value function at the time that the effect of a sequence of actions can be measured, whereas actor-critic methods commonly update the value function after each action.

For reinforcement learning of adaptive game AI, defining a good balance between exploitation and exploration is particularly troubling (Manslow, 2002; Madeira et al., 2004). On the one hand, adaptive game AI must start exploiting what it has learned as early as possible, to meet the effectiveness requirement. On the other hand, adaptive game AI must use as many learning opportunities as possible for exploration, to deal with the non-stationary environment and to meet the efficiency requirement. 'Standard' reinforcement learning, such as TD-learning, in general requires extensive and continuing exploration to ensure that exploitation will generate successful behaviour, and thus fails to meet both the requirement of effectiveness and the requirement of efficiency. This issue may be of less importance if the game environment allows for the generation of a large number of learning opportunities in little time, which may happen on an operational level of intelligence, as in the work by Graepel et al. (2004). However, on a tactical or strategic level of intelligence in general learning opportunities arise only sporadically.

A reinforcement-learning method which is deemed particularly suitable for learning in non-Markov cases, and which is able to quickly start exploiting learned behaviour, is Monte-Carlo control (Sutton & Barto, 1998). Dynamic scripting is compared to Monte-Carlo control in Subsection 4.10.

3.3. Dynamic scripting code

The two central procedures of the dynamic-scripting technique are script generation and weight adjustment, which are specified in pseudo-code in this subsection. In the code, the rulebase is represented by an array of *rule* objects. Each *rule* object has three attributes, namely (i) *weight*, which stores the rule's weight as an integer value, (ii) *line*, which stores the rule's actual text to add to the script when the rule is selected, and (iii) *activated*, which is a boolean variable that indicates whether the rule was activated during script execution.

Algorithm 1 presents the script generation procedure. In the algorithm, the function 'InsertInScript' adds a line to the script. If the line is already in the script, the function has no effect and returns 'false'. Otherwise, the line is inserted and the function returns 'true'. The algorithm aims to put *scriptsize* lines in the script, but may end up with less lines if it needs more than *maxtries* trials to find a new line. The function 'FinishScript' appends one or more generally-applicable lines to the script, to ensure that the script will always find an action to execute.

Algorithm 2 presents the weight adjustment algorithm. The function 'CalculateAdjustment' calculates the reward or penalty each of the activated rules receives. The parameter *Fitness* is a measure of the performance of the script during the encounter. The function 'DistributeRemainder' distributes the difference between the current weight total and the original weight total over all weights. Commonly it will be implemented as a loop over all weights, awarding a small fraction of the remainder to each weight if that does not cause the weight to exceed the weight boundaries, until the remainder is zero. When many of the weights in the rulebases approach the weight boundaries, this can be a highly time-consuming process that seriously interrupts gameplay. As a solution, part of the remainder can be carried over to the next weight adjustment call.

Algorithm 1 Script Generation

```
 1: ClearScript()
 2: sumweights = 0
 3: for i = 0 to rulecount−1 do
 4:    sumweights = sumweights + rule[i].weight
 5: end for
 6: {Repeated roulette wheel selection}
 7: for i = 0 to scriptsize−1 do
 8:    try = 0; lineadded = false
 9:    while try < maxtries and not lineadded do
10:       j = 0; sum = 0; selected = −1
11:       fraction = random(sumweights)
12:       while selected < 0 do
13:          sum = sum + rule[j].weight
14:          if sum > fraction then
15:             selected = j
16:          else
17:             j = j + 1
18:          end if
19:       end while
20:       lineadded = InsertInScript(rule[selected].line)
21:       try = try + 1
22:    end while
23: end for
24: Finish Script()
```

It should be noted that in Algorithm 1 the calculation of *sumweight* in lines 3 to 5 should always lead to the same result, namely the sum of all the initial rule weights. However, the short calculation that is used to determine the value of *sumweight* ensures that the algorithm will succeed even if Algorithm 2 does not distribute the value of *remainder* completely.

3.4. Dynamic scripting and learning requirements

Dynamic scripting meets five of the eight computational and functional requirements by design, as follows.

- *Speed* (computational): Dynamic scripting is computationally fast, because it only requires the extraction of rules from a rulebase and the updating of weights once per encounter.
- *Effectiveness* (computational): The effectiveness of dynamic scripting is ensured when all rules in the rulebase are sensible, based on correct domain knowledge. Every action which an agent executes through a script that contains these rules, is an action that is at least reasonably effective (although it may be inappropriate for certain situations). Note that if the game developers make a mistake and include an inferior rule in the rulebase, the dynamic-scripting technique will quickly assign this rule a low weight value. Therefore, the requirement of effectiveness is met even if the rulebase contains a few inferior rules. Obviously, if most rules in the rulebase are inferior, dynamic scripting will not be able to generate adequate game AI.
- *Robustness* (computational): Dynamic scripting is robust because the weight of a rule in a rulebase represents a statistical utility, derived from multiple samples, of the expected fitness of a script that contains the rule. An unjustified penalty will not remove a rule

Algorithm 2 Weight Adjustment

```
1:  active = 0
2:  for i = 0 to rulecount−1 do
3:    if rule[i].activated then
4:      active = active+1
5:    end if
6:  end for
7:  if active < = 0 or active > = rulecount then
8:    return (no updates are needed.)
9:  end if
10: nonactive = rulecount−active
11: adjustment = CalculateAdjustment(Fitness)
12: compensation = −active ∗ adjustment/nonactive
13: remainder = 0
14: Credit assignment
15: for i = 0 to rulecount−1 do
16:   if rule[i].activated then
17:     rule[i].weight = rule[i].weight + adjustment
18:   else
19:     rule[i].weight = rule[i].weight + compensation
20:   end if
21:   if rule[i].weight < minweight then
22:     remainder = remainder + (rule[i].weight − minweight)
23:     rule[i].weight = minweight
24:   else if rule[i].weight > maxweight then
25:     remainder = remainder+(rule[i].weight−maxweight)
26:     rule[i].weight = maxweight
27:   end if
28: end for
29: DistributeRemainder();
```

from a rulebase, and will be compensated when the rule gets selected again, or even when other rules get penalised. Similarly, an unjustified reward may cause an inferior rule to be selected more often, which only means it will collect justified penalties faster.

- *Clarity* (functional): Dynamic scripting generates scripts, which can be easily understood by game developers.
- *Variety* (functional): Dynamic scripting generates a new script for every agent, and thus provides variety in behaviour.

The remaining three requirements, namely the computational requirement of efficiency and the functional requirements of consistency and scalability, are not met by design. The dynamic-scripting technique is believed to meet the requirements of efficiency and consistency, because with appropriate weight-updating parameters it can adapt after a few encounters only against arbitrary opponents. This is investigated empirically in Section 4. Enhancements to the dynamic-scripting technique that make it meet the requirement of scalability are investigated in Section 5.

4. Performance evaluation

Since the dynamic-scripting technique is designed to be used against human players, ideally an empirical evaluation of the technique is derived from an analysis of games it plays against

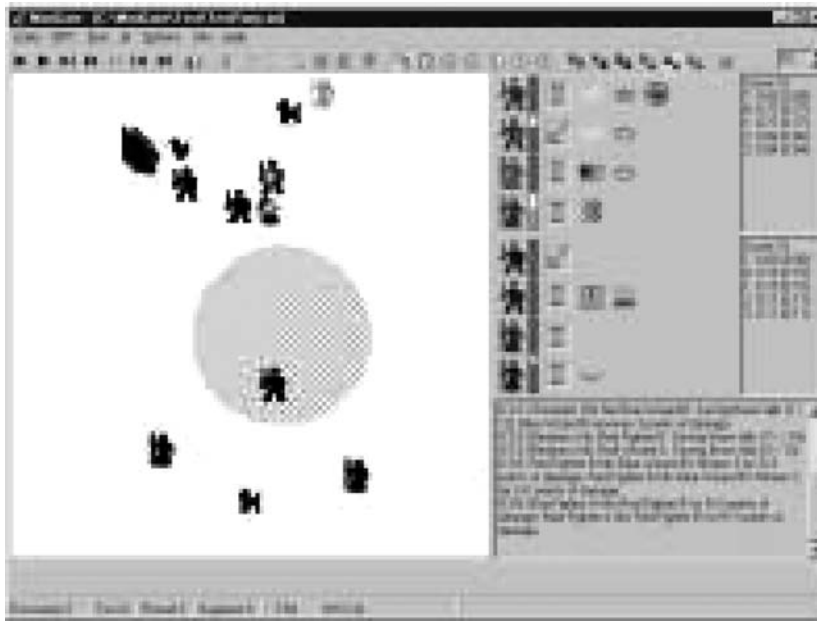

Fig. 3 The CRPG simulation.

humans. However, due to the large number of tests that must be performed, such an evaluation is not feasible within a reasonable amount of time. Therefore, we decided to evaluate the dynamic-scripting technique by its ability to discover scripts capable of defeating strong, but static, tactics. Translated to a game played against human players, the evaluation tests the ability of dynamic scripting to force the human player to seek continuously new tactics, because the game AI will automatically adapt to deal with tactics that are used often.

We performed the evaluation in a simulated CRPG. This section describes the simulation environment (4.1), the scripts and rulebases (4.2), the weight-update function (4.3), the tactics that dynamic scripting is tested against (4.4), the history-fallback mechanism that was incorporated (4.5), and the measures used to evaluate the results (4.6). Four series of experimental results are described, namely a determination of the baseline performance (4.7), the performance of dynamic scripting (4.8), the performance of dynamic scripting using biased rulebases (4.9), and a comparison with Monte-Carlo control (4.10). The section ends with a discussion (4.11).

4.1. Simulation environment

The CRPG simulation used to evaluate dynamic scripting is illustrated in Fig. 3. It is modelled closely after the popular BALDUR'S GATE games. These games are representative for the most complex and extensive game-play systems found in modern CRPGs, closely resembling classic non-computer roleplaying games.

Our simulation entails an encounter between two teams of similar composition. The 'dynamic team' is controlled by dynamic scripting. The 'static team' is controlled by unchanging scripts, that represent strong tactics. Each team consists of four agents, namely two 'fighters' and two 'wizards' of equal 'experience level'. The armament and weaponry of the

teams is static, and each agent is allowed to select two (out of three possible) magic potions. In addition, the wizards are allowed to memorise seven (out of 21 possible) magic spells. The spells incorporated in the simulation are of varying types, amongst which damaging spells, blessings, curses, charms, area-effect spells and summoning spells. As is common in CRPGs, the results of attempted actions are highly non-deterministic.

The simulation is implemented with hard constraints and soft constraints. Hard constraints are constraints that are submitted by the games rules, e.g., a hard constraint on spells is that they can only be used when they are memorised, and a hard constraint on agents is that they can only execute an action when they are not incapacitated. Soft constraints are constraints that follow as logical consequences from the rules, e.g., a soft constraint on a healing potion is that only an agent that has been damaged should drink it. Both hard and soft constraints are taken into account when a script is executed, e.g., agents will not drink a healing potion when they are incapacitated or undamaged.

In the simulation, the practical issue of choosing spells and potions for agents is solved by making the choice depend on the (generated) scripts, as follows. Before the encounter starts, the scripts are scanned to find rules containing actions that refer to drinking potions or casting spells. When such a rule is found, a potion or spell that can be used by that action is selected. If the agent controlled by the script is allowed to possess the potion or spell, it is added to the agent's inventory.

During an encounter, agents act in real time and in parallel. An encounter is divided into rounds. At the start of each round, each agent chooses an action by executing its script. All selected actions are then executed during the round, their order being determined by the speed of each selected action (e.g., stabbing with a dagger takes less time than slashing with a sword). After an action has been executed, an agent is still allowed to move until the next round starts, and as such is allowed to select move actions from its script, but cannot select any other actions.

4.2. Scripts and rulebases

The scripting language was designed to emulate the power and versatility of the scripts used in the BALDUR'S GATE games. Rules in the scripts are executed in sequential order. For each rule the condition (if present) is checked. If the condition is fulfilled (or absent), the action is executed if it obeys all relevant hard and soft constraints. If no action is selected when the final rule is checked, the default action 'pass' is used. Three examples of typical rules are (i) 'if there is an enemy standing close, attack this enemy with a melee weapon', (ii) 'if my health is below 50%, try to heal myself', and (iii) 'if it is the very first round of a fight, throw a high-level damaging area-effect spell in the middle of the enemy team.' A complete description of the rulebases is provided by Spronck (2005).

When dynamic scripting generates a new script, the rule order in the script is determined by a manually-assigned priority value. Rules with a higher priority take precedence over rules with a lower priority. In case of equal priority, the rules with higher weights take precedence. For rules with equal priorities and equal weights, the order is determined randomly. In the example rules above, the first rule should have a lower priority than the second and third rule, since the latter two need to be executed in more specific circumstances.

The selection of script sizes was motivated by the following two considerations, namely that (i) a fighter has less action choices than a wizard, thus a fighter's script can be shorter than a wizard's script, and (ii) a typical fight will last five to ten rounds, thus a maximum of ten rules in a script seems sufficient. Therefore, the size of the script for a fighter was set to five rules, which were selected out of a rulebase containing twenty rules. For a wizard, the

script size was set to ten rules, which were selected out of a rulebase containing fifty rules. At the end of each script, default rules were attached, to ensure the execution of an action in case none of the rules extracted from the rulebase could be activated.

4.3. Weight-update function

The weight-update function is based on two fitness functions, namely (i) a team-fitness function $F(g)$ (where g refers to the team), and (ii) an agent-fitness function $F(a, g)$ (where a refers to the agent, and g refers to the team to which the agent belongs). Both fitness functions yield a value in the range [0,1]. The fitness values are calculated at time $t = T$, where T is the time step at which all agents in one of the teams are 'defeated', i.e., have their health reduced to zero or less. A team of which all agents are defeated, has lost the fight. A team that has at least one agent 'surviving', has won the fight. At rare occasions both teams may lose at the same time.

The team-fitness function is defined as follows.

$$F(g) = \sum_{c \in g} \begin{cases} 0 & \{g \text{ lost}\} \\ \dfrac{1}{2N_g}\left(1 + \dfrac{h_T(c)}{h_0(c)}\right) & \{g \text{ won}\} \end{cases} \tag{1}$$

In Eq. 1, g refers to a team, c refers to an agent, $N_g \in \mathbb{N}$ is the total number of agents in team g, and $h_t(c) \in \mathbb{N}$ is the health of agent c at time t. According to the equation, a 'losing' team has a fitness of zero, while a 'winning' team has a fitness > 0.5.

The agent-fitness function is defined as follows.

$$F(a, g) = \frac{1}{10}\Big(3F(g) + 3A(a) + 2B(g) + 2C(g)\Big) \tag{2}$$

In Eq. 2, a refers to the agent whose fitness is calculated, and g refers to the team to which agent a belongs. The equation contains four components, namely (i) $F(g)$, the fitness of team g, derived from Eq. 1, (ii) $A(a) \in [0, 1]$, which is a rating of the survival capability of agent a, (iii) $B(g) \in [0, 1]$, which is a measure of health of all agents in team g, and (iv) $C(g) \in [0, 1]$, which is a measure of damage done to all agents in the team opposing g. The weight of the contribution of each of the four components to the final outcome was determined arbitrarily, taking into account the consideration that agents should give high rewards to a team victory, and to their own survival (expressed by the components $F(g)$ and $A(a)$, respectively). The function assigns smaller rewards to the survival of the agent's comrades, and to the damage inflicted upon the opposing team (expressed by the components $B(g)$ and $C(g)$, respectively). As such the agent-fitness function is a good measure of the success rate of the script that controls the agent.

The components $A(a)$, $B(g)$, and $C(g)$ are defined as follows.

$$A(a) = \frac{1}{3}\begin{cases} \min\left(\dfrac{D(a)}{D_{\max}}, 1\right) & \{h_T(a) \leq 0\} \\ 2 + \dfrac{h_T(a)}{h_0(a)} & \{h_T(a) > 0\} \end{cases} \tag{3}$$

$$B(g) = \frac{1}{2N_g} \sum_{c \in g} \begin{cases} 0 & \{h_T(c) \leq 0\} \\ 1 + \dfrac{h_T(c)}{h_0(c)} & \{h_T(c) > 0\} \end{cases} \tag{4}$$

$$C(g) = \frac{1}{2N_{\neg g}} \sum_{c \notin g} \begin{cases} 1 & \{h_T(c) \leq 0\} \\ 1 - \dfrac{h_T(c)}{h_0(c)} & \{h_T(c) > 0\} \end{cases} \tag{5}$$

In Eqs. 3 to 5, a and g are as in Eq. 2, c, N_g and $h_t(c)$ are as in Eq. 1, $N_{\neg g} \in \mathbb{N}$ is the total number of agents in the team that opposes g, $D(a) \in \mathbb{N}$ is the time of 'death' of agent a, and $D_{\max}$ is a constant ($D_{\max}$ was set to 100 in our experiments, which equals ten combat rounds, which is longer than many fights last).

The agent fitness is translated into weight adaptations for the rules in the script. Weight values are bounded by a range $[W_{\min}, W_{\max}]$. Only the rules in the script that are actually executed during an encounter are rewarded or penalised. The new weight value is calculated as $W + \Delta W$, where W is the original weight value, and the weight adjustment ΔW is expressed by the following formula (which is an implementation of the 'CalculateAdjustment' function from Algorithm 2):

$$\Delta W = \begin{cases} -\left\lfloor P_{\max} \dfrac{b - F}{b} \right\rfloor & \{F < b\} \\ \left\lfloor R_{\max} \dfrac{F - b}{1 - b} \right\rfloor & \{F \geq b\} \end{cases} \tag{6}$$

In Eq. 6, $R_{\max} \in \mathbb{N}$ and $P_{\max} \in \mathbb{N}$ are the maximum reward and maximum penalty respectively, F is the agent fitness, and $b \in \langle 0, 1 \rangle$ is the break-even value. At the break-even point the weights remain unchanged. 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. We round down the weight changes because we use integer math to gain computational speed.

In the performance-validation experiment, values for the constants were set as follows. The break-even value b was set to 0.3, since in the simulation this value is between the fitness value that the 'best losing agent' achieves and the fitness value that the 'worst winning agent' achieves (about 0.2 and 0.4, respectively).

The initialisation of the rulebases assigned all weights the same weight value $W_{\text{init}} = 100$. $W_{\min}$ was set to zero to allow rules that are punished a lot to be effectively removed from the script-generation process. The value of $W_{\max}$ is of particular importance as it controls the trade-off between exploitation and exploration. A high value for $W_{\max}$ stimulates exploitation when successful behaviour has been learned, while a low value for $W_{\max}$ will lead to a higher variety of tactics, thus stimulating exploration (this will be further discussed in Section 5). In the experiment, $W_{\max}$ was set to 2000, which allows weights to grow more-or-less unrestricted.

$R_{\max}$ was set to 100 to increase the efficiency of dynamic scripting by allowing large weight increases for agents with a high fitness. $P_{\max}$ was set to 70, which we determined in preliminary experiments to be a suitable value next to $R_{\max} = 100$. Note that if $P_{\max}$ is

chosen too small, the learning mechanism will have a hard time recovering from premature convergence, which will have a negative impact on the performance of dynamic scripting (Spronck, 2005).

4.4. Tactics

We defined four different basic tactics and three composite tactics for the static team. The four basic tactics, implemented as a static script for each agent of the static team, are as follows.

Offensive: The fighters always attack the nearest enemy with a melee weapon, while the wizards use the most damaging spells at the most susceptible enemies.

Disabling: The fighters start by drinking a potion that frees them of any disabling effect, then attack the nearest enemy with a melee weapon. The wizards use all kinds of spells that disable enemies for a few rounds.

Cursing: The fighters always attack the nearest enemy with a melee weapon, while the wizards use all kinds of spells that reduce the enemies' effectiveness, e.g., they try to charm enemies, physically weaken enemy fighters, deafen enemy wizards, and summon minions in the middle of the enemy team.

Defensive: The fighters start by drinking a potion that reduces fire damage, after which they attack the closest enemy with a melee weapon. The wizards use all kinds of defensive spells, to deflect harm from themselves and from their comrades, including the summoning of minions.

To assess the ability of the dynamic-scripting technique to cope with sudden changes in tactics, we defined the following three composite tactics.

Random team: Each encounter one of the four basic tactics is selected randomly.

Random agent: Each encounter each agent randomly selects one of the four basic tactics, independent from the choices of his comrades.

Consecutive: The static team starts by using one of the four basic tactics. Each encounter the team will continue to use the tactic used during the previous encounter if that encounter was won, but will switch to the next tactic if that encounter was lost. This strategy is closest to what many human players do: they stick with a tactic as long as it works, and switch when it fails. This design makes the consecutive tactic the most difficult tactic to learn a counter-tactic against.

4.5. History fallback

In the description of dynamic scripting, the old weights of the rules in the rulebase are erased when the rulebase adapts. Because of the non-determinism that pervades the CRPG simulation, it is possible that a set of well-functioning weights is replaced by inferior weights due to chance. To counteract this risk, we implemented a simple history-fallback mechanism, in which the last 15 rulebases are retained. When learning seems to be stuck in a fairly long sequence of rulebases that have inferior performance, it can 'fall back' to one of the historic rulebases that seemed to perform better. The mechanism is described in detail by Spronck (2005). The details are not provided here, because of two reasons: (i) in the experiments described, history fallback was activated very rarely, and (ii) it was shown empirically that history fallback did not influence the final results (Spronck, 2005).

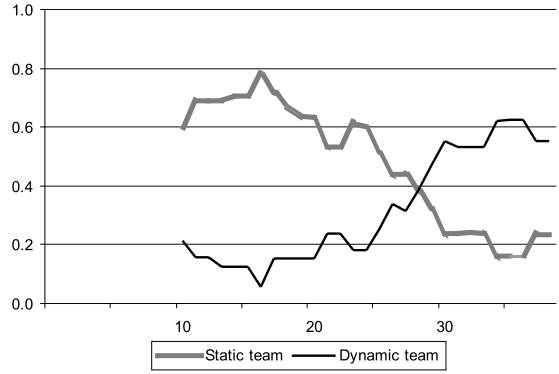

Fig. 4 Average fitness in size-10 window progression.

4.6. Measuring performance

In order to identify reliable changes in strength between teams, we define the notion of the 'turning point' (TP) as follows. After each encounter the average fitness for each of the teams over the last ten encounters is calculated. The dynamic team is said to 'outperform' the static team at an encounter, if the average fitness over the last ten encounters is higher for the dynamic team than for the static team. The turning point is the number of the first encounter after which the dynamic team outperforms the static team for at least ten consecutive encounters.

Figure 4 illustrates the turning point with a graph displaying the progression of the average team-fitness in a size-10 window (i.e., the values for the average team fitness for ten consecutive encounters) for both teams, in a typical test. The horizontal axis represents the encounters. Because of the size-10 window, the first values are displayed for encounter number 10. In this example, starting from encounter number 29 the dynamic team outperforms the static team, and maintains its superior performance for ten encounters. Therefore, the turning point is 29. Note that the lowest possible turning-point value is 10, since it must represent an average over ten previous encounters.

The absolute fitness values for the typical test are displayed in Fig. 5. Since after each encounter the fitness for one of the teams is zero, only the fitness for the winning team

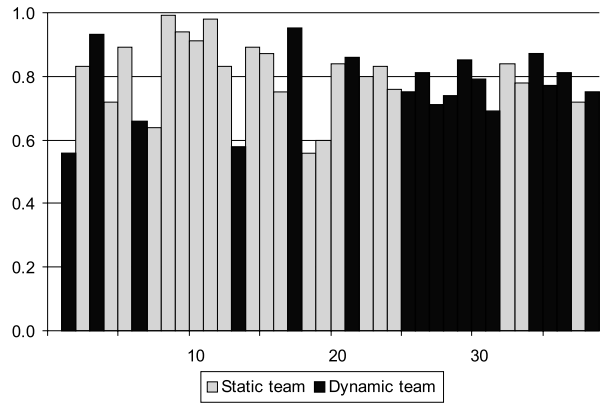

Fig. 5 Absolute fitness $F(g)$ as a function of the encounter number.

Table 1 Baseline performance against seven different tactics.

Tactic	TP > 241	Avg.TP	St.dev.	Med.TP	Wins	St.dev.
Offensive	100%	> 241	> 0.0	> 241	21	4.1
Disabling	0%	15	7.2	10	67	5.3
Cursing	77%	> 217	> 62.9	> 241	32	4.3
Defensive	33%	> 134	> 89.3	99	36	5.0
Random team	23%	> 125	> 81.0	98	39	4.7
Random agent	7%	> 87	> 64.5	60	42	5.6
Consecutive	53%	> 191	> 65.9	> 241	33	4.7

is displayed per encounter (the colour of the bar indicates which is the winning team). Evidently, after encounter 25, the dynamic team wins more often than the static team.

A low value for the turning point indicates good efficiency of dynamic scripting, since it indicates that the dynamic team consistently outperforms the static team within a few encounters only.

4.7. Baseline performance

Because of randomness inherent in the CRPG simulation, the best team does not necessarily win. Therefore, it is quite possible that the dynamic team reaches a turning point before it is actually superior to the static team. This is especially true, since dynamic scripting will generate effective scripts at all times, by means of using rulebases with predominantly effective rules. To be able to show that dynamic scripting does indeed improve the generation of successful scripts, we determined the baseline performance of a non-learning version of dynamic scripting that uses rulebases with all weights equal. The results of this determination are listed in Table 1.

We tested the performance of a non-learning dynamic team against each of the static tactics. The first column of Table 1 lists the name of the static tactic. Each test consisted of two parts.

In the first part, we let the non-learning dynamic team fight the static team until a turning point was reached, or until a run of 250 encounters had been processed. We repeated this 30 times for each static tactic. We noted the percentage of runs that did not reach the turning point before encounter number 250, which is listed in the second column of Table 1. We also determined the average of the turning points reached, whereby we set the turning point of those runs that did not reach a turning point to 241 (which is the last turning point value that could have been registered). The average is listed in the third column of the table, and the corresponding standard deviation in the fourth column. The median value is listed in the fifth column.

In the second part, we let the non-learning dynamic team fight the static team for 100 encounters exactly, and noted the number of wins of the dynamic team. We repeated this 100 times for each static tactic. The average number of wins out of 100 encounters is listed in the sixth column of Table 1, and the corresponding standard deviation in the seventh column.

From the baseline performance, we derive the following two observations.

First, the offensive tactic seems to be the hardest to defeat. Note that that does not mean that the offensive tactic is the hardest to learn a counter-tactic against. A rather specific counter-tactic is needed to defeat the offensive tactic, which is hard to produce by randomly

Table 2 Turning points against seven different tactics, averaged over 100 tests.

Tactic	Average	St.dev.	Median	Highest	Top 5
Offensive	53	24.8	52	120	107
Disabling	13	8.4	10	79	39
Cursing	44	50.4	26	304	222
Defensive	24	15.3	17	79	67
Random team	51	64.5	29	480	271
Random agent	41	40.7	25	251	178
Consecutive	52	56.2	37	393	238

selecting rules from a rulebase, but which a successful learning mechanism should be able to discover quickly.

Second, the disabling tactic is rather easy to defeat. Note that that does not mean that the disabling tactic is a bad tactic per se. It is, however, an indication that there are many counter-tactics possible against the disabling tactic, which are well-supported by the rulebases.

4.8. Performance validation results

To validate the performance of dynamic scripting, for each of the tactics we ran 100 tests to determine the average turning point. The results of these tests are presented in Table 2. The columns of the table represent, from left to right, (i) the name of the tactic, (ii) the average turning point, (iii) the corresponding standard deviation, (iv) the median turning point, (v) the highest value for a turning point found, and (vi) the average of the five highest turning points.

The aim of the first experiment was to test the viability, efficiency, and consistency of dynamic scripting. A comparison with the baseline performance presented in Table 1 (which is visualised in Fig. 6) makes it clear that dynamic scripting provided a dramatic improvement in the speed by which turning points were reached, which confirms the learning ability of dynamic scripting. The achieved results presented in Table 2 show that dynamic scripting is both a viable technique, and a highly efficient technique (at least in the present domain of combat in CRPGs). For all tactics, dynamic scripting yields low turning points.

While the results show that dynamic scripting is efficient, there is still the question whether it is sufficiently efficient. There is also the question whether it is sufficiently consistent. The results presented in Table 2 show that there are rare occurrences of fairly high turning points (outliers), which indicate that consistency should be improved. Such high turning points are often caused by the learning mechanism having difficulties recovering from premature convergence to a false optimum, which was reached through a sequence of chance runs where superior rules were punished or inferior rules were rewarded. One should realise, however, that the efficiency and consistency of the dynamic scripting process will be much improved if the weights in the rulebases are biased to give the better rules a higher chance of being selected at the start of the process, which is what game developers will do when incorporating dynamic scripting in a game.

4.9. Performance with biased rulebases

To demonstrate the effect of biased rulebases with dynamic scripting, we performed an extra experiment. We initialised the weight value of rules that often performed well against each

Table 3 Turning points with biased rulebases, averaged over 100 tests.

Tactic	Average	St.dev.	Median	Highest	Top 5
Offensive	17	8.6	15	71	42
Disabling	10	0.0	10	10	10
Cursing	13	5.3	10	40	30
Defensive	10	1.0	10	18	13
Random team	12	5.6	10	38	32
Random agent	13	5.8	10	44	31
Consecutive	12	4.6	10	28	26

of the tactics to 500, the weight values of rules that often performed well against some of the tactics to 300, and the weight values of rules that sometimes performed well against some of the tactics to 200. We then subtracted a constant value C from each of the weights so that the weight total was the same as before the additions ($C = 50$ for the fighter rulebase, and $C = 70$ for the wizard rulebase). We repeated the performance-validation experiment using the biased rulebases. The average results are displayed in Table 3. They clearly demonstrate the considerable increase in efficiency and consistency achieved with the biased rulebases. Therefore, results achieved with unbiased rulebases can be considered (very) conservative estimates of what dynamic scripting can achieve.

However, as can be observed in Table 3, in very rare circumstances still an outlier may occur even with biased rulebases, albeit a fairly low outlier. We argue that these remaining outliers are not a problem. Our argument is that, because dynamic scripting is a non-deterministic technique that works in a non-deterministic environment, outliers can never be prevented completely. However, entertainment value of a game is guaranteed even if an outlier occurs, as long as the requirement of effectiveness is satisfied.

4.10. Comparison with Monte-Carlo control

Dynamic scripting is a reinforcement-learning technique, in which states are encoded in the conditions of the rules in the rulebase. It is the responsibility of the game developers to ensure that the condition of a rule is defined in such a way that the corresponding action part is only executed when the agent is in a suitable state. If the game developers fail to do that correctly, rules which do not work well in all included states will probably receive low weights. Dynamic scripting will still work, but incorrectly implemented rules are at best useless, and at worst reduce the efficiency of the learning process. Most reinforcement-learning techniques do not have this problem, because the learning mechanism is able to creatively couple actions to states. Therefore, the question is warranted how the performance of regular reinforcement-learning techniques compares to the performance of dynamic scripting.

As is common in most games, an agent in the CRPG simulation cannot fully observe the behaviour of other agents in the system. Furthermore, state transitions depend on the actions of all agents in the simulation, and the agents are not fully aware of each other's actions. Therefore, states in the CRPG simulation do not satisfy the Markov property. As such, many reinforcement-learning techniques, such as Q-learning, cannot be expected to function well in this environment. We empirically confirmed that in the CRPG simulation a simple one-step implementation of Q-learning needed thousands of encounters to even begin to learn effective behaviour.

However, Sutton and Barto (1998) consider Monte-Carlo control particularly suitable for learning in non-Markov cases, and able to quickly start exploiting learned behaviour. Therefore, we decided to compare dynamic scripting to Monte-Carlo control. We implemented on-policy Monte-Carlo control (Sutton & Barto, 1998, Subsection 5.4) as follows.

We decided to distinguish states by the number of agents in the dynamic team still alive, and the number of agents in the static team still alive. Since each team starts with four agents, and since an encounter terminates when at least one team is reduced to zero agents, there are sixteen possible states. Clearly, states wherein the number of agents in the dynamic team exceeds the number of agents in the static team are desirable for the dynamic team.

In each state the possible actions were defined as the rules from the rulebases discussed in Subsection 4.2. Rules of which the conditions stated that the rule should only be executed in the first round of an encounter, were changed by removing the condition, since the initial state can be recognised as the one with four agents in each team. To give the algorithm a speed-up advantage over dynamic scripting, all rules that defined random behaviour, such as a rule that specified drinking a random potion, were removed entirely. Such rules generally define untrustworthy behaviour, and dynamic scripting almost always assigns them low weights. After these changes, the fighter had 13 possible actions per state, and the wizards had 37 possible actions per state. The algorithm was given a further advantage, by allowing agents to select potions and spells on the fly, instead of at the start of an encounter. For example, if a wizard wants to execute an action that requires a certain spell, the wizard is allowed to use that spell as long as he has not used up all his spells of that particular spell level.

As with dynamic scripting, rewards were calculated at the end of each encounter, as the agent-fitness function defined in Subsection 4.3. Therefore, the Q-value for each state-action pair is the average over all fitness values calculated for each time the state-action pair was executed, which is a value in the range [0, 1]. A greedy mechanism was used to select actions, but for exploration purposes every time an action was to be selected for a state, with an exploration probability of 0.05 a random action was selected instead of the action with the highest Q-value. This amounts to about one random action per agent per encounter. At the start of each test, Q-values were initialised to a random value in the range [0.2, 0.4], which is the break-even value ± 0.1.

We performed an experiment consisting of 50 tests of Monte-Carlo control against each of the seven static tactics. Each test ended when a turning point was reached, as specified in Subsection 4.6. The results of this experiment are listed in Table 4. The columns of the table represent, from left to right, (i) the name of the tactic, (ii) the average turning point, (iii) the corresponding standard deviation, (iv) the median turning point, and (v) the highest value for a turning point found.

Table 4 Turning points with Monte-Carlo control, averaged over 50 tests.

Tactic	Average	St.dev.	Median	Highest
Offensive	97	75.1	76	367
Disabling	13	6.3	10	51
Cursing	114	126.6	71	679
Defensive	55	39.6	40	146
Random team	122	98.4	97	540
Random agent	84	64.1	65	289
Consecutive	189	150.7	141	643

From Table 4, we derive the following two observations.

First, a comparison with Table 2 (which is visualised in Fig. 6) shows that in all cases the turning points achieved with dynamic scripting are much lower than those achieved with Monte-Carlo control, despite the advantages that Monte-Carlo control received. We did some tests with different exploration probabilities for Monte-Carlo control in the range [0.0, 0.1], but these did not significantly influence the results.

Second, Monte-Carlo control has problems in particular with the 'consecutive' tactic. This can be explained by the fact that the 'consecutive' tactic needs a counter-tactic that can deal with each of the four basic tactics. As soon as Monte-Carlo control has learned a counter-tactic against one basic tactic, the 'consecutive' tactic switches to another basic tactic. At that point, the Q-values have been calculated against the first tactic, and need to be re-tuned against the second tactic. Since the Q-values are calculated as averages, re-tuning slows down with the number of encounters that have been executed. Dynamic scripting always learns at the same pace, and is therefore better suited to deal with the 'consecutive' tactic. Since human players generally will not stick to one particular tactic, this is a crucial advantage of dynamic scripting over Monte-Carlo control.

Even though in the present experiments dynamic scripting outperforms Monte-Carlo control convincingly, two advantages of Monte-Carlo control should be mentioned: (i) Monte-Carlo control can discover effective state-action pairs which a game developer might not have foreseen and thus failed to include in the dynamic-scripting rulebases, and (ii) Monte-Carlo control does not need the priority mechanism that we have implemented for dynamic scripting to effectuate rule ordering. In future research, we will investigate whether it is possible to implement a hybrid of Monte-Carlo control and dynamic scripting, that combines the advantages of both techniques.

4.11. Discussion

Figure 6 presents a visual comparison of the average turning points reached against each of the seven static tactics, for (from left to right) (i) the baseline performance (from Table 1), (ii) dynamic scripting (from Table 2), (iii) dynamic scripting with biased rulebases (from Table 3), and (iv) Monte-Carlo control (from Table 4). It should be noted that for the baseline performance, the average turning points are lower bounds, and the actual average turning points are likely to be much higher. From the figure it is clear that dynamic scripting learns strong tactics, and does so very efficiently.

With the rulebases we used, we observed a wide variety of behaviour generated by dynamic scripting. For each of the static tactics, several different counter-tactics were generated, some of them quite surprising. For instance, one of the counter-tactics generated against the 'consecutive' tactic relied on one of the wizards of the dynamic team sacrificing himself after the first round, by summoning cannon fodder while neglecting his defences. The wizard usually died from the static team's actions in the first round, but after that the summoned creatures distracted the enemy long enough for the remaining three agents in the dynamic team to launch a decisive attack.

We acknowledge that the maximum playing strength game AI can achieve using dynamic scripting depends on the quality of the domain knowledge used to create the rules in the rulebase. While it can be considered the task of the game developer to provide high-quality domain knowledge, such domain knowledge can also be produced by evolutionary learning techniques, through self-play or through play against superior static tactics (Ponsen & Spronck, 2004). It is even possible to use evolutionary learning to generate complete, high-quality rulebases automatically (Ponsen et al., 2005). However, it is not possible to use

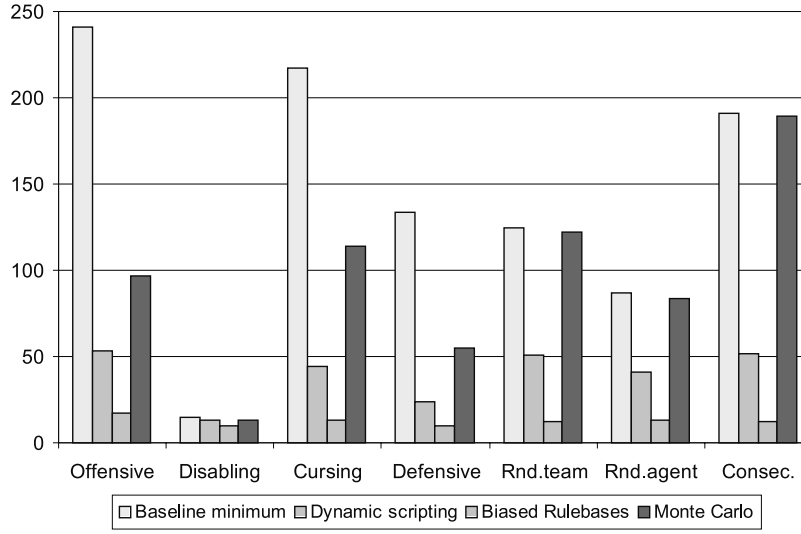

Fig. 6 Comparison of turning points achieved with different techniques.

evolutionary learning during the dynamic scripting process itself, since the generation of many rules of inferior quality would negatively influence the efficiency of the process.

The efficiency of dynamic scripting is very much dependent on the fitness function used to calculate rewards and penalties. In a small set of experiments we performed in a different environment, we used Q-learning to determine parameter values of a fitness function. In those experiments we found that when using the Q-learned parameter values instead of the values we estimated originally, turning points were more than halved. We have not tested whether the fitness functions used in the present experiments can be improved, but experience tells us that it is quite likely that they can.

5. Difficulty scaling

Many researchers and game developers consider game AI, in general, to be entertaining when it is difficult to defeat (Buro, 2003). Although for strong players that may be true, for novice players a game is most entertaining when it is challenging but beatable (Scott, 2002). To ensure that the game remains interesting, the issue is not for the computer to produce occasionally a weak move so that the human player can win, but rather to produce not-so-strong moves under the proviso that, on a balance of probabilities, they should go unnoticed (Iida et al., 1995). 'Difficulty scaling' is the automatic adaptation of a game, to set the challenge that the game poses to a human player. When applied to game AI, difficulty scaling aims at achieving an 'even game', i.e., a game wherein the playing strength of the computer and the human player match.

Many games provide a 'difficulty setting', i.e., a discrete value that determines how difficult the game will be. The purpose of a difficulty setting is to allow both novice and experienced players to enjoy the appropriate challenge the game offers (Charles & Black, 2004). The difficulty setting commonly has the following three problematic issues. First, the setting is *coarse*, with the player having a choice between only a limited number of

difficulty levels (usually three or four). Second, the setting is *player-selected*, with the player unable to assess which difficulty level is appropriate for his skills. Third, the setting has a *limited scope*, (in general) only affecting the computer-controlled agents' strength, and not their tactics. Consequently, even on a 'high' difficulty setting, the opponents exhibit similar behaviour as on a 'low' difficulty setting, despite their greater strength.

The three issues mentioned may be alleviated by applying dynamic scripting enhanced with an adequate difficulty-scaling mechanism. Dynamic scripting changes the computer's tactics to the way a game is played. As such, (i) it makes changes in small steps (i.e., it is not coarse), (ii) it makes changes automatically (i.e., it is not player-selected), and (iii) it affects the computer's tactics (i.e., it does not have a limited scope).

This section describes how dynamic scripting can be used to create new opponent tactics while scaling the difficulty level of the game AI to the experience level of the human player. Specifically, in this section the goal of dynamic scripting is to generate scripts in a way that the number of wins of the dynamic team is about equal to the number of losses at all times, even in changing environments. The section describes three different enhancements to the dynamic-scripting technique that let opponents learn how to play an even game, namely (i) high-fitness penalising, (ii) weight clipping, and (iii) top culling. The three enhancements are explained in Subsections 5.1, 5.2, and 5.3, respectively. The enhancements are evaluated in an experiment, of which the results are presented in Subsection 5.4, and discussed in Subsection 5.5.

5.1. High-fitness penalising

The weight adjustment expressed in Eq. 6 gives rewards proportional to the fitness value: the higher the fitness, the higher the reward. To elicit mediocre instead of good behaviour, the weight adjustment can be changed to give highest rewards to mediocre fitness values, and lower rewards or even penalties to high fitness values. With high-fitness penalising weight adjustment is expressed by Eq. 6, where F is replaced by F' defined as follows.

$$F' = \begin{cases} \dfrac{F}{p} & \{F \leq p\} \\ \dfrac{1-F}{p} & \{F > p\} \end{cases} \tag{7}$$

In Eq. 7, F is the calculated fitness value, and $p \in [0.5, 1]$, $p > b$, is the reward-peak value, i.e., the fitness value that should get the highest reward. The higher the value of p, the more effective opponent behaviour will be. Figure 7 illustrates the weight adjustment as a function of the original fitness (left) and the high-fitness-penalising fitness (right), with the mapping of F to F' in between. Angles α and β are equal.

Since the optimal value for p depends on the tactic that the human player uses, it was decided to let the value of p adapt to the perceived difficulty level of a game, as follows. Initially p starts at a value p_{init}. After every fight that is lost by the computer, p is increased by a small amount p_{inc}, up to a predefined maximum p_{max}. After every fight that is won by the computer, p is decreased by a small amount p_{dec}, down to a predefined minimum p_{min}. By running a series of tests with static values for p, we found that good values for p are found close to 0.7. Therefore, in the experiment we used $p_{\text{init}} = 0.7$, $p_{\text{min}} = 0.65$, $p_{\text{max}} = 0.75$, and $p_{\text{inc}} = p_{\text{dec}} = 0.01$.

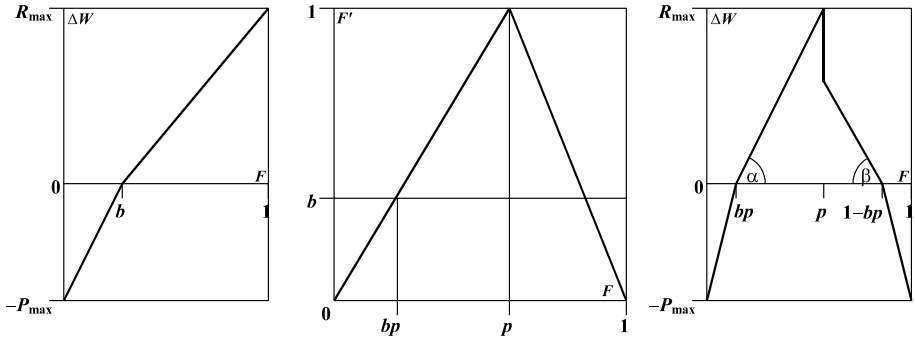

Fig. 7 Comparison of the original weight-adjustment formula (left) and the high-fitness-penalising weight-adjustment formula (right), by plotting the weight adjustments as a function of the fitness value F. The middle graph displays the relation between F and F'.

5.2. Weight clipping

During the weight updates, the maximum weight value W_{max} determines the maximum level of optimisation a learned tactic can achieve. A high value for W_{max} allows the weights to grow to large values, so that after a while the most effective rules will almost always be selected. This will result in scripts that are close to optimal. A low value for W_{max} restricts weights in their growth. This enforces a high diversity in generated scripts, most of which will be mediocre.

Weight clipping automatically changes the value of W_{max}, with the intent to enforce an even game. It aims at having a low value for W_{max} when the computer wins often, and a high value for W_{max} when the computer loses often. The implementation is as follows. After the computer wins a fight, W_{max} is decreased by W_{dec} per cent (but not lower than the initial weight value W_{init}). After the computer loses a fight, W_{max} is increased by W_{inc} per cent.

Figure 8 illustrates the weight-clipping process and the associated parameters. The shaded bars represent weight values for arbitrary rules on the horizontal axis. Before the weight adjustment, W_{max} changes by W_{inc} or W_{dec} per cent, depending on the outcome of the fight. After the weight adjustment, in Fig. 8 the weight value for rule 4 is too low, and will be increased to W_{min} (the arrow marked 'a'), while the weight value for rule 2 is too high, and

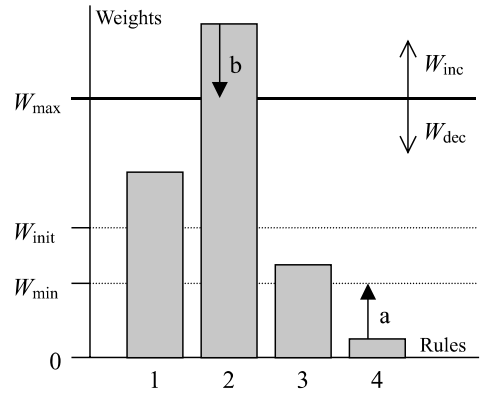

Fig. 8 Weight clipping and top culling process and parameters.

will be decreased to $W_{\max}$ (the arrow marked 'b'). In the experiment we decided to use the same initial values as were used for the performance-validation experiment, i.e., $W_{\text{init}} = 100$, $W_{\min} = 0$, and an initial value for $W_{\max}$ of 2000. W_{inc} and W_{dec} were both set to 10 per cent.

5.3. Top culling

Top culling is quite similar to weight clipping. It employs the same adaptation mechanism for the value of $W_{\max}$. The difference is that top culling allows weights to grow beyond the value of $W_{\max}$. However, rules with a weight greater than $W_{\max}$ will not be selected for a generated script. Consequently, when the computer-controlled opponents win often, the most effective rules will have weights that exceed $W_{\max}$, and cannot be selected, and thus the opponents will use weak tactics. Alternatively, when the computer-controlled opponents lose often, rules with high weights will be selectable, and the opponents will use strong tactics. So, while weight clipping achieves weak tactics by promoting variety, top culling achieves weak tactics by removing access to the most effective domain knowledge.

In Fig. 8, contrary to weight clipping, top culling will leave the value of rule 2 unchanged (the action represented by arrow (b) will not be performed). However, rule 2 will be unavailable for selection, because its value exceeds $W_{\max}$.

5.4. Difficulty-scaling results

To test the effectiveness of the three difficulty-scaling enhancements, we ran an experiment in the simulated CRPG. The experiment consisted of a series of tests, executed in the same way as the performance-validation experiment (Section 4). The experiment aimed at assessing the performance of a team controlled by the dynamic-scripting technique using a difficulty-scaling enhancement (with $P_{\max} = 100$, and all other parameters equal to the values used in the performance-validation experiment), against a team controlled by static scripts. If the difficulty-scaling enhancements work as intended, dynamic scripting will balance the game so that the number of wins of the dynamic team is roughly equal to the number of losses.

For the static team, we added an eighth tactic to the seven tactics described in Subsection 4.4, called the 'novice' tactic. The 'novice' tactic resembles the playing style of a novice CRPG player, who has learned the most obvious successful tactics, but has not yet mastered the subtleties of the game. While normally the 'novice' tactic will not be defeated by arbitrarily choosing rules from the rulebase, there are many different tactics that can be employed to defeat it, which the dynamic team will discover quickly. Without a difficulty-scaling enhancement, against the 'novice' tactic the dynamic team's number of wins in general will greatly exceed its losses.

For each of the tactics, we ran 100 tests in which dynamic scripting was enhanced with each of the three difficulty-scaling enhancements, and, for comparison, also without difficulty-scaling enhancements (called 'plain'). Each test consisted of a sequence of 150 encounters between the dynamic team and the static team. Because in each of the tests the dynamic-scripting technique starts with a rulebase with all weights equal, the first 50 encounters were used for finding a balance of well-performing weights. We recorded the number of wins of the dynamic team over the last 100 encounters.

The results of these tests are displayed in Table 5. For each combination of tactic and difficulty-scaling enhancement the table shows the average number of wins over 100 tests, and the associated standard deviation. To be recognised as an even game, we decided that the average number of wins over all tests must be close to 50. To take into account random fluctuations, in this context 'close to 50' means 'within the range [45, 55]'. In Table 5, all

Table 5 Difficulty-scaling results, averaged over 100 tests.

	Plain		High-fitness Penalising		Weight Clipping		Top Culling	
Tactic	Avg.	Dev.	Avg.	Dev.	Avg.	Dev.	Avg.	Dev.
Offensive	61.2	16.4	**46.0**	15.1	**50.6**	9.4	**46.3**	7.5
Disabling	86.3	10.4	56.6	8.8	67.8	4.5	**52.2**	3.9
Cursing	56.2	11.7	42.8	9.9	**48.4**	6.9	**46.4**	5.6
Defensive	66.1	11.9	39.7	8.2	**52.7**	4.2	**49.2**	3.6
Novice	75.1	13.3	**54.2**	13.3	**53.0**	5.4	**49.8**	3.4
Random team	55.8	11.3	37.7	6.5	**50.0**	6.9	**47.4**	5.1
Random agent	58.8	9.7	44.0	8.6	**51.8**	5.9	**48.8**	4.1
Consecutive	**51.1**	11.8	34.4	8.8	**48.7**	7.7	**45.0**	7.3

cell values indicating an even game are marked in bold font. From the table the following four results can be derived.

First, dynamic scripting without a difficulty-scaling enhancement results in wins significantly exceeding losses for all tactics except for the 'consecutive' tactic (with a reliability >99.9%; Cohen, 1995). This supports the viability of dynamic scripting as a learning technique, and also supports our statement in Subsection 4.4 that the 'consecutive' tactic is the most difficult tactic to learn a counter-tactic against. Note that the fact that, on average, dynamic scripting plays an even game against the 'consecutive' tactic is not because it is unable to consistently defeat this tactic, but because dynamic scripting continues learning after it has reached an optimum. Therefore, it can 'forget' what it previously learned, especially against an superior tactic like the 'consecutive' tactic.

Second, high-fitness penalising performs considerably worse than the other two enhancements. It cannot achieve an even game against six out of the eight tactics.

Third, weight clipping is successful in enforcing an even game in seven out of eight tactics. It does not succeed against the 'disabling' tactic. This is caused by the fact that the 'disabling' tactic is so easy to defeat, that even a rulebase with all weights equal will, on average, generate a script that defeats this tactic. Weight clipping can never generate a rulebase worse than 'all weights equal'. A comparison between Table 1 and Table 5 shows that, indeed, the number of wins achieved with weight clipping is not significally different from the number of wins achieved by non-learning dynamic scripting.

Fourth, top culling is successful in enforcing an even game against all eight tactics.

Histograms for the tests with the 'novice' tactic are displayed in Fig. 9. On the horizontal axis the number of wins for the dynamic team out of 100 fights is displayed. The bar length indicates the number of tests that resulted in the associated number of wins.

From the histograms we derive the following result. While, on average, all three difficulty-scaling enhancements manage to enforce an even game against the 'novice' tactic, the number of wins in each of the tests is much more 'spread out' for the high-fitness-penalising enhancement than for the other two enhancements. This indicates that high-fitness penalising results in a higher variance of the distribution of won games than the other two enhancements. The top-culling enhancement seems to yield the lowest variance. This is confirmed by an approximate randomisation test (Cohen, 1995), which shows that against the 'novice' tactic, the variance achieved with top culling is significantly lower than with the other two enhancements (reliability > 99.9%). We observed similar distributions of won games against the other tactics, except that against some of the stronger tactics, a few exceptional outliers occurred with a significantly lower number of won games. The rare outliers were caused

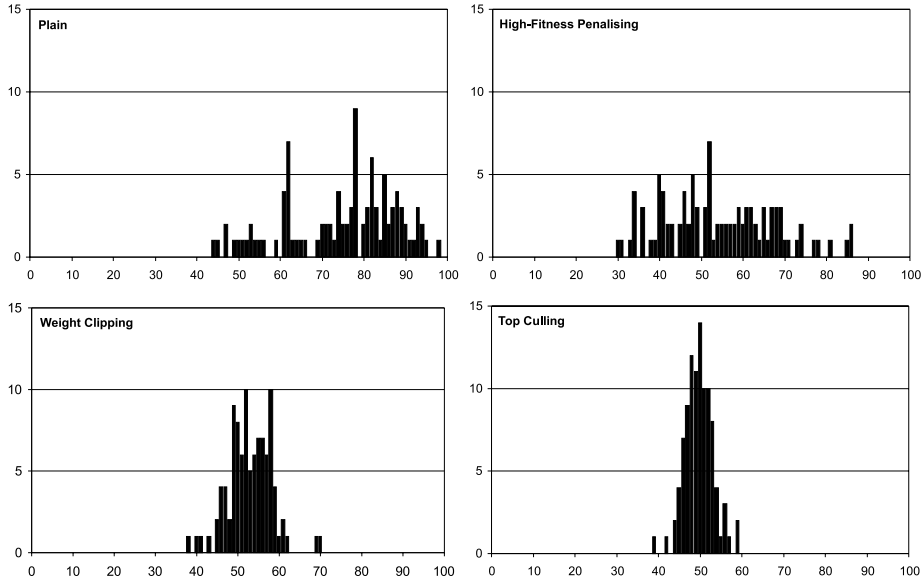

Fig. 9 Histograms of 100 tests of the achieved number of wins in 100 fights, against the 'novice' tactic.

by the fact that, occasionally, dynamic scripting requires more than 50 encounters to find a well-performing set of weights when playing against a strong static tactic.

Our results show that, when dynamic scripting is enhanced with the top-culling difficulty-scaling mechanism, it meets the requirement of scalability.

5.5. Discussion

Of the three different difficulty-scaling enhancements the top-culling enhancement is the best choice. It has the following four advantages: (i) it gives the most reliable results, (ii) it is easily implemented, (iii) of the three enhancements, it is the only one that manages to force an even game against inferior tactics, and (iv) it continues learning strong behaviour even while it exhibits scaled behaviour.

Obviously, the worst choice is the high-fitness-penalising enhancement. In an attempt to improve high-fitness penalising, we performed some tests with different ranges and adaptation values for the reward-peak value p, but these worsened the results. However, we cannot rule out the possibility that with a different fitness function high-fitness penalising will give better results.

An additional possibility with weight clipping and top culling is that they can also be used to set a different desired win-loss ratio, simply by changing the rates with which the value of $W_{\max}$ fluctuates. For instance, by using top culling with $W_{\text{dec}} = 30$ per cent instead of 10 per cent, leaving all other parameters unchanged, after 100 tests against the 'novice' tactic we derived an average number of wins of 35.0 with a standard deviation of 5.6.

Notwithstanding the successful results we achieved, a difficulty-scaling enhancement should be an optional feature in a game, that can be turned off by the player, for the following two reasons: (i) when confronted with an experienced player, the learning process should aim for the best possible tactics without interference from a difficulty-scaling enhancement, and

Fig. 10 A fight between two teams in NEVERWINTER NIGHTS.

(ii) some players will feel that attempts by the computer to force an even game diminishes their accomplishment of defeating the game, so they may prefer not to use it.

6. Validation in practice

To support the successful results of dynamic scripting in a practical setting, we decided to test the technique in an actual state-of-the-art commercial game. In this section we present the selected game (6.1), the scripts and rulebases (6.2), the weight-update function (6.3), the tactics used by the static team (6.4), the results of an evaluation of dynamic scripting in practice (6.5), and a discussion of the results (6.6).

6.1. Neverwinter nights

To evaluate dynamic scripting in practice, we chose the game NEVERWINTER NIGHTS (2002), a CRPG of similar complexity as the BALDUR'S GATE games, developed by BioWare Corp. A major reason for selecting NEVERWINTER NIGHTS is that the game is easy to modify and extend. It comes with a toolset that allows the user to develop completely new game modules. The toolset provides access to the scripting language and all the scripted game resources, including the game AI. While the scripting language is not as powerful as modern programming languages, we found it to be sufficiently powerful to implement dynamic scripting.

We implemented a small module in NEVERWINTER NIGHTS, similar to the simulated CRPG used for the previous experiments. The module contains an encounter between a dynamic team and a static team of similar composition. This is illustrated in Fig. 10. Each team consists of a fighter, a rogue, a priest and a wizard of equal experience level. In contrast to the agents in the simulated CRPG, the inventory and spell selections in the NEVERWINTER NIGHTS module cannot be changed, due to the toolset lacking functions to achieve such modifications. This has a restrictive impact on the tactics.

6.2. Scripts and rulebases

The default game AI in NEVERWINTER NIGHTS is very general in order to facilitate the development of new game modules (e.g., it does not refer to casting of a specific magic spell, but to casting of spells from a specific class). It distinguishes between about a dozen opponent classes. For each opponent class it sequentially checks a number of environmental variables and attempts to generate an appropriate response. The behaviour generated is not predictable, because it is partly probabilistic.

For the implementation of the dynamic-scripting technique, we first extracted the rules employed by the default game AI, and entered them in every appropriate rulebase. To these standard rules we added several new rules. The new rules were similar to the standard rules, but slightly more specific, e.g., referring to a specific enemy instead of referring to a random enemy. We also added a few 'empty' rules, which, if selected, allow the game AI to decrease the number of effective rules. We set priorities similar to the priorities used in the simulated CRPG.

Note that since the rules extracted from the default game AI are generalised, the rules used by dynamic scripting are generalised too. The use of generalised rules in the rulebase has the advantage that the rulebase gets trained for generating AI for agents of any experience level.

The size of the scripts for both a fighter and a rogue were set to five rules, which were selected out of rulebases containing 21 rules. The size of the scripts for both a priest and a wizard were set to ten rules, the rulebase for the priest containing 55 rules and the rulebase for the wizard containing 49 rules. To the end of each script a call to the default game AI was added, in case no rule could be activated.

6.3. Weight-update function

The weight adjustment mechanism we used in NEVERWINTER NIGHTS was similar to the mechanism we used in the simulated CRPG (4.3). We decided to differ slightly from the implementation of these functions in the simulation, mainly to avoid problems with the NEVERWINTER NIGHTS scripting language.

The team-fitness $F(g)$, which yields a value in the range [0, 1], was defined as follows.

$$F(g) = \begin{cases} 0 & \{g \text{ lost}\} \\ \frac{1}{5} + \sum_{c \in g, h_T(c) > 0} \frac{2}{5N_g} \left(1 + \frac{h_T(c)}{h_0(c)}\right) & \{g \text{ won}\} \end{cases} \tag{8}$$

All variables in Eq. 8 were defined as those in Eq. 1. The agent-fitness $F(a, g)$, which yields a value in the range [0, 1], was defined as follows.

$$F(a, g) = \frac{1}{2} F(g) + \frac{1}{2} \begin{cases} \min\left(\frac{2D(a)}{D_{\max}}, \frac{3}{5}\right) & \{h_T(a) \leq 0\} \\ \frac{3}{5} + \frac{2h_T(a)}{5h_0(a)} & \{h_T(a) > 0\} \end{cases} \tag{9}$$

All variables in Eq. 9 were defined as those in Eqs. 2 to 5.

Table 6 Turning point values for dynamic scripting NEVERWINTER NIGHTS.

Tactic	Tests	Avg.	St.dev.	Median	Highest	Top 5
AI 1.29	50	21	8.8	16	101	58
AI 1.61	31	35	18.8	32	75	65
Cursed AI	21	33	21.8	24	92	64

Weight adjustment was implemented according to Eq. 6, with all parameter values as in the performance-validation experiment, except for the maximum penalty $P_{\max}$, which was set to 50. Furthermore, rules in the script that were not executed during the encounter, instead of being treated as not being in the script at all, were assigned half the reward or penalty received by the rules that were executed. The main reason for this is that if there were no rewards and penalties for the non-executed rules, the empty rules would never get rewards or penalties. We did not use history fallback to recover earlier rulebases.

6.4. Tactics

In our experiment we used three different tactics for the static team, all based on the default game AI, implemented by the NEVERWINTER NIGHTS developers. The three tactics were the following.

AI 1.29: AI 1.29 is the default game AI used in NEVERWINTER NIGHTS version 1.29. We used this version of NEVERWINTER NIGHTS for our earliest tests.

AI 1.61: AI 1.61 is the default game AI used in NEVERWINTER NIGHTS version 1.61. We used this version of NEVERWINTER NIGHTS for our later tests. Between version 1.29 and 1.61 the game AI was significantly improved by the game developers.

Cursed AI: We created a 'cursed' version of AI 1.61. With cursed AI in 20 per cent of the encounters the game AI deliberately misleads dynamic scripting into awarding high fitness to purely random tactics, and low fitness to tactics that have shown good performance during earlier encounters.

6.5. NEVERWINTER-NIGHTS RESULTS

Table 6 summarises the results from the repetition of the performance-validation experiment in the NEVERWINTER NIGHTS environment. The columns of the table represent, from left to right, (i) the tactic used, (ii) the number of tests executed, (iii) the average turning point, (iv) the standard deviation, (v) the median turning point, (vi) the highest value for the turning point, and (vii) the average of the five highest values. We did not perform tests with penalty balancing since already in the earliest experiments with NEVERWINTER NIGHTS we used higher maximum penalties than in the simulated CRPG. From the results in Table 6 we draw the conclusion that dynamic scripting meets the requirement of efficiency easily. There are a few (fairly low) outliers, but these may be resolved with biased rulebases, as with the CRPG simulation.

We also validated the results achieved with the top-culling enhancement in NEVERWINTER NIGHTS. Without top culling, in ten tests dynamic scripting achieved an average number of wins out of 100 fights of 79.4, with a standard deviation of 12.7. With top culling, in ten tests dynamic scripting achieved an average number of wins out of 100 fights of 49.8, with a

standard deviation of 3.4. The results clearly support that dynamic scripting, enhanced with top culling, meets the requirement of scalability.

6.6. Discussion

The NEVERWINTER NIGHTS experiment supports the results achieved with dynamic scripting in a simulated CRPG. Comparison of all results seems to indicate that dynamic scripting performs even better in NEVERWINTER NIGHTS than in the simulated CRPG. This is caused by the fact that the default game AI in NEVERWINTER NIGHTS is designed to be effective for all opponents that can be designed in the toolset. Since it is not specialised, for most agents it is not optimal. Therefore, there is a great variety of tactics that can be used to deal with it, which makes it fairly easy for dynamic scripting to discover a successful counter-tactic.

In general, the better the tactic against which dynamic scripting is tested, the longer it will take for dynamic scripting to gain the upper hand. Moreover, because dynamic scripting is designed to generate a wide variety of tactics (in compliance with the requirement of variety), it will never gain the upper hand if the tactic against which it is pitted is so strong that there are *very* few viable counter-tactics. Against human players, this means that dynamic scripting will achieve the most satisfying results against non-expert players.

In a game that allows the design of 'super-tactics', which are almost impossible to defeat, dynamic scripting may not give satisfying results when used against expert players who know and use these super-tactics. However, *every* machine-learning technique will require more computational resources finding rare solutions than finding ubiquitous solutions. Therefore, against superior tactics, instead of using an online machine-learning technique, in general it will be more effective to use counter-tactics that have been trained against these optimal tactics in an offline learning process. It should be noted that the existence of super-tactics in a game is actually an indication of bad game-design, because they make the game too hard when employed by the computer, and too easy when employed by the human player.

We have shown that dynamic scripting is applicable to combat in CRPGs, but the technique is more generally applicable than that. In parallel research, good results were achieved with the application of dynamic scripting to plan-generation in Real-Time Strategy games (Ponsen & Spronck, 2004; Dahlbom, 2004). To games that use game AI that is not implemented in scripts, dynamic scripting is not directly applicable. However, based on the idea that domain knowledge must be the core of an online adaptive game-AI technique, an alternative for dynamic scripting may be designed. For instance, if game AI is based on a finite-state machine, state transitions can be extracted from a rulebase to construct the finite-state machine, in a way similar to dynamic scripting's selection of rules for a game-AI script.

7. Conclusions and future work

Dynamic scripting meets the requirements of speed, effectiveness, robustness, clarity and variety by design. In Section 4 it was shown that dynamic scripting meets the requirements of efficiency and consistency. In Section 5 it was shown that by applying top culling, dynamic scripting meets the requirement of scalability. Therefore, we conclude that dynamic scripting meets all eight requirements specified in Subsection 2.3, and thus can be applied in actual commercial games for the implementation of adaptive game AI. As was shown in Section 6, our conclusion is supported by the good results achieved by dynamic scripting in the state-of-the-art CRPG NEVERWINTER NIGHTS.

In future work, we intend to investigate the effectiveness and entertainment value of dynamic scripting in games played against actual human players. While such a study requires many subjects and a careful experimental design, the game-play experiences of human players are important to convince game developers to adopt dynamic scripting in their games.

On a final note, we expect that, even if we have high hopes for adaptive game AI, the distrust of game developers and publishers for adaptive techniques will not dissolve quickly. Therefore we wish to point out an inherently risk-free stage in game development where adaptive techniques can prove their worth, namely playtesting. During playtesting dynamic scripting can be used to design new tactics to deal with tactics that playtesters use with success, and can detect possible exploits and game-balancing issues in manually-designed game AI. Naturally, if playtesters enjoy playing against adaptive game AI, the developers might consider its inclusion in a released game.

Acknowledgments This research is supported by a grant from the Dutch Organisation for Scientific Research (NWO grant No. 612.066.406). The authors wish to express their gratitude to the University of Alberta GAMES Group for their contribution to this research, and to BioWare Corp. for their enlightening commentary.

The simulation environment and the NEVERWINTER NIGHTS modules mentioned in the article, including all rulebases, are available from the first author's website (www.cs.unimaas.nl/p.spronck/).

References

Adamatzky, A. (2000). CREATURES - artificial life, autonomous agents and gaming environment. *Kybernetes: The International Journal of Systems & Cybernetics 29:2*.

Allen, M., Suliman, H., Wen, Z., Gough, N., & Mehdi, Q. (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* (pp. 22–32). SCS Europe Bvba.

Brockington, M., & Darrah, M. (2002). How *not* to implement a basic scripting language. In: S. Rabin (ed.): *AI Game Programming Wisdom* (pp. 548–554). Charles River Media, Inc.

Buro, M. (2003). RTS games as test-bed for real-time AI research. In: K. Chen, S. Chen, H. Cheng, D. Chiu, S. Das, R. Duro, Z. Jiang, N. Kasabov, E. Kerre, H. Leong, Q. Li, M. Lu, M. Grana Romay, D. Ventura, P. Wang, and J. Wu (eds.): *Proceedings of the 7th Joint Conference on Information Science (JCIS 2003)* (pp. 481–484).

Buro, M. (2004). Call for AI research in RTS games. In: *Proceedings of the AAAI-04 Workshop on Challenges in Game AI* (pp. 139–142). AAAI Press.

Charles, D., & Black, M. (2004). Dynamic player modelling: A framework for player-centric digital games. In: Q. Mehdi, N. Gough, S. Natkin, and D. Al- Dabass (eds.): *Computer Games: Artificial Intelligence, Design and Education (CGAIDE 2004)* (pp. 29–35). University of Wolverhampton.

Charles, D., & Livingstone, D. (2004). AI: The missing link in game interface design. In: M. Rauterberg (ed.): *Entertainment Computing – ICEC 2004* (pp. 351–354). Springer-Verlag.

Cohen, P. (1995). *Empirical Methods for Artificial Intelligence*. MIT Press.

Dahlbom, A. (2004). *An Adaptive AI for Real-Time Strategy Games*, M.Sc. thesis. Högskolan i Skövde.

Demasi, P., & Cruz, A. (2002). Online coevolution for action games. *International Journal of Intelligent Games and Simulation 2:2*, 80–88.

Evans, R. (2002). Varieties of learning. In: S. Rabin (ed.): *AI Game Programming Wisdom* (pp. 567–578). Charles River Media, Inc.

Forbus, K., & Laird, J. (2002). AI and the entertainment industry. *IEEE Intelligent Systems 17:4*, 15–16.

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

Graepel, T., Herbrich, R., & Gold, J. (2004). Learning to fight. In: Q. Mehdi, N. Gough, S. Natkin, and D. Al-Dabass (eds.): *Computer Games: Artificial Intelligence, Design and Education (CGAIDE 2004)*. (pp. 193–200). University of Wolverhampton.

Iida, H., Handa, K., & Uiterwijk, J. (1995). Tutoring strategies in game-tree search. *ICCA Journal 18:4*, 191–204.

Johnson, S. (2004). Adaptive AI: A practical example. In: S. Rabin (ed.): *AI Game Programming Wisdom 2*, 639–647. Charles River Media, Inc.

Laird, J., & Van Lent, M. (2001). Human-level's AI killer application: Interactive Computer Games. *Artificial Intelligence Magazine 22:2*, 15–26.

Lidén, L. (2004). Artificial Stupidity: The art of making intentional mistakes. In: S. Rabin (ed.): *AI Game Programming Wisdom 2*, 41–48. Charles River Media, Inc.

Madeira, C., Corruble, V., Ramalho, G., & Ratitch, B. (2004). Bootstrapping the learning process for the semi-automated design of challenging game AI. In: D. Fu, S. Henke, and J. Orkin (eds.): *Proceedings of the AAAI-04 Workshop on Challenges in Game Artificial Intelligence*. (pp. 72–76). AAAI Press.

Manslow, J. (2002). Learning and adaptation. In: S. Rabin (ed.): *AI Game Programming Wisdom*. (pp. 557–566). Charles River Media, Inc.

Michalewicz, Z., & Fogel, D. (2000) *How To Solve It: Modern Heuristics*. Springer- Verlag.

Nareyek, A. (2002). Intelligent agents for computer games. In: T. Marsland and I. Frank (eds.): *Computers and Games, Second International Conference, CG 2000*, Vol. 2063 of *Lecture Notes in Computer Science*. (pp. 414–422). Springer-Verlag.

Ponsen, M., Muñoz-Avila, H., Spronck, P., & Aha, D. (2005). Automatically acquiring adaptive real-time strategy game opponents using evolutionary learning. In: *Proceedings of the Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence*. (pp. 1535-1540), AAAI Press.

Ponsen, M., & Spronck, P. (2004). Improving adaptive game AI with evolutionary learning. In: Q. Mehdi, N. Gough, S. Natkin, and D. Al-Dabass (eds.): *Computer Games: Artificial Intelligence, Design and Education (CGAIDE 2004)* (pp. 389–396). University of Wolverhampton.

Rabin, S. (2004). Promising game AI techniques. In: S. Rabin (ed.): *AI Game Programming Wisdom 2*, 15–27. Charles River Media, Inc.

Schaeffer, J. (2001). A gamut of games. *Artificial Intelligence Magazine 22:3*, 29–46.

Scott, B. (2002). The illusion of intelligence. In: S. Rabin (ed.): *AI Game Programming Wisdom*. (pp.16–20). Charles River Media, Inc.

Spronck, P., Sprinkhuizen-Kuyper, I., & Postma, E. (2003). Improving opponent intelligence through offline evolutionary learning. *International Journal of Intelligent Games and Simulation 2:1*, 20–27.

Spronck, P. (2005). *Adaptive Game AI*, Ph.D. thesis. Universitaire Pers Maastricht.

Sutton, R., & Barto, A. (1998). *Reinforcement Learning: An Introduction*. MIT Press.

Tomlinson, S. (2003). Working at thinking about playing or a year in the life of a games AI programmer. In: Q. Mehdi, N. Gough, and S. Natkin (eds.): *Proceedings of the 4th International Conference on Intelligent Games and Simulation (GAME-ON 2003)*. (pp. 5–12). EUROSIS.

Tozour, P. (2002a). The evolution of game AI. In: S. Rabin (ed.): *AI Game Programming Wisdom*. (pp. 3–15). Charles River Media, Inc.

Tozour, P. (2002b). The perils of AI scripting. In: S. Rabin (ed.): *AI Game Programming Wisdom*. (pp. 541–547). Charles River Media, Inc.

Woodcock, S. (2000). The future of game AI: A personal view. *Game Developer Magazine 7:8*.