KEEPING ADAPTIVE GAME AI INTERESTING

István Szita, Marc Ponsen,[1] and Pieter Spronck[2]
[1]Maastricht University / MICC
P.O. Box 616
6200 MD Maastricht, The Netherlands
m.ponsen@cs.unimaas.nl

[2]Tilburg University / TiCC
P.O. Box 90153
5000 LE Tilburg, The Netherlands
p.spronck@uvt.nl

KEYWORDS
Game AI, cross-entropy, reinforcement, dynamic scripting

ABSTRACT

We propose a method for automatically learning diverse but effective macros that can be used as components of game AI scripts. Macros are learnt by a selection-based optimisation method that maximises an interestingness measure. Our demonstrations are performed in a CRPG simulation with two wizards duelling. The results show that the macros learned this way can increase adaptivity, and diversity, while retaining playing strength.

INTRODUCTION

The main purpose of commercial computer games is to entertain the human player. Most of them do this by posing challenges for the human to overcome, often in the form of combat. The tactics used by the computer-controlled opponents in combat are determined by the game's artificial intelligence (AI). If the game AI manages to keep players motivated to play the game, we call it 'interesting.'

Two key factors in making game AI interesting are *effectiveness* and *diversity*. If game AI is effective, it is believable and a challenge to defeat. If game AI is diverse, it provides a variety of tactics for the player to test his skills against. A major problem is that these two factors are often in conflict: effective tactics usually consist of a specific sequence of actions with little room for variety (Spronck 2005). In theory, adaptive game AI can improve effectiveness against a specific player automatically, while maintaining diversity by constantly trying new tactics. In practice, however, in its quest to improve effectiveness adaptive game AI tends to converge to a very small number of strong tactics, thereby losing diversity. Adaptive AI in actual commercial games is also hampered by the fact that games can afford only a few ineffective opponents before the player gets bored. To counteract this issue, adaptive game AI must be based on a-priori knowledge.

In this article we pose the following problem statement: To what extent is it possible to automatically create a-priori knowledge, that can be used to generate game AI that is interesting, i.e., both effective and diverse.

PRELIMINARIES

The dominant approach to programming game AI is scripting (Nareyek 2002; Tomlinson 2003; Tozour 2002). Scripts are preferred by game developers, because they are easy to implement, interpret, and modify. However, despite their advantages, scripts also have numerous disadvantages: they are time-consuming to write, they may contain exploits, they are often predictable, and they are non-adaptive.

A simple way to increase the diversity of game AI is to use multiple different scripts. However, such diversity comes at the price of an increased amount of programming and testing needed. Furthermore, the AI remains non-adaptive and leaves the possibility of easy-to-exploit weaknesses. In practice, random actions are also often applied to make AI more diverse and unpredictable. However, the style of game AI still remains predictable because it is rarely affected by this kind of randomness. Furthermore, randomly chosen actions may seem totally out of place, and thus inversely affect the illusion of intelligent behaviour.

Dynamic scripting and Macros

Dynamic scripting (Spronck et al. 2006) is a reinforcement-learning method for automatically acquiring effective scripts for game AI. In various games, such as *Neverwinter Nights* (Spronck et al. 2006) and *Wargus* (Ponsen et al. 2005), dynamic scripting adapted quickly to a number of static tactics and learned effective counter-tactics. Ponsen et al. (2006) applied evolutionary learning to learn macros, and showed that the learned rules improve performance. All these approaches aimed at learning *effective* tactics. However, they have no built-in mechanism to improve *diversity*, so they tend to converge to static, predictable scripts.

In their simplest form, macro actions are sequences of actions that are handled as a single unit. Macro actions offer a straightforward way to augment dynamic scripting. With larger building blocks, dynamic scripting should be able to switch between different playing styles more quickly. However, without a careful selection of the actions to be grouped, additional macros can even decrease learning performance. To fulfill their purpose, macro actions have to meet at least three requirements:

1. **Effectiveness**. Macros should be effective in the sense that effective tactics can be assembled from them.
2. **Diversity**. Macros should differ considerably from each other and should represent different playing styles.
3. **Appropriate size**. To ensure diversity, the size of macros should be balanced so that they are longer than one line (otherwise it is regular dynamic scripting), but not too long (lest a macro will be equal to a full script).

Searching for diversity

Searching for diversity is a central problem in reinforcement learning. The agent must explore new, unknown situations so that it gains knowledge of the system. At the same time,

obtained knowledge should be used for collecting rewards, and the agent must find a balance between the two kinds of activities. There is a rich literature of exploration methods in reinforcement learning (Wiering 1999). Most exploration methods, however, are specific to (finite) Markov decision processes. For example, they assume that we can maintain a visit count for each state. Moreover, these methods perform exploration in order to find a single optimal policy (or value function). In contrast, our aim is to maintain diversity, with policies that are not necessarily optimal, but still effective.

Schmidhuber (1991) defines an area as "boring" either if it is well known to the agent and not much is left to learn, or if it is poorly understood and it is hard to make any progress in learning. The area in-between is where the agent can learn quickest, and is therefore most interesting to him. Naturally, the area of interestingness is continually changing.

Optimization with the cross-entropy method

The cross-entropy method (Rubinstein 1999) is a general algorithm for global optimization tasks, bearing close resemblance to estimation-of-distribution evolutionary methods (Muehlenbein 1998). A short introduction to the algorithm is given in the next section, with emphasis on its application to macro learning. De Boer et al. (2004) provide a more general description. There are many successful applications of the cross-entropy method in machine learning, even some in the area of computer games (Szita and Lõrincz 2006a, 2006b).

SCHEME FOR MACRO LEARNING

Our goal is to learn macro actions that satisfy the requirements listed previously. The rules constituting the macro actions are selected from a rulebase. We maintain a probability distribution over the rules of the rulebase, and update probabilities so that macros with higher fitness become more probable. The fitness function depends on two factors: the first rewards strong macros (in terms of combat effectiveness), while the second rewards scripts that differ considerably from previously learned macros. Macros are learned incrementally as a separate optimisation task, since the fitness function depends on previously learned macros.

The macro-generation algorithm (summarised in Figure 1) starts with an empty macro list. For each prospective macro, we start a new learning epoch and initialise the probability vector. In the main learning loop, a script is generated according to the actual probability vector. It is evaluated in the game environment and the results are recorded. Then, its overall fitness is calculated, considering both playing strength and difference from previous macros. Finally, the probability vector is updated, according to the fitness of the current script. At the end of the learning epoch, a new macro is extracted and added to the list.

```
// K: number of macros to be learned
// T: number of battles in a training epoch
L := { };                          // start with empty list of macros
for k := 1 to K do
    phase := k div (K/2);          // 0/1: opening/midgame phase
    p := InitProbabilities();
    for i := 1 to T do
        S_i := GenerateScript(p,phase);    // random script s.t. p
        G_i := EvalScript(S_i);            // get battle outcome using S_i
        F_i := GetFitness(S_i,G_i,L);      // calculate fitness
        p := UpdateProbabs(p,i,{S_1,...,S_i},{F_1,...,F_i});
    M := ExtractMacro(p);
    L := L ∪ {M};                  // add new macro to the list
```

Figure 1: Main loop of macro action learning.

Game environment: MiniGate

We use the MiniGate environment (Spronck 2005), an open-source combat simulation of the *Baldur's Gate* games. All of our experiments are carried out on a single test problem, namely the duel of two wizards. This test problem was chosen because (a) there is a large variety of possible tactics; (b) there can be multiple different playing styles that are effective; and (c) the task is simple enough so that the results can be interpreted by humans relatively easily.

Both wizards are controlled by scripts. One has a static, fixed script. The other is adaptive, and its script is updated by the macro-learning procedure. In all other respects, the two wizards' capabilities are equal. The behaviour of a wizard is determined completely by its script: each time a decision needs to be made, the rules are checked in order, and the first applicable rule is executed. If there are no applicable rules, the wizard uses his sling as default attack.

Macros

In the case of this specific computer RPG environment, we can define macros relatively easily. In a typical battle, both wizards take about 6 to 9 decisions, after which they have no spells left and can fight only with slings until one of them dies. We decided that macro actions will be uniformly three actions long, so the adaptive agent can execute two macros, rounding off with several other actions at the end of the script. This decision is arbitrary but (a) it is reasonable according to our observations of the game; and (b) makes the segmentation problem trivial.

For ease of notation, we define *opening* and *midgame* phases of a battle. The opening phase of a battle lasts until the third decision of the adaptive agent, and is covered by an opening macro. Accordingly, the midgame phase lasts from action 4 through 6 and is covered by a midgame macro. A valid script contains at most one of each type of macro. A total of K=30 macros are to be learned, 15 for the opening and 15 for the midgame phase. Note that the procedure is extensible to games where there are more than two phases.

Script generation

The script generation routine is slightly different for the opening and the midgame macros. In both cases, full scripts are generated, from which the corresponding macro is extracted at the end of the evaluation phase (detailed below). However, for the learning of the opening macros scripts are assembled from single rules, while for the learning of midgame macros, an opening macro is completed with additional single rules.

For the first case, our script generation method draws the rules from a fixed rulebase according to the actual

probability distribution, and assembles a script from them. Let the number of rules in the rulebase be M (in our case, M=24). The probability vector $\underline{p}$ is an M-dimensional vector $\underline{p} = (p_1,...,p_M)$ where $p_i \in [0,1]$ for all $i \in \{1,...,M\}$. The script generation procedure selects rule i with probability p_i. Different rules are drawn independently from each other. The resulting script contains the rules in the same order as they appear in the rulebase.

For the learning of midgame macros, we assume that a set of opening macros has already been learned. An opening macro is drawn randomly according to probability distribution $\underline{p}_o=(\underline{p}_{o1},...,\underline{p}_{oO})$, where O is the number of opening macros, $\underline{p}_{oi} \geq 0$ for all i, and the sum of all $\underline{p}_{oi}$ is 1. One opening macro is selected; the probability of choosing the i^{th} one is $\underline{p}_{oi}$. We let the new script begin with the selected opening macro, then the consecutive rules are determined by the previous procedure. An example is shown in Figure 2.

```
[ opening macro #7 ]
   cast("Monster Summoning I", closestenemy);
   cast("Magic Missile", closestenemy);
   cast("Chromatic Orb", closestenemy);
[ simple rules ]
   if healthpercentage < 50 then
      drink( "Potion of Healing");
   cast("Fireball", closestenemy);
   cast("Luck");
   cast("Stinking Cloud", closestenemy);
   cast("Grease", closestenemy);
   cast("Flame Arrow", closestenemy);
   cast("Magic Missile", closestenemy);
```

Figure 2: A randomly generated script. The rule that casts a "Fireball" is never executed, because the wizard can cast only one level-3 spell, which was "Monster Summoning I". Further optimisation reduces the probability of such coincidences.

Script execution and generation of game records

To evaluate a script, the wizards play one battle. The adaptive wizard is controlled by the script to be tested, while the static wizard uses a fixed script. In our experiments, during training this fixed script was always the "summoning tactic" (described below). At the end of battle, we record the script of the adaptive wizard; the winner; the remaining hit points of the wizards at the end of battle; the duration of the battle (in game rounds); and the ordered list of rules that were used by the adaptive wizard (note that this may be different from his script, since rules can be skipped if their conditions are not met, or if the battle ended early). The recorded data is used for fitness calculation, updating the parameters and extracting macros.

Fitness calculation

There are two sources of reward for the agent: he gains rewards for the diversity of his script, and for being an effective combattant.

When calculating the reward for diversity, we compare the current script to all the previously extracted macros for the same phase. Let the number of such macros be K. Let the characteristic vector $\underline{v}_k$ of macro k ($1 \leq k \leq K$) be an M-dimensional 0/1-vector, its j^{th} component being 1 if macro k contains rule j and 0 otherwise. Furthermore, consider the macro that we could extract from the current script, that is, either the first three rules applied (opening phase) or the three rules applied after finishing an opening macro (midgame phase). Denote its characteristic vector by $\underline{v}_0$. The reward of the script for diversity will be proportional to the difference of $\underline{v}_0$ from all the other characteristic vectors:

$$F_{div} = \frac{1}{K}\sum_{k=1}^{K}\left\|\underline{v}_k - \underline{v}_0\right\|$$

Thus, if the rules of a macro are used in few or none of the already established macros, it gets a high reward for diversity.

The second part of the reward, F_{str}, comes from playing well. For this purpose, we used the individual fitness function of Spronck et al. (2006), without modifications. The agent receives rewards for (a) winning the fight, (b) for remaining health, (c) for causing damage, and (d) for staying alive as long as possible. We define the *overall fitness* of the script as a weighted sum of the two components: $F=F_{str}+c \cdot F_{div}$. Experimentally, we found that c=0.25 is a suitable value for balancing the two terms. F is used to guide the search in the space of possible scripts. Note that the fitness is highly stochastic, because the outcome of a battle depends on many random factors.

Parameter update: cross-entropy learning

Our goal is to update the probability vector $\underline{p}$ so that the chance of drawing high-fitness scripts increases. For updating we utilise the *cross-entropy method* (CEM). We provide here a short algorithmic description in the context of script-learning.

CEM is a population-based algorithm similar to evolutionary algorithms. Samples are drawn from a parameterised probability distribution. After evaluating a generation of samples, the best few percent of them is selected (the "elite samples") and used to update the probabilities. The pseudocode of CEM for script optimisation is shown in Figure 3. The algorithm has three parameters: the population size N, the selection ratio ρ and the step-size α. Our settings were N=100, ρ=0.1 and α=0.7. CEM is quite insensitive to the choice of ρ and α: several preliminary experiments indicated that its performance was fairly uniform in the intervals $0.05 \leq \rho \leq 0.25$ and $0.5 \leq \alpha \leq 0.8$. We generated a total of 1500 samples, corresponding to 15 update iterations. We found this choice sufficient for converging to near-deterministic solutions.

```
// update only if an N-sample generation is ready
if n mod N = 0 then
   // Sort last N samples according to fitness, best ones first
   ScriptList := SortLastN(ScriptList,FitnessList,N);
   Ne := ρ · N;                          // number of elite samples
   // count rule frequencies in elite samples
   for j := 1 to M do
      p'j := 0;                          // p': new probability vector
      for i := 1 to Ne do
         if ScriptListi contains rule j then p'j ++;
      p'j := p'j / Ne;
      for j := 1 to M do                 // Update probability vector
         pj := (1- α) · pj + α · p'j;
```

Figure 3: CEM script optimisation.

Macro extraction

As the result of the script optimisation procedure, we obtain a vector of probabilities containing one entry for each rule in the rulebase. For obtaining macros, we need to decide the probability that a particular rule was used in the opening (or midgame) phase, and probabilities need to be discretised to 0 or 1. We proceed as follows.

Let the number of the macro to be extracted be k. Let $\underline{w}_1,...,\underline{w}_N$ be the list of characteristic vectors of samples in the last iteration of CEM; that is, component j of vector $\underline{w}_i$ is 1 if rule j was fired during the corresponding phase of battle i, and 0 otherwise. Let the vector corresponding to macro k be:

$$\underline{v}_k = \frac{1}{N}\sum_{i=1}^{N}\underline{w}_i$$

From this vector, we can easily extract a macro: we select the three rules with the largest values. Note that for the calculation of diversity, we will use the non-integer vector $\underline{v}_k$, because it is more informative than just the list of the three most-often-used rules.

Results

We ran the algorithm described above for learning 15 opening-phase macros, and consecutively 15 midgame-phase ones. We repeated the experiment three times with different random number seeds.

An example of a learned opening-phase macro is:

```
cast( "Monster Summoning I", closestenemy )
cast( "Deafness", closestenemy )
cast( "Blindness", closestenemy ),
```

This macro starts with summoning monsters around the enemy wizard, then deafening the enemy (so that his spells will fail with 50% probability) and blinding (so that his physical attacks will fail with high probability and his defense is lowered). Upon success, the combination of these spells makes the enemy completely harmless and easy to kill.

Qualitatively, the learned macros have high diversity, although some very similar ones were also generated. "Monster Summoning I" seems to be the single most powerful spell in these duels, as it occurs in 9 out of 15 opening macros. The damage that is caused by the summoned monsters is low, but happens often, and makes spellcasting nearly impossible (a spell fizzles harmlessly if the wizard is hit during casting). It is notable that this rule does not occur in *all* of the macros, which is most likely due to the diversity-rewarding learning procedure. It is also notable that the frequency of the other rules is relatively balanced.

USING INTERESTING MACROS IN ADAPTIVE AI

The purpose of our experiments is to investigate whether the set of new macros can improve the adaptivity and diversity of dynamic scripting in MiniGate, while retaining playing strength.

Description of experiments

Throughout the experiments, the behaviour of the adaptive wizard was adapted by dynamic scripting. The method assigns weights to each rule in the rulebase. In the beginning of a battle, 10 rules are drawn randomly and assembled into a script. The probability that a rule is included in a script is proportional to its weight. After each battle, weights are adjusted, based on the playing strength score F_{str} (Spronck et al. 2006). Dynamic scripting can be easily extended to utilise the macros that were learned for the opening and midgame phases. To this end, we add the macros to the rulebase as extra rules, also assigning weights to them. A script will consist of one opening macro, one midgame macro and ten simple rules.

We compared two systems: dynamic scripting with the basic rulebase (DS-B) and dynamic scripting with the extended rulebase that also contains macros for the opening and midgame (DS-M). The adaptive players were tested against three different static tactics:

- **Summoning tactic**. The wizard summons monsters around his opponent. After that, the wizard throws various offensive spells.
- **Offensive tactic**. The wizard throws a fireball at its opponent and continues with various direct damage spells and disabling spells. This tactic represented strong play in earlier experiments (Spronck et al. 2006).
- **Optimised tactic**. This is a script learned by dynamic scripting (DS-B), when trained against the Summoning tactic. It is similar to the Summoning tactic, but much stronger: when playing against each other, the Optimised tactic wins over 65% of the time.
- **Novice tactic**. This tactic tries to simulate a novice player's tactic, derived from the Novice tactic of Spronck et al. (2006). This tactic was added to test the behaviour of our learning algorithms against a fairly weak opponent.

Except for the Novice tactic, all of these tactics are highly efficient, and, in fact, hard to defeat even for a human player. For each of the 2×4 combinations of the adaptive and static opponents, 50 parallel runs were performed. Each run consisted of 500 battles.

Performance measures

We use three performance measures: (1) the average turning point, (2) the percentage of wins, and (3) the diversity.

The *average turning point*, defined by Spronck et al. (2006), measures the point from which on the adaptive player becomes consistently better than the static player.

The *percentage of wins* measures how strong the adaptive player became at the end of training. We quantify this by counting the battles won by the adaptive player during the last 100 battles out of 500 (the strategies typically stabilised long before the 400th battle).

The *diversity* quantifies how many equally good optima exist to converge to. It is measured as follows. Typically, dynamic scripting does not converge to a deterministic script but rather to a distribution of scripts. We measure the difference between the final distributions of different training epochs. The number of different training epochs is denoted by N. For each epoch $i \in \{1,...,N\}$, consider the last 100 battles. For each rule $k \in \{1,...,M\}$, its frequency of application $p_{i,k}$ is noted. Let $\underline{p}_i=(p_{i,1},..., p_{i,M})$. The diversity D is the average of pairwise distances between the $\underline{p}_i$ vectors:

$$D = \frac{\sum_{1 \le i < i' \le N} \left\| \underline{p}_i - \underline{p}_{i'} \right\|}{\frac{1}{2} N(N-1)}$$

Experimental results

The results of our experiments are summarised in Tables 1 to 3. The DS-B method learns to defeat the Offensive tactic quickly and consistently, and performs reasonably well against the Summoning tactic. It is much less effective against the Optimised tactic. This may seem surprising: in principle, the method should be able to learn a tactic with a win ratio of at least 50% (by selecting the rules constituting the Optimised tactic). However, this happens only rarely because a positive reinforcement happens rarely, therefore the chance is low that the appropriate rules get reinforced.

The trends are the same for dynamic scripting with macros, but the results are uniformly better regarding both the time needed for adaptation and the quality of the learned tactic. DS-M is able to reach a win ratio close to 50% even against the Optimised tactic. Furthermore, DS-M is able to increase the efficiency of adaptation and the win ratio in parallel to an increase in the diversity of learned policies. The results against the Novice tactic are particularly interesting: apart from the slightly worse average turning point and slightly better win ratio, DS-M reached approximately the same diversity level as against the other, stronger tactics. This is in sharp contrast with the diversity loss of DS-B.

To get an interpretation of the diversity results, consider the two extremes: if convergence was always to the same solution, diversity was 0. On the other hand, the diversity of a population that is drawn according to the initial distribution, i.e., each rule is included with equal probability, the diversity measure is 5.82 (note that the rules are not necessarily *applied* with equal probability, because there may be rules that are included in the script but are not executed). In light of these values, we may conclude that DS-M learns considerably more diverse tactics than DS-B.

CONCLUSIONS

We proposed a method for learning diverse but effective macros that can be used as components of game AI scripts. We demonstrated that the macros learned this way can increase adaptivity: the dynamic scripting technique that uses these macros is able to learn scripts that are both more effective and more diverse than dynamic scripting that uses rulebases consisting of singular rules. Our demonstrations were performed in a CRPG simulation with two wizards duelling, but our approach is readily applicable for other script-controlled game AIs. The learned macros can be used either by an adaptive system such as dynamic scripting, or by game developers to speed up the construction of new AI scripts.

Acknowledgements

The first author is supported by the Eötvös grant of the Hungarian State. The second author is sponsored by the Interactive Collaborative Information Systems (ICIS) project, supported by the Dutch Ministry of Economic Affairs, grant nr: BSIK03024. The third author is supported by a grant from the Dutch Organisation for Scientific Research (NWO grant 612.066.406).

	DS-B	DS-M
Summoning tactic	132.3 (0)	105.0 (0)
Optimised tactic	>427.1 (31)	>265.6 (2)
Offensive tactic	42.1 (0)	35.7 (0)
Novice tactic	20.2 (0)	23.4 (0)

Table 1: Average turning points. In parentheses: the number of epochs where the turning point was higher than 500.

	DS-B	DS-M
Summoning tactic	46.7	62.7
Optimised tactic	19.7	44.4
Offensive tactic	70.0	76.2
Novice tactic	94.0	98.4

Table 2: Percentage of wins over the last 100 games.

	DS-B	DS-M
Summoning tactic	3.84	4.70
Optimised tactic	3.63	4.24
Offensive tactic	3.68	4.69
Novice tactic	1.22	4.69

Table 3: Diversities. For a set of scripts drawn uniformly randomly, the diversity is 5.82.

REFERENCES

Boer, P.-T. de, Kroese, D.P, Mannor, S., and Rubinstein, R.Y. 2004. "A Tutorial on the Cross-Entropy Method." *Annals of Operations Research* 134, no. 1, pp. 19-67.

Muehlenbein, H. 1998. "The Equation for Response to Selection and Its Use for Prediction." *Evolutionary Computation* 5, pp. 303-346.

Nareyek, A. 2002. "Intelligent Agents for Computer Games." In *Proceedings Computers and Games*, pp. 414-422.

Ponsen, M., Muñoz-Avila, H., Spronck, P., and Aha, D.W. 2005. "Automatically Acquiring Adaptive Real-Time Strategy Game Opponents Using Evolutionary Learning." In *Proceedings of the 17th Innovative Applications of Artificial Intelligence Conference*, AAAI Press, Menlo Park, CA, pp. 1535-1540.

Rubinstein, R. 1999. "The Cross-Entropy Method for Combinatorial and Continuous Optimization." *Methodology and Computing in Applied Probability* 1, pp. 127-190.

Schmidhuber, J. 1991. "Curious Model-Building Control Systems." In *Proceedings of the International Joint Conference on Neural Networks*, pp. 1458-1463.

Spronck, P.H.M. 2005. *Adaptive Game AI.* Ph.D. thesis, Maastricht University, The Netherlands.

Spronck, P., Ponsen, M., Sprinkhuizen-Kuyper, I., and Postma, E. 2006. "Adaptive Game AI with Dynamic Scripting." *Machine Learning* 63, no. 3, pp. 217-248.

Szita, I. and Lõrincz, A. 2006a. "Learning Tetris Using the Noisy Cross-Entropy Method." *Neural Computation* 18, no. 12, pp. 2936-2941.

Szita, I. and Lõrincz, A. 2006b. "Learning to Play Using Low-Complexity Rule-Based Policies: Illustrations Through Ms. PacMan." *Journal of Artificial Intelligence Research* 30, pp. 659-648.

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

Tozour, P. 2002. "The Perils of AI Scripting." In *AI Game Programming Wisdom* (ed. S. Rabin), Charles River Media, Hingham, MA, pp. 541-547.

Wiering, M.A. 1999. *Explorations in Efficient Reinforcement Learning*. Ph.D. thesis, University of Amsterdam, The Netherlands.