Adaptive combat AI in strategy games - An approach based on Dynamic Scripting

David Lindh

Thesis for a diploma in computer science, 30 ECTS credits,
Department of Computer Science, Faculty of Science, Lund University

Examensarbete för 30 hp,
Institutionen för datavetenskap, Naturvetenskapliga fakulteten, Lunds Universitet

Abstract

When we talk about computer learning in competitive games, we think of an opponent that can learn from its mistakes and use creative new tactics to counter our own. Dynamic Scripting is a method for letting the game AI do just that – and do it in a believable way. The setting for this work is the strategy game World in Conflict. The dynamic scripting agent will be learning how to utilize its units and their different abilities to win tactical battles against different opponents. This thesis aims to show how a list of actions, actions that humans also use to make up their game play, can be combined by the dynamic scripting agent into fully working tactics, capable of adapting to beat its opponents. It will also investigate how the knowledge of such an agent can be used to create a static, well-performing AI. The results clearly show that both the learning agent and the static agent derived from it perform well. I therefore conclude that dynamic scripting works in the setting of tactical combat in a strategy game, both as an on line and as an off line learning method.

Introduktion

När man talar om inlärning hos en datorstyrd motståndare i olika datorspel, tänker man på en motståndare som anpassar sig genom att lära sig av sina misstag och uppfinner nya taktiker för att övervinna spelaren. Dynamic Scripting är en metod som låter den AI-styrda motståndaren utnyttja sina erfarenheter för att lära sig spela bättre. Detta examensarbete kommer att visa hur denna metod kan användas i det moderna strategispelet World in Conflict för att skapa ett trovärdigt och effektivt lärande hos AI-spelaren. Metoden går ut på att AI-spelaren har tillgång till en mängd grundläggande beteenden som kombineras för att skapa olika taktiker, som utvärderas och förbättras för att anpassa sig till den nuvarande motståndarens spelstil. Jag försöker i detta examensarbete visa dels hur denna metod kan användas av AI:n för att genom aktivt lärande hitta ett framgångsrikt beteende i strid, men även hur sådana inlärda taktiker kan användas som grund för utvecklingen av mer traditionella, statiska taktiker. Resultaten visar tydligt att både den dynamiska AI:n med aktivt lärande och den statiska AI:n som härleds från den dynamiska är framgångsrika. Jag drar därför slutsatsen att Dynamic Scripting är en lämplig metod både för att träna upp en traditionell AI och för att skapa en intressant dynamisk motståndare.

Table of Contents

Acknowledgements

Supervisors: Eric Astor and Jacek Malec
Technical supervisor: Johan Pfannenstill

I extend my greatest gratitude to Johan Pfannenstill at Massive Entertainment for the many hours he spent answering my questions and showing me the details of the World in Conflict AI framework. Without his help this thesis would truly not exist. I would also like to thank Eric Astor for his guidance and readiness to answer any questions I had when I was working on this thesis, and Jacek Malec for his invaluable help after Eric retired. Finally, I want to thank Gregor Brunmar, Patrik Hagberth and everyone else at Massive for answering my questions and helping me in different situations.

1. Introduction

1.1. Learning in computer games

Looking back, the first significant computer program that could learn was a checker-playing program written by Arthur Samuel [Russell2003]. It was able to learn good checker play, by combining search methods and updating its evaluations of board positions. Since then many more programs capable of learning to play board games have seen the light of day, and they have grown increasingly good at playing their respective games. Checkers and similar games have large search spaces, but they are not very complex, as the possible actions in a given state are limited. Modern computer games, however, generally have huge complex search spaces of behaviour which makes searching through them very difficult and ineffective. This means successful learning methods that will affect the game play as a whole must make decisions based on abstractions and limited models of the world it can observe. Furthermore it must use its knowledge and experience in such a way that the learned behaviour is undeniably an improvement of the old one. The risk that an AI player[1] might as well learn a behaviour that is worse than the original has been a major concern regarding on line learning for game developers in the past [Woodcock2002], and its general unpredictability seems to remain a problem for the industry [Champandard2007]. Hopefully this will change, and this thesis is an attempt to take a step in the right direction. Another problem with adaptive AI is that it might not even be an attractive feature of a game, in which case commercial game developers are unlikely to implement it in their games, no matter how good it is technically.

1.2. Dynamic Scripting

Dynamic Scripting [Spronck2004] is a reinforcement learning technique [Russell2003] developed for on line learning in computer games. It behaves much as one expects a human to; it plays the game for a while, evaluates its own behaviour, and builds up a tactic from behaviours that it has been successful with so far. If it starts to lose, it quickly adapts and discards current tactics, trying to learn how to deal with the new threat. Dynamic scripting has for the largest part been used for controlling agents in role-playing games, with each character or unit in the game using its own dynamic scripting “brain”. It has also been used to control the strategical decisions of an AI player in a strategy game, such as base building plans, technological research, and amassing an army. A more detailed

1 'Player' in this context refers to an agent that is playing a game. The expression 'AI player' refers to a player that is controlled by an artificial intelligence, as opposed to a human player.

description of the Dynamic Scripting algorithm can be found in section 3.2.

1.3. Purpose

This thesis will investigate the use of dynamic scripting for an AI player in a strategy game, but instead of controlling its strategical thinking it will be operating its tactical skills in combat. The first goal will be to create an AI player that can learn to use its available units in such a way that it can outperform its opponent. The game in question is the strategy game World in Conflict released in September 2007, a game where combat is in focus and such things as resource gathering or technological advancement do not exist. It is all about calling in units - such as helicopters, tanks and infantry - to the battlefield, and winning ground from your opponent through battle. Combat tactics therefore have a very large part in the game, and utilizing every aspect of your arsenal is required to beat the opponents.

Humans playing this game make up their tactics from smaller actions that we know we can use in the game. Examples are moving a unit, ordering a unit to attack another unit, hiding a unit from enemy fire, spreading out your units, scouting with a unit. My first goal is to have dynamic scripting work very similar to a human in this game. It is given exactly those types of actions described above to choose from, and it will test them in different combinations and situations to see which ones are worth using more often and which ones are not. Hopefully, this will amount to tactics that can be considered good, by winning a lot of battles in practice. The second goal is to use the dynamic scripting player as a test agent by letting it develop winning tactics, and then using the actions it has proven successful to create a traditional static AI[2].

2 The expression 'static AI' refers to an AI player that does not learn new behaviour as the game progresses.

2. Problem Definition

The problem consists of two parts.

1. The first part is the question whether or not dynamic scripting can be successfully applied to a tactical combat situation in a computer strategy game, where the AI player rather than a single game unit, such as a rifleman or a tank, is the learning agent. Another way to phrase it would be: Is it possible for an AI player to learn successful combat tactics in a computer strategy game through use of dynamic scripting? First, I will focus on learning to outmatch a particular AI player, and if that succeeds, I will see if the system can adapt to another opponent with different tactics as well. Logically, it would be quite difficult to prove that it is not possible to use dynamic scripting to these ends, so I will simply try to give examples showing that it is.

2. The second problem is to determine, if possible, a way to use the results of dynamic scripting training against a particular opponent to create a static AI which is far superior to said opponent. This is mentioned as one of the possible uses of dynamic scripting [Spronck2006].

The answer to the second question is heavily dependent on the answer to the first one, since if it is not possible to learn well against the opponent in the first place, then the resulting behaviour is not likely to be successful either.

3. Method

3.1. Preconditions

The setting for this thesis is the newly released game World in Conflict (WiC), and the opponents for the majority of the experiments will be the game AI[3] shipped with the game. Each AI-controlled player in WiC has a role; either Infantry, Armor, Air or Support. Each role governs a particular subset of all the unit types in the game, for example, the air player governs the use of all helicopters and the support player handles anti-air vehicles and artillery. The players communicate and ask each other for help, just like a team of human opponents would. I will therefore consider a team of four players, each player filling one each of the roles and commanding the units belonging to that role, to be the opponent in the cases where the WiC game AI is used.

In all experiments each side of the battle will command an equal set of units. While it would be interesting to study the ability to learn in situations where this is not the case, it would simply bring too much risk of imbalance between the fighting sides to be worth the effort, at least for the purpose and time span of this thesis.

In the game there are twenty different unit types, such as heavy tank, medium tank, infantry squad, sniper, heavy attack helicopter, and so on. Some were excluded because the extra effort to include them in the dynamic scripting framework would have taken too much time, or because the game AI did not utilize them properly.

For all the experiments, each side commanded one unit of each unit type, except those excluded as described above. The tests were performed on a certain map, with both sides having the mission of capturing a particular area on the map, resulting in a battle at this location.

3.2. Dynamic Scripting

The behaviour of an agent in dynamic scripting is represented by state-action pairs, where the action is executed only if the state in the pair matches the state of the world at the time of execution. Such a pair might be “If the enemy has helicopters and I do not have anti-air, then we should retreat” or “If the enemy is bombarding the area with artillery fire, then we should spread out”. Using the terminology of Spronck et al. (2004) I will refer to the state-action pairs as rules

3 The expressions 'game AI' and 'standard game AI' refer to the artificial intelligence framework that controls the behaviour of the computer controlled players, also known as 'bots', in the World in Conflict game.

and the generated sequences of those rules as scripts. All the rules are available in a rule base, where they are all associated with a weight value. Each rule is coded by an AI programmer and should produce reasonable and plausible behaviour. This requires some domain knowledge on the programmers side.

For each round of combat, a predetermined number of rules are selected to make up a script, which is a list of rules in a certain order. The probability for any given rule in the rule base to be selected for the script is directly proportional to its weight, and the sum of the weights of all rules in the rule base is always kept the same throughout the learning process. In a round of combat this script is then continuously polled for applicable rules, and the respective behaviours are executed. The implementation of this process is described in greater detail under the section named "Combat Procedure" below.

When combat has ended the result is evaluated by a fitness function, and rules that participated in the script will be rewarded with a weight increase if the battle went well, or a weight penalty, i.e. a weight decrease, if it went poorly. The design of the fitness function is highly domain specific and of great importance to the effectiveness of the algorithm. Once the participating weights have received their weight change, the rules in the rule base which where not part of the script will be compensated in the opposite direction, in order to keep the weight of all sums equal from round to round. This lets the system adapt to a new tactic quickly, since it is sufficient that the currently prominent rules start to lose for the other rules to have their chance of being selected for a script increased.

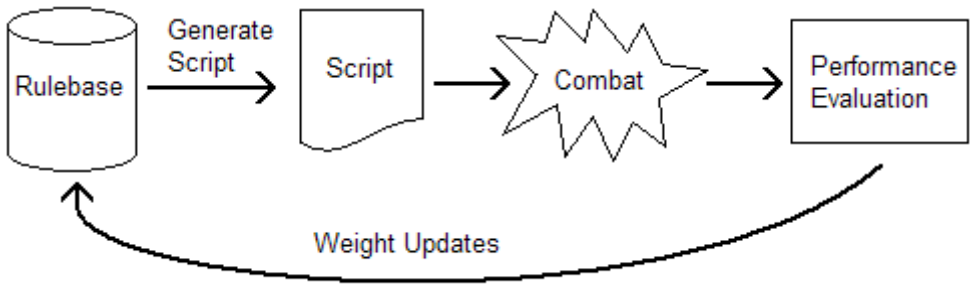

Figure 1: The flow of the Dynamic Scripting algorithm.

3.3. Implementation

The larger part time-wise of my work on this thesis was spent integrating the learning system with the game code, and implementing the rules in the rule base for the dynamic scripting agent. The complete rule base can be seen in Appendix A. The structure of the chain of command was such that during combat, the AI player would give orders with a certain frequency to all its units in accordance with the generated script. This frequency had to be quite high to get an effective behaviour, the reasons for which are explained more thoroughly later on. Below

follows a more detailed description of the different stages of the learning algorithm.

Script Generation

Generating the script was a fairly straightforward task. Before combat was initiated, the script was generated according to the pseudo code in Figure 2 below. A random number between zero and the weight sum of all weights in the rule base was generated, and then mapped to one of the rules in such a way that the chance of selecting a certain rule is proportional to its weight. This was then repeated until the desired number of rules had been selected for the script. For reasons mentioned in the next section, the script was also sorted by rule weight in descending order before the script was returned.

```
script = new List();
while script.size() < WANTED_SCRIPT_SIZE:
      fraction = random(0, rulebase.getWeightSum());
      for each rule in rulebase:
            if (fraction <= rule.weight):
                  if (rule not in script):
                        script.add(rule);
                  break;                        // This breaks the for-loop
            else:
                  fraction = fraction – rule.weight;
return script;
```

Figure 2: The script generation function in pseudo code. A new rule is added to the script repeatedly until the it has reached the desired size.

Combat Procedure

The method with which to give orders to the units during the battle given a script is not part of the dynamic scripting algorithm as such. There were, however, two basic choices made obvious by the game mechanics in question. One could either iterate over the units and give each unit the first order that was applicable to that unit in that situation, or one could iterate over the rules in the script and for each rule give orders accordingly to all units that matched the prerequisites of that rule. I chose the latter, mostly for practical reasons. The major factor in my decision was the fact that I would then have available a list of units for each rule when it was to be executed, rather than having to keep track of who had gotten what order in some other manner. This was convenient for certain behaviours, for example movement orders that were dependent on group cooperation, such as flanking tactics or formation tactics. After a unit had received an order, it was removed from the list of available units for that update, i.e. a unit could only receive one

order per update.

This order of rule execution makes the rule order in the script an important detail. There are several ways to do this, for example by manually assigning a priority value to each rule [Spronck2006]. There is also some research on alternatives for automatic ordering of the execution of rules in such a script [Timuri2007]. I chose the simple automatic method of ordering by weight. This was partly due to lack of time, but also because more sophisticated methods are merely an enhancement of a working system, not part of its requirements, and thus it falls outside the scope of this thesis. In practice this means the iteration order when executing rules starts at the rule with the highest weight and ends with the rule that has the lowest weight, and so whenever a high weight rule is applicable, it is certain that it will execute. This seems fairly reasonable as a high weight should only be acquired by rules that are generally very successful, and consequently those rules should have high priority.

The frequency with which the script should be queried for orders during combat was not obvious from the start. It soon became apparent, however, that an effective AI player would need to deliver new orders to its units quickly as the combat progressed and changed. One example was when a tank was rolling towards an infantry unit and the infantry unit didn't get the order to move out of the way until it was too late. After some testing, I set the order update frequency to twice per second. This created a new problem where orders that had random elements were repeated every 0.5 seconds, which resulted in stale behaviour where no order was carried out to any particular length. One example was with an order to scout around a certain area by moving a random distance in a random direction. Given the high update frequency the unit in question barely moved anywhere in practice, since it repeatedly got new a order which contradicted the recently received one. I remedied this by checking in each update for each new order given if the new order was the same as the last order given to the unit and if the unit was still carrying out an order. If both of those conditions were true, that order was not given again, but the unit was still considered unavailable for other orders this update.

Combat was considered to be over when one of the following conditions was fulfilled.

- Either side has zero troops left.
- More than five minutes has passed since the start of combat.

While the first point is quite reasonable the second warrants an explanation. Some orders would bring the fight to a stale mate, for example an order to flee the battlefield if you couldn't harm any of the remaining enemy units. After some testing I concluded that if a fight was not over in five minutes, it was not likely to end at all.

Performance Evaluation

Evaluation of combat is performed by a fitness function which serves to evaluate the success of the script as a whole, taking into account several factors about the battle. The fitness function I used produced a value in the range [0, 1], where 1 was a perfect win, and 0 was a complete loss. During the course of the testing, I changed fitness function to try to improve the results. This can be seen in the "Experiments" section below, and I will not go into greater detail about it here.

Given a fitness value from this fitness function, the rule weights of the rule base could now be updated accordingly. Below in figure 3 is the formula I used to calculate the weight change for a rule in the script, which is practically the same formula used in [Spronck2006]. I chose this formula as a starting point, with intention to modify it if needed. This proved unnecessary, and it was used in its original form throughout the experiments:

$$\Delta W = \begin{cases} -P_{max}\dfrac{b-F}{b}, & F < b \\ R_{max}\dfrac{F-b}{1-b}, & F \geqslant b \end{cases}$$

Figure 3: The weight change formula.

where ΔW is the weight change for each rule in the script. P_{max} and R_{max} are the maximum penalty and the maximum reward for a single round respectively, limiting ΔW to the range $[-P_{max}, R_{max}]$. F is the fitness value computed by the fitness function, and b is the breakpoint. The breakpoint is the threshold value for the fitness value. A fitness value higher than the breakpoint means the script rules will be rewarded, whereas a fitness value lower than the breakpoint will yield a penalty.

The distribution of weight updates in the function I used consists of four parts, and are presented in pseudo code in figures 4, 5, 6 and 7 below. The first part is to apply the ΔW change to each rule's weight in the script. In [Spronck2006] a modification is used to award the rules that were not actually activated in the fight, but present in the script, only half the weight change of the ones that were activated. This is to allow empty rules to gain and lose weight, and to reduce punishment of rules that may be good but could not affect the battle since they were not activated in this particular round of combat. Empty rules are rules that are never activated. They are present in the rule base to let the script in effect adjust its own size when the preselected number of rules is too high for optimal performance. I decided to use this approach as well.

```
delta_w = CalculateWeightChange();
script_adj_sum = 0;
for each rule in script:
    if (rule.activated):
        rule.weight += delta_w
        script_adj_sum += delta_w
    else:
        rule.weight += 0.5 * delta_w
        script_adj_sum += 0.5 * delta_w
```

Figure 4: Pseudo code for distribution of weight change over the participating rules.

The second part is to compensate all rules that were not in the script, to keep the weight sum of all rules in the rule base constant.

```
compensation = – script_adj_sum / (rulebase.size – script.size);
for each rule in rulebase:
    if (rule in script):
        continue;
    rule.weight += compensation;
```

Figure 5: Pseudo code for weight compensation for non-participating rules.

The final weights are limited by maximum and minimum weight values, W_{max} and W_{min}, and so the third part is to adjust all weight values that exceed these limits and keep track of the remainder this creates.

```
remainder = 0;
for each rule in rulebase:
    if (rule.weight > W_MAX):
        remainder += rule.weight – W_MAX;
        rule.weight = W_MAX;
    else if (rule.weight < W_MIN):
        remainder += rule.weight – W_MIN;
        rule.weight = W_MIN;
```

Figure 6: Pseudo code for clipping the rule weights to their maximum and minimum allowed values.

The fourth part is distributing this remainder among the weights in such a way that it does not make them exceed the limits.

```
fraction = remainder / (rulebase.size * 3);
while remainder not 0:
    for each rule in rulebase:
        new_weight = rule.weight + fraction;
        if (new_weight > W_MAX or new_weight < W_MIN):
            continue;
        else:
            rule.weight = new_weight;
            remainder = remainder – fraction;
            if (remainder == 0):
                break;
```

Figure 7: Pseudo code for distributing the remainder from the weight clipping over all eligible rules.

Once all the weights have been updated and the remainder distributed the rule base is ready for new script generation and a new round of combat.

3.4. Experiments

I conducted a total of seven different experiments. Four of these were actual on line learning sessions for the dynamic scripting player against the standard game AI, and I will call these Learning Experiments 1 through 4 according to their chronological order. One was a reference experiment where no learning was active, and I will call this the Reference Experiment. The other two tried a static AI derived from the best of the four learning sessions against a regular game AI opponent and a dynamic scripting opponent respectively, and I will call them Static Experiments 1 and 2.

The learning experiments and the reference experiment will serve to achieve the first goal of this thesis, answering the question: “Is it possible for an AI player to learn successful combat tactics in a computer strategy game through use of dynamic scripting?”. The first static experiment, where the standard game AI meets the static scripted AI, is meant to answer the second question: “Is it possible to use the results of dynamic scripting training against a particular opponent to create a static AI which is far superior to said opponent?”. The last static experiment, trying a regular dynamic scripting agent against this static scripted opponent, will hopefully shed additional light on both questions.

Throughout the experiments, the initial weight of all rules was 100, W_{min} was 20, and the breakpoint b for the weight update formula was 0.4. For a player to win a battle, it would need to kill all the opponent's units and still have units left, or hold command over the area for which the battle was fought when time had run out. Holding the area meant clearing two perimeter points from enemies, fortifying them with your own troops, and keep the enemy from taking any of them back.

The reason I did four learning experiments is that this is the time it took

before I felt I had a set of parameters and a fitness function which performed adequately.

Learning Experiment 1

For Learning Experiment 1 I constructed a fitness function based on three factors: winning the game, keeping as many units alive as possible, and keeping all units at as high health as possible. From this the fitness value *f* was calculated as follows:

$$f = 0.4 + 0.4F_{units} + 0.2H_{units} \quad \text{if the battle was won, and}$$

$$f = 0.4F_{units} + 0.2H_{units} \quad \text{if the battle was lost}$$

where F_{units} is the number of friendly units alive divided by the number of friendly units alive at the start of combat, and H_{units} is the total health[4] of the remaining friendly units divided by the total health of all friendly units at the start of combat. This function rewards keeping units alive no matter what their health, but also gives a bonus for a low overall health loss. I set the script size to 15, R_{max} and P_{max} to 100 and 50 respectively, and W_{max} to 250. I limited the number of generations for the dynamic scripting agent to learn to 100. Although it seemed that the dynamic scripting agent is learning, as can be seen under "Learning Experiment 1" in the "Results" chapter, it certainly seemed like there was room for improvement.

Learning Experiment 2

For Learning Experiment 2 I constructed a new function to calculate the fitness value *f:*

$$f = 0.4 + 0.4F_{units} + 0.2H_{units} \quad \text{if the battle was won, and}$$

$$f = 0.3E_{killed} + 0.1F_{units} \quad \text{if the battle was lost.}$$

where F_{units} and H_{units} mean the same as above, and E_{killed} is the number of enemy units killed divided by the number of enemy units present at the start of the fight. The reason for changing the fitness function was that while in theory it could assume any value between 0 and 1, in practice it was almost always either zero or something over 0.4. This is because the chances of actually being rewarded for living units and healthy units are quite slim, if the agent did indeed lose the fight.

4 'Health' in this context refers to hit points, which represents how much more damage the unit can take before it is destroyed.

The new fitness function takes into account how well the agent performs under the circumstances that it actually performs badly, which creates a better learning opportunity from lost battles. This function was also used for the remaining experiments.

Learning Experiment 2 used the same values for the constants as Learning Experiment 1, except for the learning rate, which was halved by setting R_{max} to 50 and P_{max} to 25. Since learning rate was halved it also seemed appropriate to double the time from 100 to 200 generations during which the agent could learn. From the results in "Learning Experiment 2" in the "Results" chapter below, it seems that these parameters worked slightly better than in the previous experiment.

Learning Experiment 3

For Learning Experiment 3 I decided to try to let rules that performed particularly well have an even greater chance of being selected in each script generation, so I doubled W_{max}, setting it to 500, while keeping the learning rate and the generation count the same as in Learning Experiment 2. This proved successful, as can be seen under "Learning Experiment 3" in the "Results" chapter.

Learning Experiment 4

I wanted to continue and explore the effects of different parameter values, so for Learning Experiment 4 I changed the script size from 15 to 10 and set W_{max} to 750. I also set the learning rate and generation count back to what it was in Learning Experiment 1, hoping for better results in less time. However, the agent performed worse under these conditions than in Learning Experiment 3, as can be seen under "Learning Experiment 4" in the "Results" chapter.

At this point I settled with the parameters I had used in Learning Experiment 3. Clearly it was possible that there existed better sets of parameters, but it was not feasible to keep testing more when my goal was not to find the best set of parameters but to find a set that worked well enough.

Reference Experiment

For the Reference Experiment, I used a regular dynamic scripting agent, with the alteration of disconnecting the weight updates from the learning algorithm. I let this non-learning dynamic agent fight against the standard game AI, and the results are found under "Reference Experiment" in the "Results" chapter. By disconnecting the weight updates, the only parameter affecting the result was the script size. This was set to 15 for the experiment to compare well to the best learning experiment.

Static Experiment 1

The static AI was implemented by altering the original dynamic scripting agent, changing the script generation function to always return an ordered script with the fifteen most successful rules. To select these rules, each rule was given a score by counting how many times it had had a weight of at least 80% of the maximum weight in the last generation of Learning Experiment 3, the most successful one. The script, which can be seen in Appendix B, was then composed by picking the fifteen rules with the highest score, sorted in descending order.

In Static Experiment 1, the static scripted agent fought the standard game AI. The scripted agent was very successful, and the results are found under "Static Experiment 1" in the "Results" chapter.

Static Experiment 2

For Static Experiment 2, the static scripted AI fought the dynamic scripting agent from Learning Experiment 3, using the same parameter values and generation count as in that experiment. These parameters proved less successful against the static AI than against the standard game AI, but it is clear from the results under "Static Experiment 2" in the "Results" chapter that it adapted and learned a to some extent successful tactic.

4. Results

4.1. Experiment Results

In this section I will present the results of all the experiments I conducted. I will present the win ratios of the agents in each learning experiments. The win ratio of a certain agent, for a certain number of battles, is the number of wins for that agent in those battles divided by the total number of battles.

Learning Experiment 1

This experiment consisted of twelve tests with one hundred generations of training in each test, where the dynamic scripting agent faced the standard game AI.

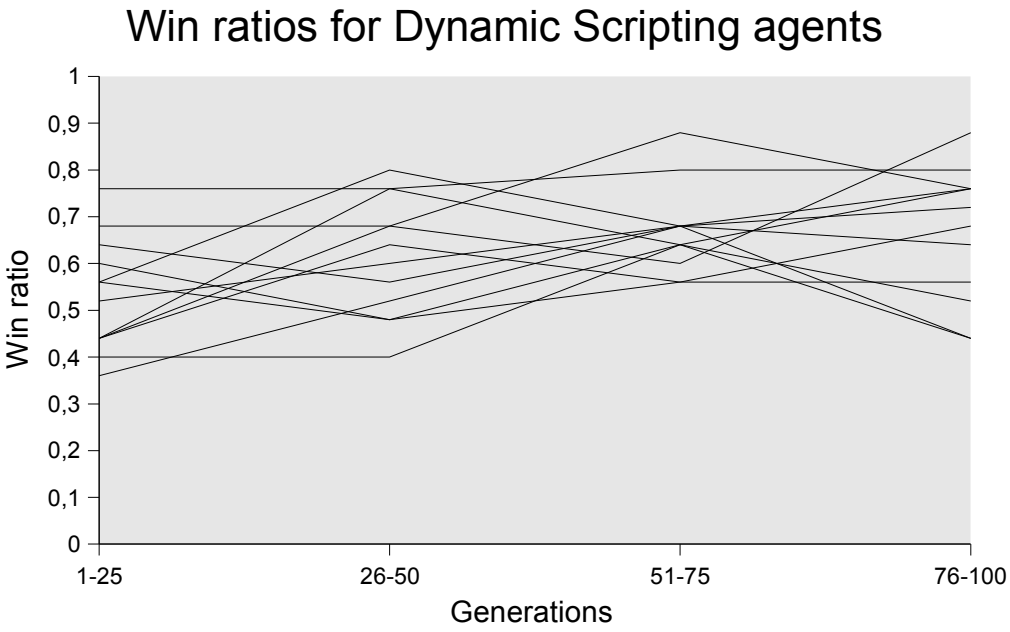

Figure 8: A graph showing the learning development of the twelve Dynamic Scripting agents in Learning Experiment 1. Each line represents one agent.

Generations	*Mean win ratio*	*Standard deviation*
1-25	0.53	0.12
26-50	0.61	0.12
51-75	0.67	0.09
76-100	0.66	0.14

Table 1: A table showing the mean win ratio and standard deviation of the win ratio for all the agents during each generation interval in Learning Experiment 1.

Learning Experiment 2

This experiment consisted of ten tests. Each test had a dynamic scripting agent fighting for 200 generations against the standard Game AI.

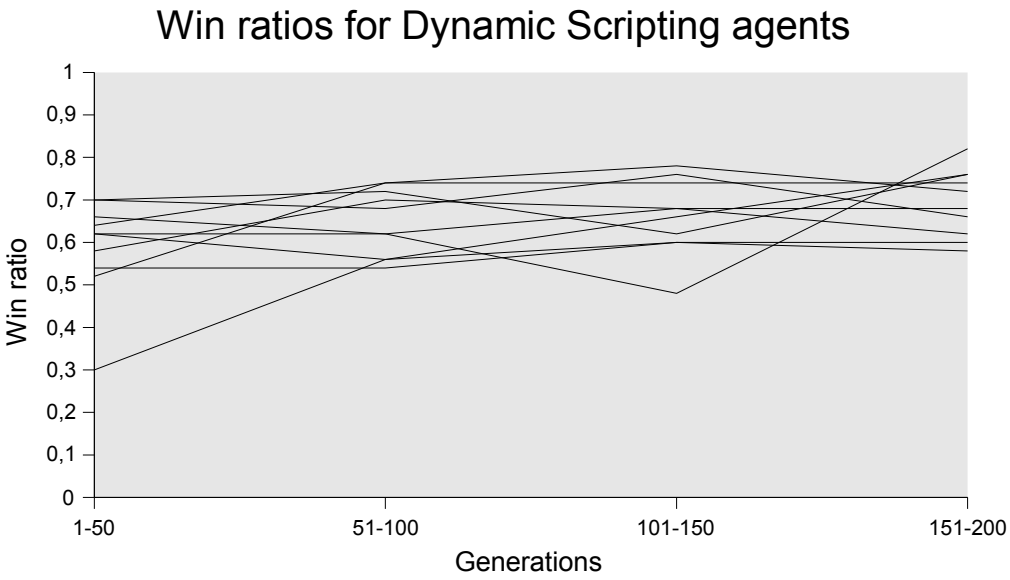

Figure 9: This graph shows the learning process of the ten agents in Learning Experiment 2. Each line represents an agent.

Generations	*Mean win ratio*	*Standard deviation*
1-50	0.59	0.11
51-100	0.65	0.07
101-150	0.66	0.09
151-200	0.69	0.07

Table 2: This table shows the mean win ratio and standard deviation of the win ratio for all the agents during each generation interval in Learning Experiment 2.

Learning Experiment 3

This experiment consisted of twenty tests, each in which a dynamic scripting agent fought against the standard game AI for 200 generations.

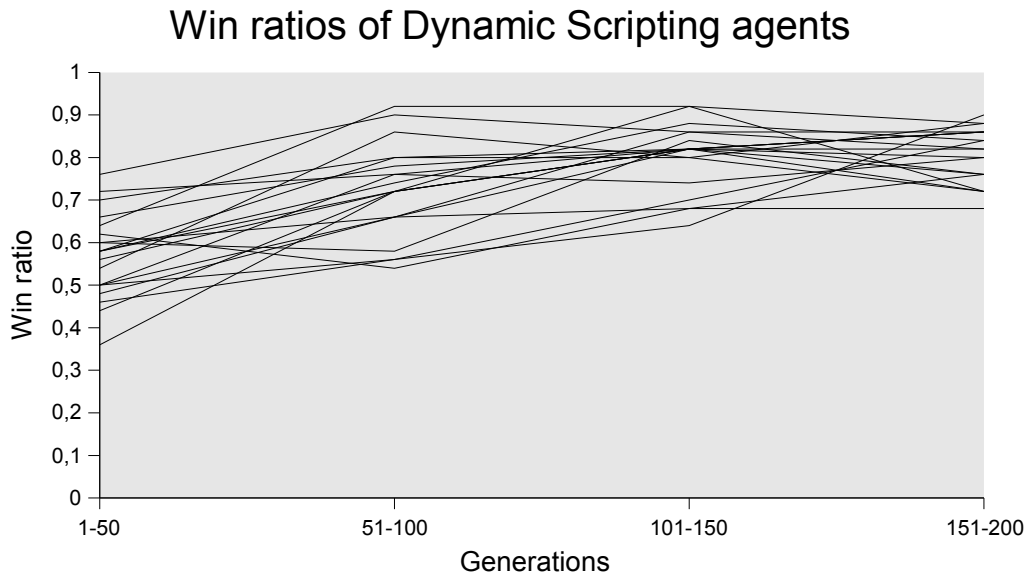

Figure 10: This graph shows the learning process of the twenty agents in Learning Experiment 3. Each line represents an agent.

Generations	*Mean win ratio*	*Standard deviation*
1-50	0.57	0.10
51-100	0.72	0.11
101-150	0.80	0.08
151-200	0.81	0.06

Table 3: This table shows the mean win ratio and standard deviation of the win ratio for all the agents during each generation interval in Learning Experiment 3.

Learning Experiment 4

This experiment consisted of thirteen tests. Each test had a dynamic scripting agent play 100 battles against the standard game AI.

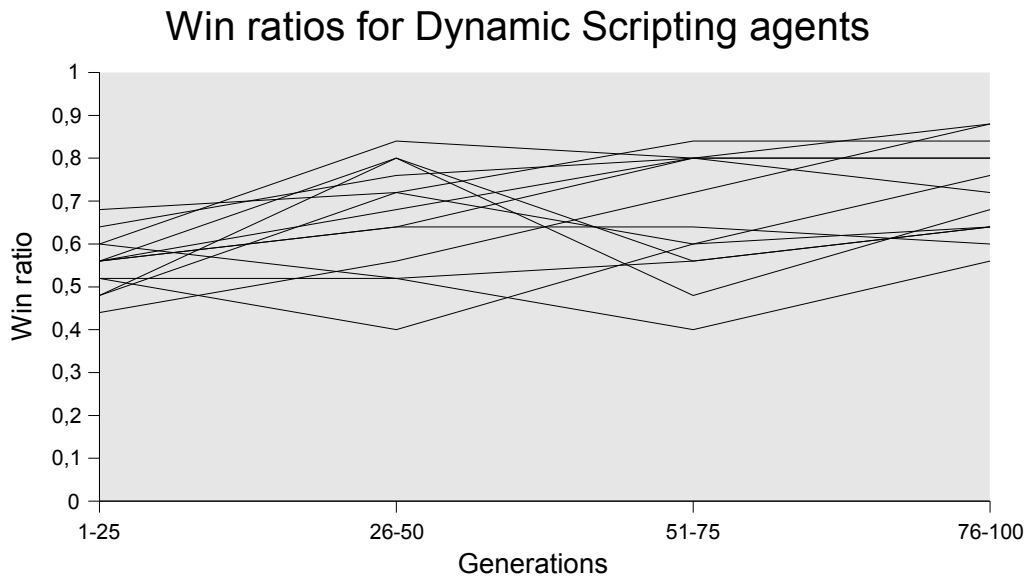

Figure 11: This graph shows the learning process of the thirteen agents in Learning Experiment 4. Each line represents an agent.

Generations	*Mean win ratio*	*Standard deviation*
1-25	0.55	0.06
26-50	0.66	0.13
51-75	0.66	0.14
76-100	0.73	0.10

Table 4: This table shows the mean win ratio and standard deviation of the win ratio for all the agents during each generation interval in Learning Experiment 4.

Reference Experiment

Although no learning progression could be expected, and in fact no change in win ratio over time at all, I decided to test and present this experiment in the same manner as the learning experiments, to make it easier to compare the results. Thus, for the reference experiment twenty tests where conducted with 100 generations in each test. A dynamic scripting agent without weight updates fought the standard game AI.

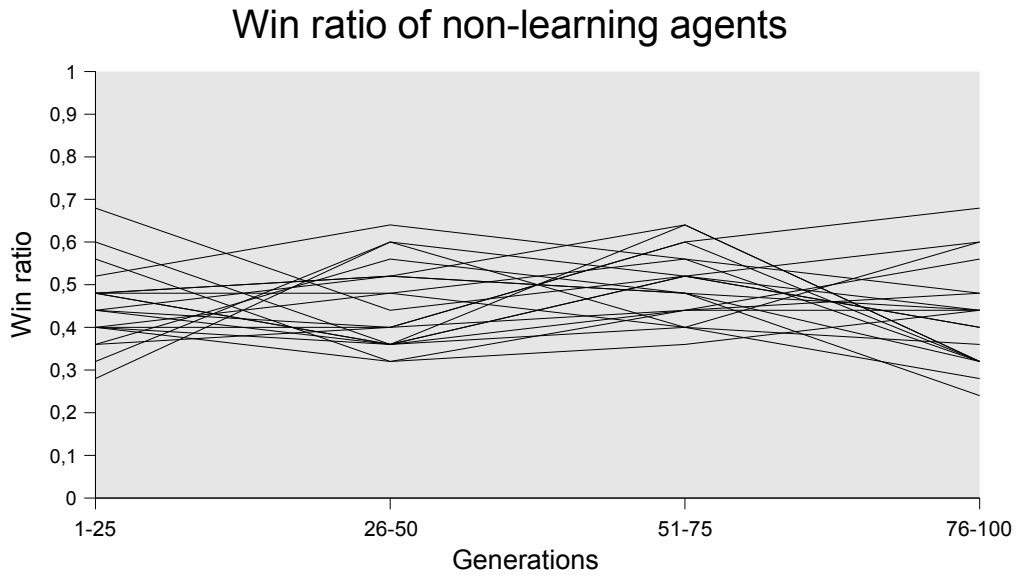

Figure 12: This graph shows the win ratio progression of the twenty agents in the Reference Experiment. Each line represents an agent.

Generations	*Mean win ratio*	*Standard deviation*
1-25	0.45	0.09
26-50	0.45	0.10
51-75	0.50	0.08
76-100	0.42	0.12

Table 5: This table shows the mean win ratio and standard deviation of the win ratio for all the agents during each generation interval in the Reference Experiment.

Static Experiment 1

This experiment was slightly different from the ones presented above. It consisted of five tests of variable length, not being limited to a certain number of generations. This would have had no purpose anyway, since each battle will always start out under the exact same circumstances as the previous one, and is thus independent of all other battles. During these five tests, a total number of 2385 battles were fought between the static scripted agents and the standard game AI. The scripted agents won 2121 of these, which is a win ratio of 0.89.

Agent	*Wins*	*Losses*	*Win ratio*
One	481	73	0.87
Two	404	37	0.92
Three	415	49	0.89
Four	403	52	0.89
Five	418	53	0.89

Table 6: This table shows the wins, losses, and win ratios for the five different static agents in Static Experiment 1.

The standard deviation of the mean win ratio between the agents is 0.015.

Static Experiment 2

This experiment consisted of six tests where a dynamic scripting agent with the parameters from Learning Experiment 3 fought the static scripted AI for 200 generations.

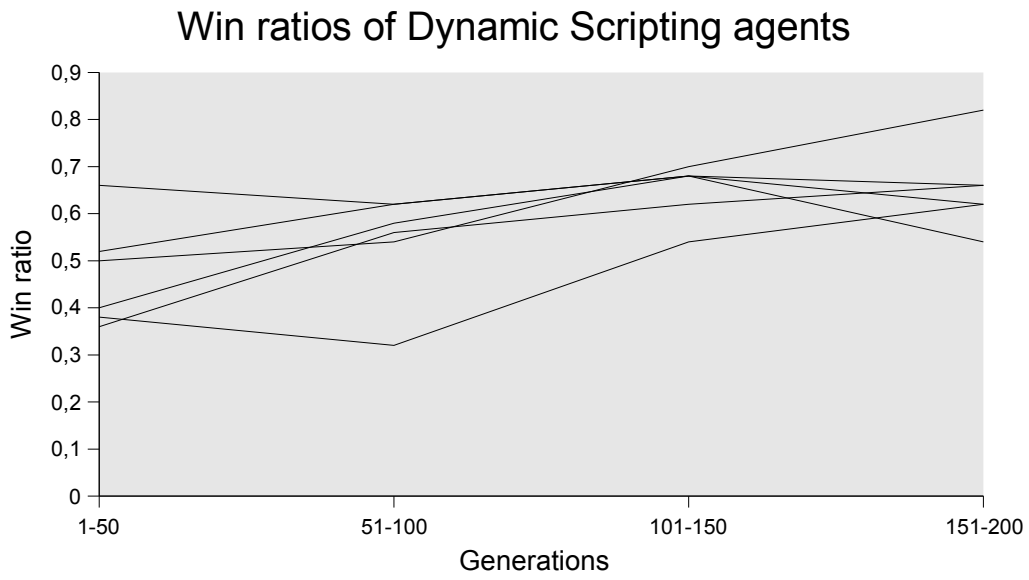

Figure 13: This graph shows the learning progression of the six agents in Static Experiment 2. Each line represents an agent.

Generations	*Mean win ratio*	*Standard deviation*
1-50	0.47	0.10
51-100	0.54	0.10
101-150	0.65	0.06
151-200	0.65	0.08

Table 7: This table shows the mean win ratio and standard deviation of the win ratio for all the agents during each generation interval in Static Experiment 2.

4.2. Comparisons and notes

To compare the different learning results and the reference test I will display the mean win ratios of all the experiments in one graph. These can be found in tables 1, 2, 3, 4 and 5 above. I will use four discrete points for the x-axis for the intervals in each experiment, noting that these points will refer to different generation intervals in reality. For some it will be intervals of 50 generations and for the rest it will be 25 generations. This means that the graph will not be a direct comparison between these results. It will, however, provide us with some clarification about the performance of the agents in each experiment.

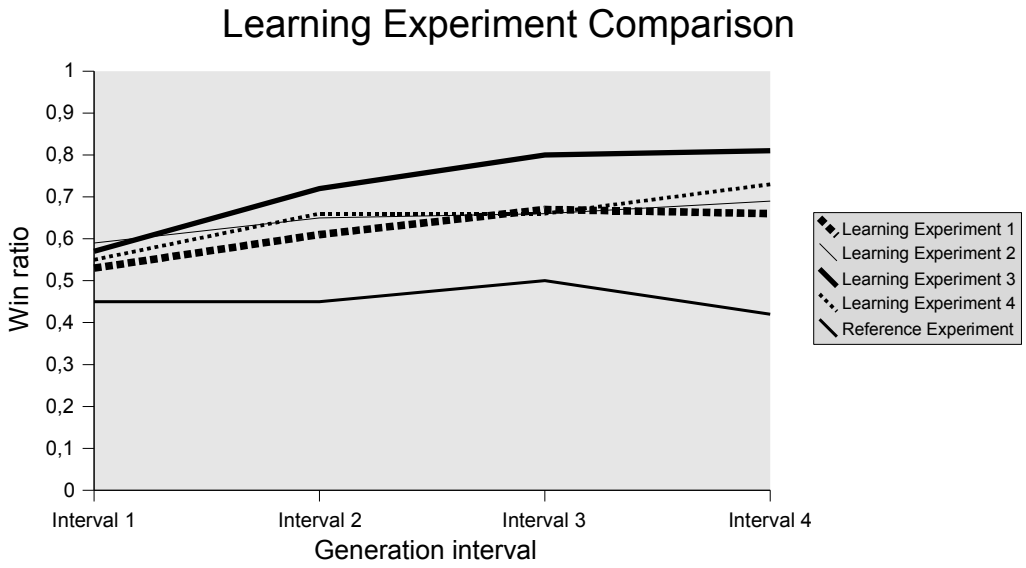

Figure 14: This graph shows the progression of each of the learning experiments alongside each other. The reference test is also included.

From Figure 14 two things are made clear. First, by comparing them with the reference data, we can conclude that all learning experiments were successful in learning a superior combat behaviour. It should be noted, however, that Learning Experiment 4 only used ten rules for the script, and no separate reference test was made with these parameters. The second thing to note is that the agents in the third learning experiment clearly outperformed the others.

From Static Experiment 1, we should notice two things. From the mean win ratio of all the fights it is clear that the scripted agents severely outperformed the standard game AI. We should also note that the standard deviation of win ratios between agents is very low.

Static Experiment 2 shows that a dynamic scripting player with parameters from Learning Experiment 3 can adapt to other tactics than the standard game AI. However, the dynamic scripting agent is not as successful as it was against the

standard game AI.

Below is a graph showing a comparison between the best learning experiment, the static agent versus the standard game AI, and the reference experiment. Since there is no progression data for the static experiment it will be displayed as a horizontal line.

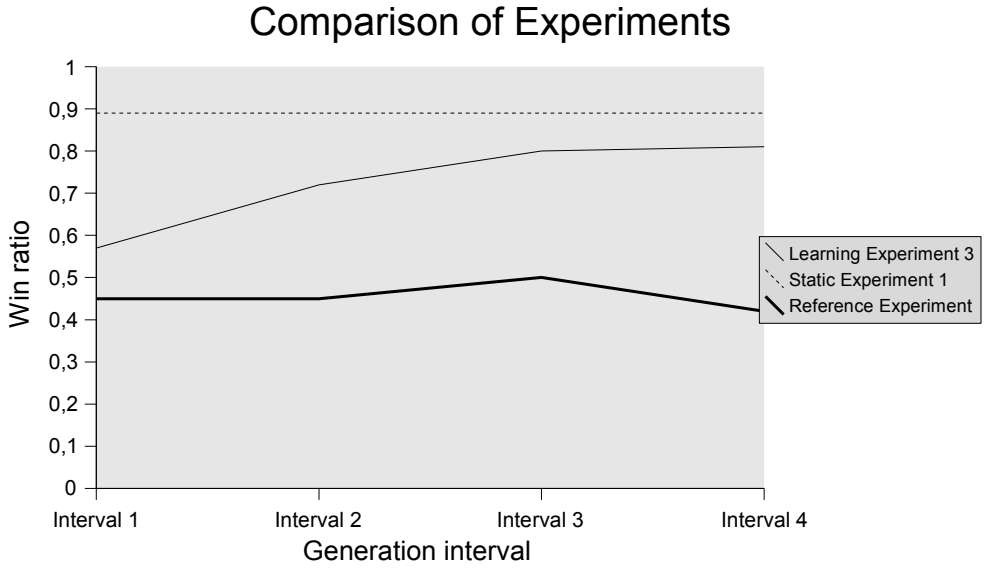

Figure 15: This graph shows a comparison between Learning Experiment 3, Static Experiment 1, and the Reference Experiment.

5. Discussion

The exact results are of course highly game specific and depend much on the game, my implementation of the rules in the rule base, and the implementation of the combat procedure itself. As such, this thesis will serve not primarily as a description of how to implement dynamic scripting in a strategy game, but perhaps to indicate some of the things one must have in mind when trying and as proof that dynamic scripting can be used for this purpose. The results answer the two questions which this thesis aims to cover:

1. Is it possible for an AI player to learn successful combat tactics in a computer strategy game through use of dynamic scripting?

The answer is yes, it is possible. This question is most readily answered by the third learning experiment, in which the agents clearly learned tactics that outmatched the standard AI.

2. Is it possible to use the results of dynamic scripting training against a particular opponent to create a static AI which is far superior to said opponent?

The answer is again yes, it is possible. This is clear when comparing the results of the static scripted AI facing the standard game AI with the results of the reference experiment. The static scripted AI won around 90% of the battles, whereas the random scripted AI won less than half.

One thing that is interesting is that the static AI performed notably better against the standard game AI than the best dynamic scripting agent did in its last generations, even though the first is derived from the latter.

5.1. Implementation weaknesses

The remainder distribution from the weight update function can often lead to a situation where two rules that should have equal weight will not, and in combination with ordering the script by weight this means that one of them will be placed before the other in the script. This is expected with the current implementation, but in conjunction with the fact that the remainder was iteratively distributed it meant that rules with low index in the rule base would at times take precedence in the execution order. This is an unwanted feature and should probably have been remedied with random remainder distribution.

Another possible problem is that most rules are applicable only to a subset

of the possible combat situations that will arise naturally in the game. For the thesis itself I do not believe this was a problem, since the test area and unit composition was such that almost all rules were activated at least at some point during the testing. For more general use, however, the implications of this need to be considered.

Finally, one thing that could be a problem is the fact that the rule base I used did not fully cover all actions that can be taken by the units in the game. For example, maybe a certain special ability of a unit might not have been activated by any rule in the rule base. The reason for this limit was lack of time, and I had to prioritize implementation according to their expected relevance in a fight. I can not without further testing determine the impact a larger rule base would have on the learning speed or effectiveness.

5.2. Improvements

There are several improvements available to enhance the dynamic scripting method, such as penalty balancing and history fallback [Spronck2004b]. Penalty balancing is a method for increasing the speed and effectiveness of the adaptation process by optimising the value of P_{max} compared to R_{max}, in order for the rule base to recover better from states where rules have been unjustly rewarded over the last generations. History fallback means storing each state of the rule base and falling back on previously successful versions when the current rule base seems to be stuck and performance is poor.

Another way to increase the stability and adaptive ability of dynamic scripting is letting the rule base learn the best ordering of rules instead of using a simple weight ordering [Timuri2007]. This is accomplished by using a relation-weights table that keeps track of two values for each pair of rules, representing the two ways to order those rules. A high value for a certain pair in a certain order means that those two rules have a beneficial effect on the battle in that specific order. This table can then be used both to influence the probability of choosing a particular rule in the script generation and for the internal ordering of the rules in the script.

6. Conclusions

This thesis has taken the dynamic scripting method and adapted it to the specific situation of having a commanding agent in a strategy game learning combat tactics using different units on the battlefield to different ends. One thing that I find appealing is that the method was very straightforward to implement, and it did not require extensive testing and tweaking to get it working properly. In fact, it worked fairly well right off the bat. Furthermore, the whole concept of using small building blocks to construct complex behaviours is something I think will appeal to game developers in general, since the rule base, once it exists, does not generally need much alteration, apart from additions when new units and abilities are entered into the game. The rule base can also be used for other learning methods, or perhaps to simply script the wanted behaviour manually.

Initially I was sceptical to whether it could be used as an off line learning technique in the manner we can see in the static scripting experiments above, and indeed there is no guarantee that it has come up with the best possible tactic. It has, however, using very simple statistical methods, produced a static AI clearly superior to the standard game AI and according to Static Experiment 2 an opponent which is to some extent difficult to beat. This static AI can be retrained quickly. In fact, all the tests needed for the agents in the static scripting experiments were completed over one night, using five computers running tests side by side.

6.1. The future

I believe the results of my experiments, while overall very positive, are not good enough to convince game developers in this genre to use it as an on line learning technique. Much work can be done, both with testing different sets of parameters for the dynamic scripting algorithm and with trying the improvements I bring up in the “Discussion” chapter.

However, considering all of the work around dynamic scripting, including methods for generating the actual rule base automatically [Ponsen2006], I think we may not be far from seeing the first examples of games with self-generated adaptive AI, using minimal human input throughout the creation process.

References

[Ponsen2006] Ponsen, M., Muñoz-Avila, H., Spronck, P., Aha, D. (2006) *Automatically Generating Game Tactics with Evolutionary Learning.* AI Magazine, Vol.27, No. 3, pp.75-84.

[Russell2003] Russell, S., Norvig, P. (2003) *Artificial Intelligence – A modern approach*. Pearson Education Inc., Upper Saddle River, New Jersey.

[Spronck2006] Spronck, P. (2006) *Dynamic Scripting.* AI Game Programming Wisdom 3 (ed. Steve Rabin), pp. 661-675. Charles River Media, Hingham, MA.

[Spronck2004] Spronck, P., Sprinkhuizen-Kuyper, I., Postma, E. (2004) *Online Adaptation of Game Opponent AI with Dynamic Scripting,* 2004. International Journal of Intelligent Games and Simulation, Vol.3, No.1, pp. 45-53. University of Wolverhampton and EUROSIS.

[Spronck2004b] Spronck, P., Sprinkhuizen-Kuyper, I., Postma, E. (2004) *Enhancing the Performance of Dynamic Scripting in Computer Games.* Entertainment Computing - ICEC 2004, Lecture Notes in Computer Science 3166, pp. 296-307. Springer-Verlag.

[Timuri2007] Timuri, T., Spronck, P., van den Herik, J. (2007) *Automatic Rule Ordering for Dynamic Scripting.* Proceedings, The Third Artificial Intelligence and Interactive Digital Entertainment Conference, pp. 49-54. AAAI Press, Menlo Park, CA.

[Woodcock2002] Woodcock, S. (2002) *AI Roundtable Moderator's Report.* 2002 Game Developer's Conference, March 21-23.
http://www.gameai.com/cgdc02notes.html (verified March 16, 2008)

[Champandard2007] Champandard, A. (2007) *Apply AI 2007 Roundtable Report.* 2007 Game Developer's Conference, March 5-9.
http://aigamedev.com/coverage/applyai-roundtable (verified March 16, 2008)

Appendix A – The rule base

Following is a list with of the behaviours I programmed and used in the dynamic scripting rule base. The exact meaning of the rules might be hard to extract without having played the game, and additionally there may be minor inconsistencies in the rule descriptions.

Legend

Heli = Any helicopter
HHeli = Heavy attack helicopter
MHeli = Medium attack helicopter
SHeli = Scout helicopter
AA = Any anti-air vehicle
HAA = Heavy anti-air vehicle
MAA = Medium anti-air vehicle
Inf = Any infantry unit
InfSq = Infantry squad
ATInf = Anti tank infantry squad
Tank = Any tank
HTank = Heavy tank
MTank = Medium tank
Transport = Infantry transport vehicle
RepTank = Repair tank

PP = Perimeter Point. These are strategic points located on the battlefield, and capturing and holding them gives a bonus to the player.

'+' means it is owned by me
'-' means it is owned by the enemy
'!' means not, e.g. !+HHeli means “I don't have a heavy attack helicopter”.

Rules

1. IF +Heli AND -MAA THEN kill MAA
2. IF +MHeli AND -HHeli THEN MHeli attack HHeli
3. IF +Heli AND -HAA THEN kill HAA
4. IF +Heli AND -InfSq THEN kill InfSq
5. IF +Heli AND -MHeli THEN kill MHeli
6. IF +HHeli AND -Inf AND Inf in buildings THEN HHeli attack building

7. IF +MHeli AND -MHeli THEN launch sidewinders on MHeli and attack MHeli
8. IF +Tank AND -HHeli THEN kill HHeli
9. IF +Tank AND -ATInf THEN kill ATInf
10. IF +Transport AND -HHeli THEN kill HHeli
11. IF +Transport AND -ATInf THEN kill ATInf
12. IF +MTank AND -Inf THEN Pop WP Shell on Inf and attack Inf
13. IF +Inf AND -Transport THEN kill Transport
14. IF +Inf AND -MAA THEN kill MAA
15. IF +Inf AND -MTank THEN kill MTank
16. IF +Inf AND -SHeli THEN kill Sheli
17. IF +InfSq AND -Inf THEN Use Grenade Launcher on Inf and attack Inf
18. IF +RepTank AND RepTank below 70% health THEN Field repair RepTank
19. IF +RepTank AND RepTank below 50% health THEN Emergency repair RepTank
20. IF +RepTank AND +Tank AND Tank below 50% health THEN Emergency repair Tank
21. IF +RepTank AND +Tank AND Tank below 100% health THEN Repair Tank
22. IF +RepTank AND (MAA OR HAA) AND AA below 50% health THEN Emergency repair AA
23. IF +RepTank AND (MAA OR HAA) AND AA below 100% health THEN Repair AA
24. IF -RepTank AND -Tank THEN kill RepTank
25. IF (-HAA OR -MAA) AND -RepTank THEN kill RepTank
26. IF -few and weak land units THEN kill weakest land units first
27. IF true THEN Move all units around every 5 seconds
28. IF true THEN Move all units around every 15 seconds
29. IF true THEN Move all units around every 30 seconds
30. IF Holding Command point THEN place one unit on each PP (fortify)
31. IF not holding each PP THEN spread out evenly to PPs
32. IF true THEN Move all to one PP but send a weak unit to each other PP
33. IF no friendly unit present in PP closest to attacker THEN move all units toward the PP closest to the attacker
34. IF enemy PP empty THEN move unit to empty enemy PP
35. IF +Inf AND Inf close to building THEN move Inf to buildings
36. IF +Inf AND Inf close to forest THEN move Inf to Forest
37. IF +Inf AND Inf under artillery/TA fire THEN Pop sprint and move away
38. IF +Inf THEN Spread Inf out
39. IF +Inf AND -Sniper THEN Fall back with infantry
40. IF +Inf AND -Tank AND tanks incoming THEN Pop sprint and run “out of the way”
41. IF +Inf AND Inf unit more than 30% casualties THEN refill squad

42. IF +Inf AND Inf unit more than 60% casualties THEN refill squad
43. IF +Tank AND -HHeli THEN Pop smoke screen and stand still
44. IF +Tank AND -HHeli AND +AA THEN Move Tanks to AA
45. IF +Tank AND -ATInf THEN Run over the ATInf
46. IF +Tank AND -ATInf THEN Pop smoke screen and stand still
47. IF +Tank AND -ATInf THEN Flee from ATInf
48. IF +Tank AND -InfSq THEN Run over the InfSq
49. IF +Heli AND -HAA THEN Pull back all helicopters
50. IF +Heli AND -MAA THEN Pull back all helicopters
51. IF +HHeli AND !-Vehicles THEN HHeli scout around area
52. IF +MHeli AND !-Heli THEN MHeli scout around area
53. IF +SHeli THEN SHeli scout around area
54. IF +HAA AND (-HHeli OR -MHeli) THEN Move HAA towards helicopters
55. IF (+HAA OR +MAA) AND !-Heli THEN Fall back with AA
56. IF (+HAA OR +MAA) AND !-Heli THEN Move AA to the front
57. IF +Unit AND Unit under attack but cannot see any enemies THEN approach hidden enemy
58. IF +Sniper AND +Sniper THEN Sniper attack Sniper
59. IF +Inf AND -Sniper THEN kill Sniper
60. IF +HTank AND -HTank THEN Heavy tanks kill HTank
61. IF false THEN No action (empty rule)
62. IF false THEN No action
63. IF false THEN No action
64. IF false THEN No action
65. IF false THEN No action

Appendix B – The Static AI

These are the rules the static AI used, ordered from highest priority to lowest. See the legend in Appendix A for explanations of abbreviations and special characters.

Rules

1. IF +Inf AND -MTank THEN kill MTank
2. IF +MTank AND -Inf THEN Pop WP Shell on Inf and attack inf
3. IF -few and weak land units THEN kill weakest land units first
4. IF +RepTank AND +Tank and below 100% health THEN Repair Tank
5. IF +Inf AND Inf unit more than 30% casualties THEN refill squad
6. IF +HAA AND (-HHeli OR -MHeli) THEN Move HAA towards helicopters
7. IF true THEN Move all units around every 30 seconds
8. IF +InfSq AND -Inf THEN Use Grenade Launcher on Inf and attack Inf
9. IF +Inf AND -Transport THEN kill Transport
10. IF enemy PP empty THEN move unit to empty enemy PP
11. IF +Inf AND Inf close to building THEN move Inf to buildings
12. IF +Transport AND -ATInf THEN kill ATInf
13. IF +RepTank AND (MAA OR HAA) AND AA below 50% health THEN Emergency repair AA
14. IF +RepTank AND (MAA OR HAA) AND AA below 100% health THEN Repair AA
15. IF (+HAA OR +MAA) AND !-Heli THEN Fall back with AA