TECHNISCHE UNIVERSITÄT DARMSTADT

Department of Computer Science
Knowledge Engineering Group

Supervisor: Prof. Dr. Johannes Fürnkranz

Diploma Thesis

Applying Dynamic Scripting to “Jagged Alliance 2”

Author: Manuel Ladebeck

September 2008

Declaration of authenticity

I hereby declare that I wrote this thesis myself without sources other than those indicated herein. All parts taken from published and unpublished scripts are indicated as such.

This thesis has not been previously presented as an examination paper in this or a similar form.

Darmstadt, September 2008 Manuel Ladebeck

Abstract

Many modern computer games are fairly complex, with a vast amount of game states and a high degree of randomization. Most of those games fail to provide an adequate artificial intelligence which leads to the player being more adept than the computer controlled agents after a short amount of time. As a consequence, the developers need to artificially strengthen the computer controlled agents to keep up the challenge. Ultimately the player may loose interest in the game because he feels that the odds are not even. A real adaptive artificial intelligence would go a long way towards immersive digital games.

However the above mentioned characteristics of todays computer games raise some issues when trying to apply standard procedures known from machine learning and data mining. The desired learning processes need to be fast and reliable. An algorithm named 'Dynamic Scripting' has been developed by Pieter Spronck specifically for the deployment in modern computer games.

This thesis covers the application of 'Dynamic Scripting' in 'Jagged Alliance 2', a commercially developed and distributed computer game. After explaining the game's basics and the mechanisms of the learning algorithm, the implemented artificial intelligence is evaluated thoroughly. The emphasis of this paper lies in providing a real world example of implementing an adaptive artificial intelligence in a complex game, along with various suggestions of customizing or combining 'Dynamic Scripting' with other techniques.

Keywords:

artificial intelligence, reinforcement learning, adaptive AI, dynamic scripting, jagged alliance 2, computer games, digital games

Table of contents

List of figures

List of abbreviations

AI	artificial intelligence
AP	action point
FPS	first-person-shooter
HP	hit point
JA2	Jagged Alliance 2
mod	modification (of a computer game)
RPG	role playing game
RTS	real-time strategy

1. Introduction

1.1 Motivation

Digital computer games are a relatively new form of entertainment. The first serious attempt to commercially distribute larger amounts of copies was less than 40 years ago. In 1971 Atari developed a simple ping-pong simulation called 'Pong', arguably triggering the huge boom of the computer games industry.

Since then digital computer games have evolved tremendously. Modern graphic engines require special hardware to cope with the huge amount of calculations necessary for special effects and detailed environments. However, the artificial intelligence of computer-controlled agents has advanced far slower, now significantly lagging behind the visual representation (“It looks like a human, but it does not act like one”).

One common aspect of current game intelligences is the lack of learning processes. The behavior of game agents is static and thus unable to adapt to the player. This leads to very unbelievable characters because in reality everybody – even animals – learn from past experiences. While the player is mastering a game and continually getting better at it, the computer controlled agents always employ the same tactics. As a result, the only way to keep the game challenging is to artificially strengthen the computer controlled agents. This destroys the immersion for the player and ultimately makes him not want to play the game anymore.

There are several scientific approaches to realize learning processes. However, the one major issue with on-line learning systems is that you never know what they learn once distributed. So many game developers frown at the thought of actually shipping their product with an actively learning game intelligence. A procedure called 'Dynamic Scripting' was developed to dissipate these doubts and provide a simple way of implementing learning in modern games.

Basically this paper should be understood as a consolidation of the work of Peter Spronck et al. He was the person who developed the 'Dynamic Scripting' technology and successfully applied it to three different games in the context of his thesis, namely a self-made RPG called 'Mini Gate', the commercial RPG 'Neverwinter Nights' and an open source 'Warcraft 2' implementation called 'Wargus' **[Spr05]**.

Now this paper aims to apply 'Dynamic Scripting' to a slightly different type of game, specifically a tactic-based game called 'Jagged Alliance 2'. Interesting aspects include its turn-based nature and the fact that most of the actions in the game are fairly randomized. If it could be proven that 'Dynamic Scripting' really adds to the performance and flexibility of the game AI and therefore to the entertainment value of the game itself, then this would be another indicator that 'Dynamic Scripting' is indeed a technology which is ready to be deployed in real commercial games.

1.2 Jagged Alliance 2

1.2.1 General information

'Jagged Alliance 2' is a commercially developed computer game which was published in 1999. Like its predecessor 'Jagged Alliance' it was well received by many gamers and earned numerous awards from the specialized press **[JA2awards]**. Some years later two expansions providing additional content were released. The first one called 'Jagged Alliance 2: Unfinished Business' was published in 2000 and the second one (which originally started as a user modification) called 'Jagged Alliance 2: Wildfire' was released in 2004. The latter one also contained the complete source code of the game on the CD.

A community evolved who tried to build upon this code to fix existing bugs and add new content. Derived from the latest official patch, which was version 1.12, a popular modification called 'JA2 v1.13' **[JA2v113]** emerged. The goal of this modification was to keep the game's principles and rules intact by adding new content in the form of new items and polishing the game engine a little. The source code of the 'JA2 v1.13' modification is available to the public and constitutes the base of the implementation of this paper.

Unfortunately the source code released on the 'Jagged Alliance 2: Wildfire' CD was error-prone and could not even be compiled in the form at hand. That was the reason I decided to build upon the 'JA2 v1.13' code instead of the originally released one. It can be assumed that the results of this study could be transferred to the original 'Jagged Alliance 2' game without any divergences; if anything the results should be better, because the game AI was slightly improved in this modification.

Since the 'JA2 v1.13' mod is an active project, the source code gets updated quite often. I went through various revisions during my implementation phase, but I had to stop at one point because merging the new releases with my own changes of the code was too time-consuming. All of the statements in this thesis regarding the 'JA2 v1.13' mod are referring to the revision 2088. The source code is included in the publication of this thesis.

1.2.2 Gameplay

The plot of 'Jagged Alliance 2' takes place on a fictional island called 'Arulco'. An evil female dictator with the name 'Deidranna' conquered the isle and oppressed the people living there. At the start of the game most of the area is under her control. The player controls a group of mercenaries who were hired by a few remaining rebels to reclaim the territory.

The gameplay in 'Jagged Alliance 2' is composed of three different layers, each one representing a task the player has to fulfill. Two of these layers are actually not important in the scope of this study, so their description will be rather brief. I concentrated on the third layer, because it is the core of the game and the player spends most of his time playing on this layer by design.

First of all the player has a limited amount of money in the game. Managing his finances is the first layer of gameplay ('financial layer'). It is possible to use the money for hiring more mercenaries or buying new weapons and other equipment. The game simulates a kind of virtual (and very limited) internet allowing the player to visit certain fictional websites to buy the above mentioned things.

Figure 1.1: *'Jagged Alliance 2' laptop view*

Another task is managing the logistics and troop movements over the island ('strategic layer'). Since the player can control more than one squad of mercenaries he needs to decide where the best positions would be and which part of the island should be freed next. There is a strategic view in the game containing a small map of the island split into quadratic sectors.

Figure 1.2: *'Jagged Alliance 2' strategical sector view*

Finally the last and most challenging task is to control the mercenaries on the tactical screen ('tactical layer'). This is a much more detailed view of one sector, showing the whole environment including buildings, plants and of course the mercenaries. Enemy soldiers or civilians are only visible when one of the mercenaries can actually see them. Basically the player gives movement orders to his troops by clicking with the mouse on the desired location and then they start to walk towards this point. Everything happens in real time while no enemy soldiers are seen.

However if any of the player controlled agents sees an enemy soldier, the game switches to a turn-based mode. In this mode every team can only act during their turn. There are different teams in the game, the most important ones are the enemy team, the player team and the militia team (which is allied to the player's team). A complete round is finished when every team has had its turn once. The sequence of the turns is fixed for a specific battle (i.e. enemy team – player team – enemy team – player team – ...). There is one exception though, at very specific situations actions can be performed 'in between'. This is called an interrupt, but the detailed mechanisms are beyond this basic introduction.

During a team's turn each agent of that team can act in arbitrary order, but only one at a time. There are many kinds of actions agents can perform, including but not limited to shooting at enemies, running, using items, climbing, tossing explosives, etc. But to perform one of these actions the agent needs to have enough 'Action Points' (= APs). APs are a resource that gets refreshed after each round of combat. The amount of APs depends on the attributes and equipment of the specific agent and each action has a different AP cost associated with it.

Figure 1.3: *'Jagged Alliance 2' tactical view*

After each agent of a team has consumed his APs or saved them, which can be useful for various reasons (i.e. hoping for interrupts), the turn is ended and the next team can make its move. There is no time limit for a turn and the player can think about different tactics as long as he wants since the other team cannot do anything during his turn. Of course the computer controlled teams act immediately when it is their turn because the player does not want to wait and get bored.

It should be mentioned that all agents in the game have several skills, expressed as numerical integer values between 1 and 100. Examples of these skills are things like strength, marksmanship, technical adeptness, etc. A high value in a skill means that corresponding actions have a high chance of success (for instance an agent with a high marksmanship will hit his targets more often than an agent with a lower marksmanship).

Another aspect that influences the performance of an agent is his equipment. There are several slots containing different types of items: For example, the head slot can only hold helmets or different types of glasses, whereas the hand slot is designated for weapons and tools. Weapons have an especially large impact on the power of an agent – this is important to consider when measuring the performance of different artificial intelligences.

The artificial intelligence developed within the frame of this study concentrates on the tactical layer for several reasons. First of all, as already mentioned, the player spends at least 90% of his time with this task, making it the core element of the game. Secondly the 'Dynamic Scripting' procedure learns after each 'trial', which is a point in time at which the success of previously used tactics can be determined. Looking at the strategic or financial layer, this is rather difficult to define. Probably the end of the game would be the only definite event with meaningful information, but learning processes after the player has already finished the game are not very useful. On the other hand tactical success can be measured after each battle, providing many more occasions for the artificial intelligence to learn and adapt.

As with every game it is recommended to actually play the game yourself to fully understand the game dynamics and rules. The above mentioned basics should be enough to comprehend the results presented in chapter 5 though.

1.3 Goal of this paper

The aim of this study is to develop an adaptive artificial intelligence for the digital computer game 'Jagged Alliance 2'. It should be shown that the AI performs well enough to actually replace the existing AI in this commercial game. That would be another indicator that the 'Dynamic Scripting' technology developed by Pieter Spronck is indeed ready to be used in modern computer games.

This paper builds upon Pieter Spronck's work and takes his results as given, in particular that the 'Dynamic Scripting' algorithm fulfills the eight computational and functional requirements (speed, effectiveness, robustness, clarity, variety, efficiency, consistency and scalability) **[Spr05]**. Therefore the main task is to show that 'Dynamic Scripting' really improves performance of the agents in this game by making them smarter and able to adapt to consistently employed tactics. Chapter 2 specifies the architecture and the mechanics of the 'Dynamic Scripting' algorithm.

To reach the above mentioned goal, I implemented the 'Dynamic Scripting' algorithm as described in Pieter Spronck's thesis with the necessary accommodations to cope with the mechanics of 'Jagged Alliance 2'. Please refer to chapter 4 for more details about the implementation. Then I ran several evaluations to test the 'Dynamic Scripting' intelligence against the original (static) artificial intelligence. The results are presented in chapter 5.

2. Machine learning in computer games

2.1 Machine learning introduction

The 'Dynamic Scripting' technique is a machine learning algorithm. This sub-chapter is meant to provide a short introduction into the basic mechanics and taxonomies of machine learning processes. Please feel free to skip this part if you are familiar with machine learning basics. The subsequent chapter 3 provides more details about the mechanics of the 'Dynamic Scripting' procedure.

Machine learning is a subfield of artificial intelligence concerned with the development of techniques which allow computer systems to improve their performance over time on a certain task. This goal is reached with various computational and statistical methods, depending on the algorithm. Please note that the field of machine learning is closely related to the field of data mining. There are many different machine learning procedures, each with unique properties influencing the performance on certain learning tasks.

The field of application for machine learning procedures is quite broad. Many real world problems have been successfully approached with machine learning solutions. Examples are medical diagnostics, meteorologic prognosis, natural language processing, handwriting recognition, robot locomotion and many more.

Machine learning algorithms can be classified into distinct categories, depending on the available input and desired output. It should be mentioned that algorithms in the same category may reach their goal in quite different ways.

Supervised learning

There is a set of data available which encodes examples of “how to do it right”. These examples are called 'labeled examples' or 'training data', because a machine learning process is supposed to learn from them. The task is to analyze this data and learn a general concept to deal with new examples. There are different ways to reach that goal and different ways of storing the knowledge. Examples include rule-based approaches, decision tree learners or support vector machines.

Unsupervised learning

There are no labeled examples available. The machine learning algorithm has to discover similarities or differences in the presented data. Typical examples are several clustering algorithms, which try to unite similar data elements together in a group.

Semi-supervised learning

This is basically a mixture between supervised and unsupervised learning. Only a subset of the examples are labeled and the machine learning algorithm combines the information of the labeled and unlabeled data to generate a concept for dealing with new unlabeled examples. For instance the EM[1] algorithm and the Co-learning procedure belong to this type of learning algorithms.

Reinforcement learning

There is no labeled data available, but the learning process receives feedback about the quality of its decisions. In the context of computer games this feedback could be a boolean value (0 meaning a game was lost and 1 meaning a game was won). Artificial neural networks can be used for reinforcement learning; another example of a reinforcement learning procedure is 'Dynamic Scripting', the technique implemented in the frame of this study.

Please note that the introduction above is a very compressed form of knowledge, merely a short overview. For more in-depth information refer to **[Mit97]** or **[Alp04]**.

1 Expectation-Maximization

2.2 Application of machine learning in digital computer games

There are many different learning mechanisms that are potentially applicable for learning tasks in digital computer games. While research on this topic is still in its early stages **[Für01]**, some algorithms have already proven to be quite successful at certain tasks. Especially classic games like chess, backgammon or several card games are popular candidates for this kind of research. For example, there is a backgammon program called TD-Gammon that uses temporal difference learning to improve by playing against itself **[Tes95]**. This program plays at the level of the best players of the world.

However most classic games differ in their nature from modern computer games. While a game like chess is unquestionably very complex, it provides full information at all time. There are no random factors and no hidden events that influence the outcome of the game in any way. An action (which would be a move in the game of chess) has always exactly the same effects when performed in a constant state. And while the amount of game states is huge, it is still finite.

Backgammon on the other side has some common characteristics of typical digital games, for example, it has a certain inherent randomness. Temporal difference learning belongs to the class of reinforcement learning algorithms and basically it would be applicable for the learning tasks in modern randomized games. Unfortunately the standard application of temporal difference learning is off-line learning, which means the whole learning process occurs during development and not while playing against real opponents. While the algorithm itself was actually designed to allow on-line learning, the reason for this circumstance is the huge amount of trials required to effectively learn new tactics with temporal difference learning. The aforementioned system TD-Gammon for example played over 1,000,000 games against itself. Many standard procedures known from machine learning research share this property.

So those methods are not suited for the task in this study, because we want an on-line learning system where the artificial intelligence is able to adapt to the player in a short amount of time. Unfortunately the research in this area has just recently come of age and real examples of games with such characteristics are very rare. One technology which tries to answer those expectations is 'Dynamic Scripting' developed by Pieter Spronck **[Spr05]**. This is the learning algorithm implemented in this study.

It is obvious that reinforcement learning procedures are especially interesting for the application in computer games. Their only requirement is that a feedback function can be designed, which is able to judge if the applied strategies were successful. This is a non-issue in almost any game, because victory conditions are usually well-defined. The learning techniques supervised and semi-supervised on the other hand require data to operate on. For many games there is no such data available. However, some modern games (especially those popular in the e-sport scene) allow the recording of whole matches, which can be re-viewed later on in the game engine. Such a recording contains information about all actions made by a player. An interesting approach would be to analyze data of games made by high skilled players with (semi-)supervised learning techniques.

Unsupervised machine learning algorithms are the least interesting form of learning techniques for the application in computer games. One could image a niche implementation, for example to automatically form groups of agents in some games, but most probably it would be simpler to just encode a few rules to specify the group compositions.

Please note that there are other learning algorithms which may be suitable for learning tasks in modern computer games. For example there are techniques which try to adapt the computer's strategy to the player by observing and analyzing certain key elements of his play-style. This task is called 'opponent modeling', an example can be found in **[SBS07]**. There are other approaches as well, for a more detailed overview of the application of various machine learning technologies in digital computer games please refer to **[Spr05]**.

2.3 'Dynamic Scripting' basics

Just like temporal difference learning, 'Dynamic Scripting' belongs to the class of reinforcement learning procedures. Basically these kind of algorithms work like this: At the beginning they produce random strategies (while of course obeying all game rules) and use them in a match against an opponent. The outcome of this match is fed back into the learning algorithm, which uses this signal to rate the just applied strategy. On a win, all used actions get a higher score, on a loss, their score is decreased. Then a new strategy is generated based on the updated results and once again it is tested in a match against an opponent. Over the course of time this gives high ratings to strategies which are often successful.

Thus reinforcement learning procedures can be applied whenever a reinforcement signal can be collected, in many cases this is just a boolean value where 0 or $false$ encodes a loss and 1 or $true$ encodes a win. Accordingly this means that learning only occurs after a trial has been finished. The definition of 'trial' depends on the specific game. In many cases this is the end of a match, battle or round.

'Dynamic Scripting' has a few additional requirements which are in reality almost always met. First of all, as the name implies, the game AI needs to be available in the form of scripts. This is actually not only common but a de-facto standard in commercial computer games. In former times the AI of all computer controlled agents used to be hard-coded, which means it was implemented directly in the source code of the game itself. This way of programming entailed several problems though. Any change regarding the game AI required a re-compilation of the game. Furthermore many modern games encourage the players to change and add content of the game itself (called 'modding') by providing special tools and externalizing as many parts of the game as possible. Scripts are basically text files encoding the behavior of game agents with *if-then* rules. They either use one of the many scripting languages available (i.e. Lua[2], Python[3], etc.) or come with a custom designed scripting language. Interpreting these scripts during run time provides more freedom for both the developer and the players who are interested in modding.

2 www.lua.org
3 www.python.org

Scripts in the context of computer games – and therefore this study – are an ordered set of *if-then* rules. Each rule consists of one or more conditions (*if* part) and one or more actions (*then* part). Whenever a game agent needs to decide upon an action, the script is called and all rules are checked in sequence. The first rule that returns *true* (meaning that the actions of the rule can be exerted right now in the current game state) gets selected and the agent performs the corresponding action(s).

Normally these scripts are static, which means they are written by a (hobby) developer before the game (or the mod) ships and they don't change while the players play the game. Even though standard game scripts sometimes tend be quite long and complex, they encode a static set of tactics and experienced players can predict actions of computer controlled agents after a certain time of playing. Of course there is always the possibility to encode different tactics and select one randomly, but this doesn't change much. Firstly, the AI is still limited and cannot come up with new tactics and, secondly, the selection of different tactics based on a die roll does not seem very smart.

Now what 'Dynamic Scripting' does is to assign every rule a so called 'weight' value which is an integer expressing the quality of a rule. Based on these weights, scripts are being put together whereby rules with high weights are preferred. This enables the dynamic generation (hence the name) of high quality scripts for basically any given scenario. Scripts (and therefore tactics) are no longer static but rather flexible and able to adapt to even unforeseen game strategies, which meets at least partially the definition of learning. The following chapter goes more into detail about the mechanisms used by 'Dynamic Scripting'

3. Dynamic Scripting

3.1 Mechanics

As mentioned before 'Dynamic Scripting' operates on script files by putting together a set of rules. First of all these rules need to be implemented, which is of course totally dependent on the game. For more information about the rules implemented for 'Jagged Alliance 2' in the frame of this study refer to chapter 4. All implemented rules are registered in a so called 'rulebase', which is an index of all rules and their corresponding weights. Please note that there may be several rulebases, one for every agent class in the game or one for certain game states. An agent class is the generic name for all computer controlled agents in a game sharing the same behavior. For example: In a digital soccer game there would be at least two different agent classes (with their own rulebases): goal keeper and field player. This is because a goal keeper has different actions available than field players (e.g. he can catch the ball with his hands while field players should never do this).

So a rulebase represents a behavior of a type (or class) of players, a script represents the behavior of a concrete instance of that type. The script generation works as follows: First an empty script is generated, containing only a default rule (which specifies an action that can always be performed or an action that is equivalent to 'do nothing'). This is just to ensure that scripts always return a valid behavior. Then new rules are added to the script by copying them from the rulebase and inserting them before the default rule, up to a maximum number of rules (see figure 3.1). The concrete value of the maximum rules in a script depends on the game. Which rules are selected is random but dependent on the rules' weights. Rules with a higher weight have a higher probability of getting selected. Rules with a weight of 0 are inactive and cannot be inserted in a script. However, their weight can possibly increase again over time. More details on the weight calculations will follow in the next sub-chapter.

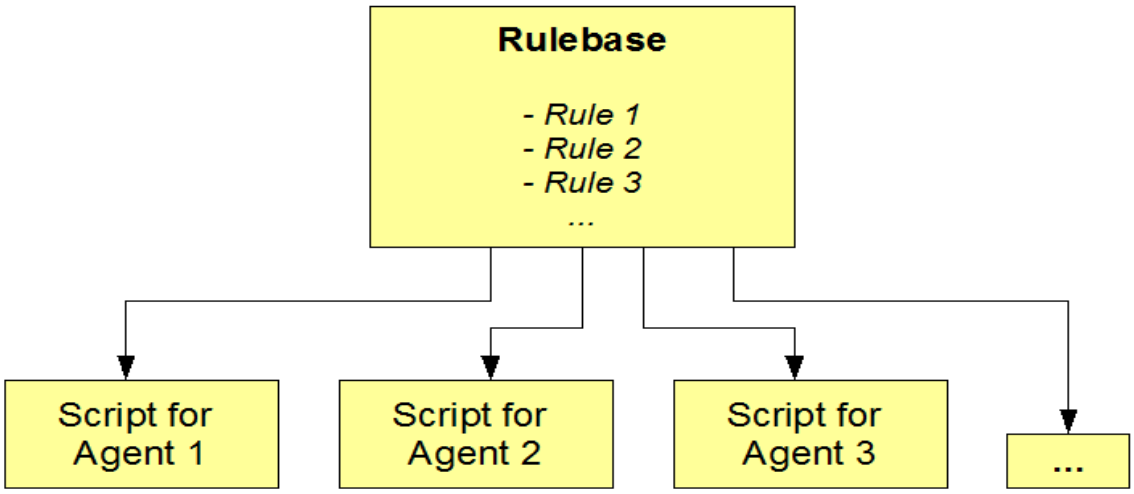

Figure 3.1: *Script generation*

For now it is enough to understand the concept behind the rule weights. They should be an indicator of the quality (or importance) of a rule at the current situation. If a rule has a high weight then the reason is one of the following: 1. The rule was several times in a script which was successful (the agent controlled by this script won) or 2. The rule was not in a script for several runs but the agent(s) always lost the game.

To sum it up: If an agent wins a match then all rules in his script that were actually executed receive a weight increase. All rules that were not activated get a weight decrease. The intuition behind this is: Rules that were executed during a match that was won are probably good, because they led to a victory. Rules that were not executed during a won match are apparently not very important, because the match was won without them.

On the other hand, if an agent lost his match then all activated rules have their weights decreased and all rules that were not activated get a corresponding weight increase. The idea is that the activated rules were not successful, so they are not very good. However, maybe the not activated ones perform better, so their probability to be selected during script generation is increased.

It is important to mention that scripts are generated for each instance of an agent class. For example imagine a game where the player has to fight against a squad of enemies who are all copies of each other with the same attributes and visual appearance. Normally they would be controlled by the same script, but with 'Dynamic Scripting' each one gets his own script. If multiple learning runs already occurred and some rules got a much higher weight than others, then it is likely that the resulting scripts are the same or at least quite similar. In the beginning all rule weights are the same, resulting in more diverse scripts - which is good because this way different tactics are tested at once and the learning process is faster.

There is one more guideline to follow when using 'Dynamic Scripting' in a game: The rules should not encode elementary game moves, but rather more abstract tactics. A rule can only appear once in a script and the first applicable rule always gets selected. So an example of a bad rule would be 'move one square forward'. Even if a successful script could be learned with rules like this, it would be too specific for exactly one situation.

Imagine a tile-based game where two agents fight each other by jumping on the others head. The 'Dynamic Scripting' agent learns to move one square (or tile) forward at the beginning of each match, because he evades the other agent's leap attack this way. Now the starting positions of the agents are changed and suddenly the 'Dynamic Scripting' agent cannot move one square forward, because there is a wall! He has to learn a new script in this case, even though evading at the start is still a good tactic. The developer of this game should rather implement an action like 'evade attack', which always checks for available evade spots in the vicinity. As a consequence, no new learning would be necessary in the above example, because the same script leads to success, even with changed starting positions.

3.2 Rewards and penalties

The rule weight adaption process was outlined in the previous section, this section should provide some more details. We already know that the rule weight is a quality indicator of a rule – the higher, the better. Successful rules get a reward at the end of a fight, failed rules receive a penalty. But how big should these adjustments be? A constant reward for every win and a constant penalty for every loss would not be the optimal solution. It would be much better to scale the adjustments based on how close the outcome of a match was. And this is exactly what 'Dynamic Scripting' does.

The success of each generated script is evaluated with a so-called 'fitness function'. This is basically a mathematical function which generates a value between 0 and 1 indicating how good the script performed during the last match. A 1.0 means perfect performance, the agent controlled by this script played really well. On the other hand, 0.0 means a plain loss, the agent controlled by this script achieved basically nothing. Since computer games tend to be quite different, there is no general fitness function which can be used in every game. Instead one has to be designed for each game. In reality this poses no major problem. Most games have certain subgoals and the fitness function could just return the percentage of completed subgoals or something like the percentage of resources left. Please refer to chapter 3 for the fitness function used in 'Jagged Alliance 2'.

Now, what do we need this fitness value for? It is used to scale the rewards and penalties to the rule weights. There is a standard value called 'break-even point' which represents a kind of neutral point. If the fitness function returns a value equal to the break-even point then the meaning is: The performance of the agent was neither very good nor really bad. As a consequence all rule weights stay the same, there are no adjustments. Again, the concrete value of the 'break-even point' is dependent on the fitness function and therefore on the game. Basically the point lies between the fitness values of the worst winning agent and the best losing agent. So imagine a game where there is a competition between two teams each one consisting of three agents. A fitness function was designed for this game and after a few runs one notices that the winning team's agents have fitness values in the range from 0.6 to 1.0 and the losing team's agents have fitness values ranging from 0.0 to 0.4. One could set the break-even value to 0.5 in such a case. Usually it is in the range of 0.3 – 0.5. Please note that this value is normally identified by the programmer, an automatic calculation would be possible though.

With a fitness value and a break-even point, how does one calculate the adjustments? Basically the difference of these two values determines the magnitude of the weight modulations. As the break-even point lies somewhere in the middle, high and low fitness values (close to 1 and 0 respectively) entail big adjustments whereas values close to the break-even point only bring minor adjustments. A fitness value equal to the break-even value results in no changes at all, as already mentioned (see figure 3.2).

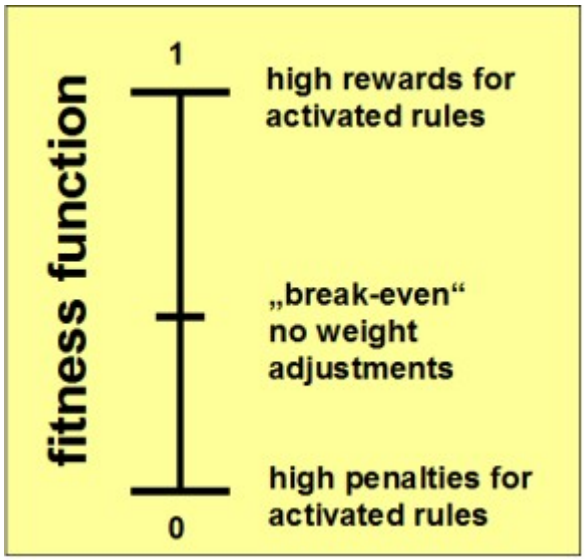

Figure 3.2: *Rule weight adjustments depending on the fitness function*

The 'Dynamic Scripting' algorithm specifies some parameters affecting the weight adjustments and therefore the whole learning process. In addition to the break-even point there are five parameters limiting the rewards and penalties:

R_{MAX} : Maximum possible weight increase per trial

P_{MAX} : Maximum possible penalty per trial

W_{MAX} : Maximum weight for a single rule

W_{MIN} : Minimum weight for a single rule

W_{INIT} : Initial weight for each rule

These parameters influence the learning process. For example, a high value for W_{MAX} causes the algorithm to favor exploitation over exploration. W_{MIN} can be set to 0 to allow actual elimination of rules. If W_{MIN} is set to a high value then even bad rules still have a chance of being selected, again resulting in more diverse tactics (exploration).

So comparing the return value of the fitness function to the break-even point determines whether an increase (reward), a decrease (penalty) or nothing should happen to the activated rules' weights. Then the fitness value is used in conjunction with the corresponding parameter to calculate the actual weight adjustment. The exact formula is defined as follows:

$$\Delta w = \begin{cases} -\left\lfloor P_{MAX} \dfrac{b-F}{b} \right\rfloor & (F < b) \\ \left\lfloor R_{MAX} \dfrac{F-b}{1-b} \right\rfloor & (F \geq b) \end{cases}$$

In this equation Δw is the weight adjustment for each active rule. F is the calculated fitness value and b is the break-even point. The function evaluates to a negative value if $F < b$ (resulting in a weight decrease), a positive value if $F \geq b$ (weight increase) and 0 if $F = b$ (weights remain unchanged).

After all the weights of the active rules have been accommodated, the next step in the 'Dynamic Scripting' algorithm is the revision of all non-active rules. It is important to ensure that the sum of all rule weights always stays constant. This allows rules with a weight of 0 to grow even if they do not appear in a script. So if the activated rules are penalized, all non-activated rules (including all those which were not inserted in a script) receive a weight increase. On the other hand, if the active rules are rewarded then all non-activated rules receive a weight decrease.

Example

There is a rulebase R with the following rules:

1. Rule 1: weight 100
2. Rule 2: weight 100
3. Rule 3: weight 100
4. Rule 4: weight 100

Let's assume there is an agent controlled by the script A, which looks like this:

1. Rule 1 (activated)
2. Rule 2 (activated)

Furthermore the learning parameters were defined as follows:

$b = 0.5$

$R_{MAX} = 50$, $P_{MAX} = 25$

$W_{MIN} = 0$, $W_{MAX} = 300$

Now let us look at two different cases:

1. The agent won ($F = 0.8$)

   $$\Delta w = \left\lfloor R_{MAX} \frac{F - b}{1 - b} \right\rfloor = \left\lfloor 50 \frac{0.8 - 0.5}{1 - 0.5} \right\rfloor = 30$$

   That means that both rules in the script (Rule 1 and Rule 2) get a weight increase of 30, resulting in a new weight of 130. All other rules (Rule 3 and Rule 4) need to get lower weights to keep the sum of all weights constant. Therefore their weight is decreased by 30. Please note that this is not limited by the maximum penalty P_{MAX}, because technically this is not a penalty but a step to ensure the constant weight sum.

2. The agent lost ($F = 0.1$)

   $$\Delta w = -\left\lfloor P_{MAX} \frac{b - F}{b} \right\rfloor = -\left\lfloor 25 \frac{0.5 - 0.1}{0.5} \right\rfloor = -20$$

   Accordingly the rules in the script (Rule 1 and Rule 2) get a weight decrease of 20, resulting in a new weight of 80. The other rules (Rule 3 and Rule 4) get their weights increased by 20 which leads to the same weight sum as before.

Please note that the constant sum of all weights always has priority. This means activated rules cannot gain weight if all non-activated rules are already at the minimum weight.

3.3 Rule ordering

There is one more important aspect when thinking about script generation. It was stated before that the rules in a script are checked in a sequential order and the first one applicable gets executed. This implies that two scripts consisting of the same rules are not necessarily equivalent. Scripts are not just a set but an ordered set of rules. Which raises the question: How does rule ordering work in 'Dynamic Scripting'?

Actually there are different possibilities:

1. Use manually assigned priorities: Basically one fixed value for each rule is used, where rules with the highest priority are checked first, second-highest priority are checked second and so on. This works quite well in some games/scenarios, but sometimes it is too inflexible and does not allow enough adaption.
2. Sort the rules in a script by their weight: The rule with the highest weight is always checked first ('weight ordering'). This approach, while intuitive at first, has some flaws and performs the worst of the automatic rule ordering mechanisms.
3. Learn the rule priorities in parallel with the rule weights. This mechanism is called 'relation-weight ordering' and is the one implemented for the tests in this study. For that reason I want to provide some more details in the following paragraphs. Please refer to the paper **[TSH07]** for more details about the other rule ordering approaches.

It was mentioned that the rule priorities are learned in parallel with the rule weights. This is technically not a complete statement, because an important point is the *relative* rule priority. That means that the algorithm does not learn a single priority value for each rule but rather multiple values that determine the priority over the other rules. As a consequence we need to keep $n \times n$ values in memory, where n is the amount of rules in the rulebase. Accordingly these values are stored in a table (or matrix) which looks like this:

Rule	$\boldsymbol{R_1}$	$\boldsymbol{R_2}$	$\boldsymbol{R_3}$	**...**	$\boldsymbol{R_n}$
$\boldsymbol{R_1}$	–	p_{12}	p_{13}	...	p_{1n}
$\boldsymbol{R_2}$	p_{21}	–	p_{23}	...	p_{2n}
$\boldsymbol{R_3}$	p_{31}	p_{32}	–	...	p_{3n}
...	...	...	...	...	...
$\boldsymbol{R_n}$	p_{n1}	p_{n2}	p_{n3}	...	–

A single entry p_{ij} is an integer and has the following meaning: The rule i occurred before rule j in a script and gained the weight(s) p_{ij} while in that constellation. Analogous to the weights a higher value is an indicator of better performance. Please note while weights are usually limited to natural numbers, the relative priorities can be negative.

If a new script is to be generated, the rule ordering is determined by adding up the relative priorities for each rule per row and sorting the rule by descending priorities. The rule with the highest ordering weight is inserted at the first position, followed by the one with the second-highest ordering weight and so on.

Example

Script A consists of the following rules:

1. Rule 1
2. Rule 2
3. Rule 3

At the end of a match the rule weights get updated. After that, the relative priorities are actualized too, receiving the same adjustments as the rule weights. Let's say the agent with this script lost the game and after calculating the fitness function there is an adjustment of -30 (the agent lost, so the adjustment is negative). All rules in the script, namely Rule 1, Rule 2 and Rule 3 get their rule weights decreased by 30. Then the priority values p_{12}, p_{13} and p_{23} are decremented by 30 as well. Why? Well because in this particular script, the Rule 1 occurred before Rule 2 (p_{12}), the Rule 1 also occurred before Rule 3 (p_{13}) and finally the Rule 2 occurred before Rule 3 (p_{23}). All other values remain unchanged. If the agent would have won, then all these relative priority values would have been increased, because the constellation would have proved to be positive.

[Example continues on next page]

If we assume that this was the first learning run ever, the relative priority weight table would look like this:

Rule	Rule 1	Rule 2	Rule 3	...
Rule 1	–	-30	-30	...
Rule 2	0	–	-30	...
Rule 3	0	0	–	...
...	...	...	...	–

Now during the next learning run, new rules are selected which are inserted in the script. Let's say the same three rules Rule 1, Rule 2 and Rule 3 get selected, which is unlikely because their weights are lower than the other rules but still possible. We calculate the rule ordering weights by adding up all the relative priorities in a row:

Rule 1: $-30+(-30)=-60$

Rule 2: $0+(-30)=-30$

Rule 3: $0+0=0$

From this follows that the resulting script B would look like this:

1. Rule 3
2. Rule 2
3. Rule 1

4. Implementation

4.1 General

The former chapter explained how the 'Dynamic Scripting' algorithm worked in principle. Now I want to shine some light on the concrete implementation in 'Jagged Alliance 2'. Of course all of the aforementioned mechanisms still apply, but a few customizations had to be made. Some are by design, for instance the fitness function or the specific rules in the rule base. These are just concepts in the 'Dynamic Scripting' procedure, their actual implementation is game dependent.

Aside from that there were some problems originating from the code itself. The following sub-chapter deals with one of those quirks, more specifically the issue that 'Jagged Alliance 2' uses no extern scripts for the control of the agents. There is a work-around though which I would like to describe briefly, since this is a problem that could possibly arise in other (especially older) games as well.

To allow full understanding of the actual rule implementations we need to take a look at game states in 'Jagged Alliance 2'. Basically there are different states in which an agent can be, resulting in different rulebases. Again this is something which can be interesting for other games as well.

This chapter also includes all information about game specific elements of the 'Dynamic Scripting' algorithm. First the fitness function, which was designed to represent the core game goals and provide a reasonable estimation about the performance of the agents. And, secondly, the most important part: the rules. Basically the quality of the rules is the greatest performance affecting factor and thus they need to be created carefully.

4.2 Absence of scripts

The 'Dynamic Scripting' algorithm operates on scripts by design. I want to start this sub-chapter with a quotation from Pieter Spronck's thesis **[Spr05, page 131-132]:**

> *“To games that use game AI 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.”*

Now 'Jagged Alliance 2' is such a case where the artificial intelligence is not available in the form of scripts. Instead every behavior is implemented in the game's source code (hard-coded), so a direct application was not possible. It turned out that simulating the presence of scripts was quite simple and straightforward though, and I want to present this solution briefly.

Please note that the following solution is not meant as a design paradigm for game developing. It is but a work-around because incorporating a scripting system and externalizing thousands of lines of source code was beyond the scope of this study. Using real scripts is way more flexible and provides numerous advantages. However, if one wants a “quick and dirty” way of using 'Dynamic Scripting' in games with no scripting capabilities, the subsequent solution may prove useful.

Basically what I did was simulating the concept of scripts with object-oriented mechanisms or in other words: classes. In the following, I will describe the key ideas of my approach: First of all there is a `Script` class, which is in principle a wrapper class for an ordered set of rules, including methods for adding and removing rules (as shown in figure 4.1). Each rule is an instance of the `Rule` class, which is presented in figure 4.2.

Figure 4.1: *UML outline of the* `Script` *class*

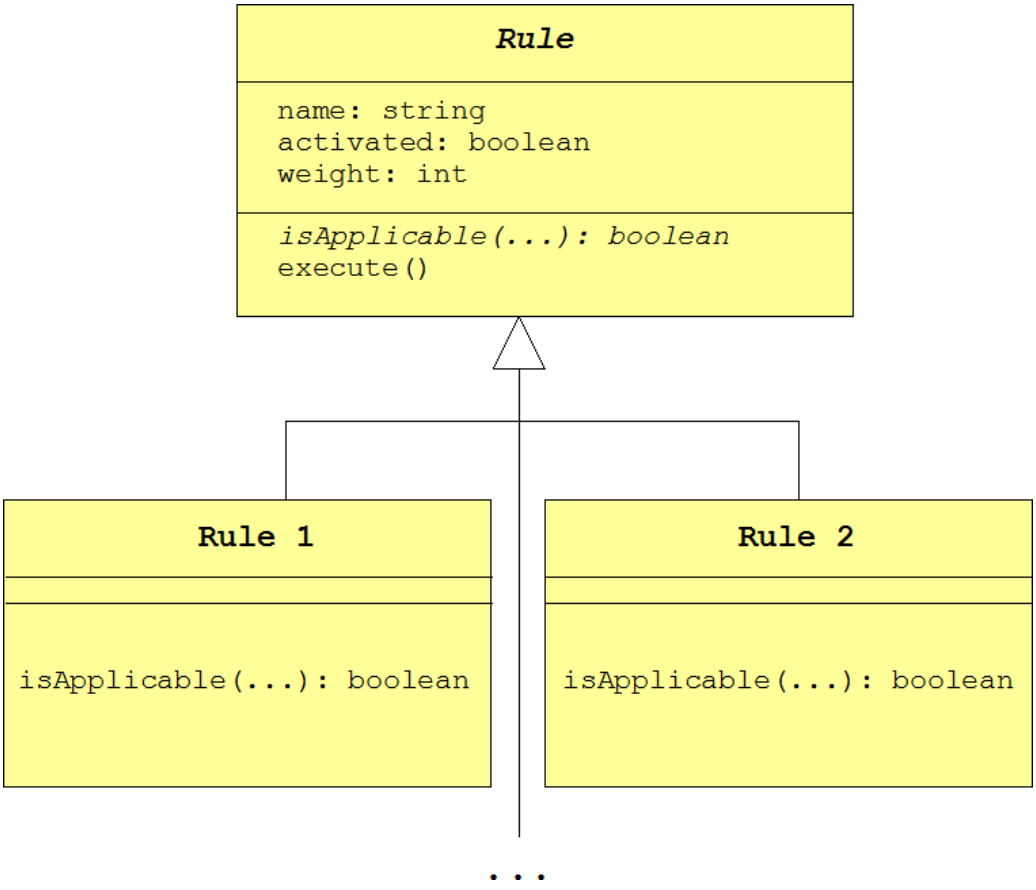

Figure 4.2: *UML outline of the* `Rule` *class*

Please note that the `Rule` class itself is virtual (also called abstract in some programming languages), just like the method `isApplicable(...)`. The `execute()` method does not need to be virtual, because it just returns a variable which encodes the next action. Each rule inherits from the virtual super-class and implements the `isApplicable(...)` method. This method should return *true* if the rule can be executed in the current game state and *false* otherwise; the `execute()` method returns an action code and sets the attribute `activated` to *true*. In chapter 2.3 rules were defined consisting of two parts: The *if* part (conditions) and the *then* part (actions); and this is just what the two methods are modeling.

The figure 4.2 is a simplified outline with a high degree of abstraction. Of course there are other methods, including various Getter- and Setter-methods. As already mentioned, each rule needs to be implemented as a new class. Then it is possible to use the 'Dynamic Scripting' algorithm without any further changes, resulting in the same behavior as using real scripts. I want to emphasize again that this is no “nice” solution (from a programmer's point of view) and was only implemented because of the time constraints of this work.

4.3 Game states and rulebases

Before we attend to the implemented rules and the fitness function, we need to take a closer look to the inner workings of the game first. It is important to get a grasp of how the engine selects an action for an agent, as this has several consequences for the organization of the rules. The original game AI chooses an action based on the alarm state of an agent. When playing 'Jagged Alliance 2' as a player there is no visual indicator for the alarm state of an agent – this is a concept which is only visible in the code. There are four different alarm states:

green	nothing suspicious heard or seen
yellow	heard a suspicious noise
red	high alertness – definite knowledge of enemy presence
black	has line of sight to an enemy at the moment

The procedure handling the selection of an action discriminates over the alarm state; that means that the main procedure dispatches the decision-making to an alarm-specific one. There is one procedure for every alarm state (namely `decideActionGreen()`, `decideActionYellow()`, `decideActionRed()` and `decideActionBlack()`).

After analyzing the code of all those procedures, one comes to the conclusion that some are more interesting than others. More specifically, the procedures responsible for the green and yellow states are fairly simple. The `decideActionGreen()` procedure for instance simulates idle behavior of soldiers, making them walk to random points in the vicinity. Yellow behavior is hardly more complicated, soldiers in this alarm state basically try to investigate the source of the noise they heard. There is little room for variations and the implemented behavior is pretty much the only one meaningful.

As a consequence, I decided to leave the behavior in the green and yellow alarm state as it is. It just made no sense to apply 'Dynamic Scripting' at those states, because the behavior at hand is just fine. Actually the goal of this study is to implement an adaptive combat AI for 'Jagged Alliance 2' and combat hasn't even started when in green or yellow alarm state. One could even argue that the application of a learning procedure at those states is unrealistic, because the game agents should not know about enemy presence yet and therefore their behavior should be easy-going (and not optimized).

Thus 'Dynamic Scripting' is only used for the red and black alarm state. These two states differ in many ways. An agent in the red state knows for sure that an enemy is in the sector and most of the time he has a vague idea where. So basically this state is all about moving. There are several options: Hiding, searching, preparing some other actions, supporting team mates, etc. Some of these actions make sense in the black state as well, but there the focus lies more on actual combat actions: Shooting, throwing explosives or taking cover. The rules were designed to reflect this, please refer to the next sub-chapter for more details.

At the moment it is enough to understand that the two states (red and black) favor diverse actions resulting in different corresponding rules. As a consequence there are two different rulebases, namely a 'red rulebase' (containing all the rules for the red alarm state) and a 'black rulebase' (containing all the rules for the black alarm state). Hence each agent has two different scripts: a 'red script' and a 'black script', which get called in the respective alarm states.

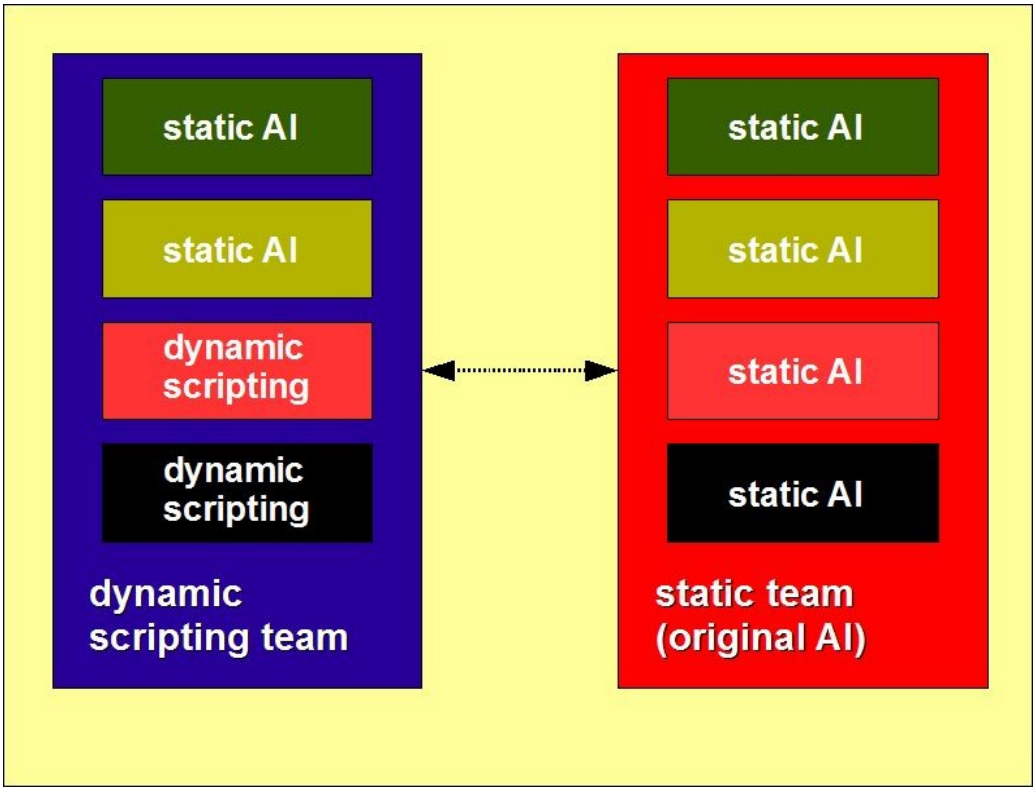

Figure 4.3: *Alarm states and their assigned behaviors*

4.4 Rules

This sub-chapter is meant to provide some details on the implemented rules. Each rule is presented with a name and a short description. I decided to print no full source code here, because it would be too confusing since the implementation turned out rather long. This is because the available AI functions require the programmer to use low-level code such as checking and changing animation states. Instead I opted to show the source code of just one short rule as an example and present the others in the form of tables for a much better readability.

Sample rule: 'TakeCover':

```
// constructor
GeneralBlackRule4::GeneralBlackRule4() : Rule() {
     // name of the rule
     setName("TakeCover");
}

bool GeneralBlackRule4::isApplicable(SOLDIERTYPE *pSoldier) {
     INT16 sBestCover;
     INT8 ubCanMove = (pSoldier->bActionPoints >= MinPtsToMove(pSoldier));
     int iCoverPercentBetter;
     // only try to find a cover if the agent is able to move and under fire
     if (ubCanMove && !gfHiddenInterrupt && pSoldier->aiData.bUnderFire)
          sBestCover = FindBestNearbyCover(pSoldier,pSoldier->aiData.bAIMorale,
                                &iCoverPercentBetter);
     else
          return false;
     // if there is a cover which provides at least 15% more coverage than
     // the current spot
     if (sBestCover != NOWHERE && iCoverPercentBetter >= 15)
     {
          // then forward the movement data to the soldier
          pSoldier->aiData.usActionData = sBestCover;
          setAction(AI_ACTION_TAKE_COVER);
          return true;
     }
     return false;
}
```

Some general annotations on the following tables: Most of the rules require a certain amount of APs (= action points, as explained in chapter 1) to execute their corresponding actions. The check for sufficient APs is always part of the condition(s), even if not explicitly mentioned. If an agent has lesser AP than required by the action(s), then the rule evaluates as not applicable.

The rule names are quoted from the source code hence they consist of capitalized words with no spaces in between. So this is no orthographic mistake but the programmer's way of naming things.

Red rules:

SeekEnemy	
Conditions: ● not heavily wounded ● last action was not to take cover	Actions: ● go towards the position where an enemy is presumed

HelpCompanion	
Conditions: ● an friendly team mate is under fire	Actions: ● go towards the team member

TakeCoverPreventively	
Conditions: ● there is a good cover spot nearby	Actions ● go there, even if no enemy is at sight currently

SaveAPForInterrupt	
Conditions: *this rule has no conditions, it is always applicable*	Actions: ● do nothing, resulting in all APs to be reserved for an interrupt

BroadcastRedAlert	
Conditions: ● nobody else has broadcast a red radio alert before	Actions: ● make a radio call, resulting in all team mates entering the red alarm state (if they are not already at black state)

MoveOutOfDangerousStuff	
Conditions: ● standing in gas, water or light during nighttime ● there is a reachable safe spot	Actions: ● move to that spot

TurnTowardsEnemy	
Conditions: ● enough APs to turn around	Actions: ● turn towards the direction where an enemy is presumed[4]

4 agents in red alarm state have no exact idea where enemies are, they just turn towards the location of the last visual contact or the location of a noise

Black rules:

ShootAtEnemy	
Conditions: ● a target worth shooting at ● weapon with at least one shot left ● halfway decent chance to hit	Actions ● shoot (a single bullet) at the target

ShootBurstAtEnemy	
Conditions: ● a target worth shooting at ● automatic weapon with enough ammunition ● decent chance to hit	Actions: ● switch weapon to automatic fire mode and shoot

Headshot	
Conditions: ● a target worth shooting at which has a head ● weapon with at least one shot left ● very high chance to hit	Actions: ● shoot at the head of the target

ThrowGrenade	
Conditions: ● at least one target worth throwing a grenade at ● no friendly team members in the target area ● a throwable explosive ● decent chance to hit	Actions: ● throw explosive at the target

FightWithKnife	
Conditions: ● a knife ● a target worth stabbing ● half-way decent chance to hit	Actions: ● stab ● or throw knife (depending on the distance)

TakeCover	
Conditions: ● under fire ● there is a spot nearby that offers better cover than the current position	Actions: ● move to the cover

StormTheFront	
Conditions: ● enough APs to move at least one square (smallest unit) ● there is a path to the enemy	Actions: ● move towards the enemy

HideAfterAttack	
Conditions: ● the last action was an active attack with any weapon ● there is a hiding spot nearby	Actions: ● move to the hiding spot

Some conclusive words about the implemented rules: These rules do not raise the claim to be perfect. In fact, when developing a real game, this would be one of the points where heavy optimization is possible. 'Dynamic Scripting' can only build scripts as good as the underlying rules. A bunch of bad rules won't magically combine into a good tactic just because they are put together by a learning procedure. Actually the rules specified above are the result of several revisions and enhancements, but there is still room for further improvement.

Please note that the listed rules were created using my personal domain knowledge of the game and analyzing the existing game AI code. There are approaches of generating rules automatically or semi-automatically using off-line evolutionary learning algorithms (**[PMSA06]** and **[PMSA07])**. Unfortunately these techniques were beyond the scope of this study.

4.5 Fitness function

The task of the fitness function is to express the performance of a script as a number in the interval [0,1]. All weight adjustments scale with this value, so the estimation should be as accurate as possible. A bad performance should always entail a low value and a good performance should be rewarded with a high value. Basically it is important that the core goals of the game are represented in one form or another in the formula.

Now the key element of 'Jagged Alliance 2' is the combat between the player's mercenaries and the enemy soldiers. To win a combat the player needs to defeat all enemies in the sector, which is then flagged as liberated. An enemy is defeated when his HPs (= hit points) are at or below zero. HPs are a common concept in various types of games; basically they are integer values modeling the physical condition of an agent, the lower the more wounded. The maximum number of HPs is dependent on the attributes of the agent. Whenever an agent is hit by a weapon, his current HPs are reduced based on the power of the weapon and numerous other factors (distance, environment, etc.). The actual HP decrease is called damage. Certain attacks inflict enough damage to kill an agent with one hit, others can be completely absorbed by armor, doing no damage at all.

So at the end of the day it all comes down to doing as much damage as possible on enemy agents while at the same time trying to avoid as much own damage as feasible. Achieving this goal is more difficult than it sounds, but at least makes the design of a fitness function quite straight forward.

Actually there are two different fitness functions, namely the team-fitness function and the agent-fitness function. The agent-fitness function is the 'real' fitness function, which is used to scale the rewards or penalties to the rule weights and priorities (as described in chapter 3.2). Technically this one is the only required for the 'Dynamic Scripting' algorithm, but it makes sense to use a second fitness function for more convenience, called team-fitness function. The exact specification follows soon, in principle that function is a simple performance estimation of a team of agents.

Since 'Jagged Alliance 2' is a game where teams (or squads) fight each other by design, the fitness of a single agent is unimportant when measuring performance. What really counts is the success of the whole team. So the agent-fitness function is used to scale the weight adjustments to the scripts, and the team-fitness function is used to measure performance of the teams.

The team-fitness function is defined as follows:

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

In this equation, 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 $h_t\in\mathbb{N}$ is the health of agent c at time t. As a consequence, a 'losing' team has a fitness of zero, while the 'winning' team has a fitness exceeding 0.5.

This is the same team-fitness function as used by Pieter Spronck in his implementation in 'Neverwinter Nights'. While quite different games, the concepts of HPs and damage are the same and therefore the above function is perfectly suitable. Actually it should be adequate for every game where teams of agents try to defeat each other by reducing the opponents HPs.

The agent-fitness function is defined as follows:

$$F(a,g)=\frac{1}{10}\left(5F(g)+2A(a)+2C(g)+D(g)\right)$$

This equation is quite similar to the agent-fitness function used by Pieter Spronck in 'Neverwinter Nights' **[Spr05, page 87]**. The only difference is the introduction of $D(g)$ and the removal of $B(g)$ (explanation following soon). As you can see, it consists of several components with different factors, reflecting their respective weighting. It is obvious that the team-fitness ($F(g)$ as defined above) is a part of the agent-fitness function and even the most important feature, contributing 50% to the final value. The other components are $A(a)$, which is a rating of the survival capability of the agent a, $C(g)$, which is a measure of the damage done to all agents of the opposing team and $D(g)$, which rates the duration of the encounter. All functions return values in the interval [0,1].

The functions used in the agent-fitness function are specified as follows:

$$A(a)=\frac{1}{3}\begin{cases}\dfrac{d(a)}{d_{max}} & \{a\,\text{dead}\}\\ 2+\dfrac{h_T(a)}{h_0(a)} & \{a\,\text{alive}\}\end{cases}$$

In this function $d(a)$ stands for the round the agent died. So if an agent did not survive a match, the function returns higher values the more rounds an agent survived. The parameter d_{max} specifies the total number of rounds passed during the last match. As a consequence, $A(a)$ returns at maximum $1/3$ for a dead agent. Each surviving agent gets an value greater than $2/3$ up to 1, depending on his HPs left.

$$C(g)=\frac{1}{N_{\neg g}}\sum_{c\notin g}\begin{cases}1 & \{c\,\text{dead}\}\\ 1-\dfrac{h_T(c)}{h_0(c)} & \{c\,\text{alive}\}\end{cases}$$

A straightforward implementation of the damage-done function. $N_{\neg g}$ is the number of agents in the opposing team and $c\notin g$ refers to all agents not in the team g. Basically this function returns 1 if all agents of the enemy team were eliminated. Each surviving enemy agent reduces the returned value based on his remaining HPs. The function returns 0 if all enemies are alive and at full health. Again this is the same function as used by Pieter Spronck in his tests with 'Neverwinter Nights' **[Spr05, page 87]**.

Please note that there is no $B(g)$ as in **[Spr05, page 87]**. I decided to not include this element, because it measures the same thing as the team-fitness function, namely the remaining HPs of all team members. Instead I introduced the component $D(g)$, because it became obvious in first experiments, that the learned scripts contained unnecessary defensive rules. This is because hiding and waiting until an opponent enters the own line of sight by accident yielded the same result as taking out the enemy in the first round. Technically this is a valid tactic and definitely in the spirit of the game, but I felt that winning in a very short amount of time should be rewarded at least a little. At the end of the day that is a design decision which has to be made by the developers.

$$D(g)=\begin{cases} 0 & \{g \text{ lost}\} \\ min\left(\frac{10}{d_{max}}, 1\right) & \{g \text{ won}\} \end{cases}$$

The function returns 1 if a fight is won within ten rounds. After ten rounds it slowly scales downwards. A lost fight is always evaluated with 0 in $D(g)$.

4.6 Penalty balancing

The R_{MAX}/P_{MAX} ratio is quite important as it has a significant influence on the learning process. Pieter Spronck argued in his thesis that one has to find a good balance between these values **[Spr05, page 93]**:

"As stated in Subsection 5.2.3, the argument for the relatively small maximum penalty is that, as soon as a local optimum is found, the rulebase should be protected against degradation. However, when a sequence of undeserved rewards leads to wrong settings of the weights, recovering the appropriate weight values is hampered by a relatively low maximum penalty."

I decided to implement a mechanism that scales the maximum penalty P_{MAX} with the number of lost matches in a row. On the first loss the normal value for P_{MAX} is used, on the second loss that value is multiplied with 2 and when three or more losses occur in a row the value is multiplied with 3. As a result a single loss (possibly caused by bad luck) does not lead to the degradation of the rulebase. Multiple losses in a row however are probably an indicator of a wrong tactic, hence a higher value is used for P_{MAX} to allow for faster weight recovering.

4.7 Testing environment

Since 'Jagged Alliance 2' was designed as an interactive game and not as a testbed for AI tests, some changes were required to allow automatic benchmarking. The plan was to let two different AI teams fight each other, where one team should be controlled by 'Dynamic Scripting' and the other one by the original (static) game AI. Unfortunately the engine prohibits to load a sector without at least one player mercenary. However this proved to be no problem at all, because all computer controlled agents act based on their visual and aural perceptions. The (simple) solution was to position the player mercenary at a spot at the edge of the sector where he had no impact at all on the other agents' actions.

Every time the player has his turn, it is automatically ended (technically just skipping the player's team). This allows a single fight to run without any human interaction. A few additional changes were necessary to enable the automatic execution of a batch of matches. Whenever a fight is over (i.e. one of the teams is defeated) the sector is reset to the initial situation. Eventually, after a user-defined number of matches, all rule weights are discarded and the learning starts again at zero. A sequence of fights with no rule weights reset in between is called a 'batch'. The following chapter 5 provides more details of how the evaluation works.

5. Results

5.1 Premises

All of the results presented in this chapter are based on the assumption that a strong playing artificial intelligence increases the entertainment value of a game. Of course a game should not be too challenging right at the beginning, but generally it is always easier to decrease the strength of the artificial intelligence than to increase it. More specifically there are techniques for 'Dynamic Scripting' to adapt to the players skills. Please refer to **[Spr05]** for more details about difficulty scaling.

To measure the strength of the agents controlled by 'Dynamic Scripting' they are competing with agents controlled by the standard (static) artificial intelligence. The standard artificial intelligence is the one implemented by the developers of 'Jagged Alliance 2' with a few enhancements made by the developers of the mod 'JA2v1.13'. At the start of a new game the player has to decide upon four difficulty settings, namely 'Novice', 'Experienced', 'Expert' and 'INSANE'. Please note that the player generally has to fight against superior numbers of similar (or better) equipped agents than his own mercenaries, because the game would be too easy otherwise. Choosing a higher difficulty further increases the enemy numbers while at the same time decreasing the players financial income. The highest difficulty setting even boosts the APs of all enemy agents by about 25%, allowing them to perform more actions per round than the player.

Technically the first three difficulty options are equivalent for the sake of the evaluation. Money and enemy numbers have no influence on the tests, because the soldiers and their equipment are fixed in the test scenarios. The fourth option would make a difference though, because more APs for the enemy soldiers does noticeable impact the performance of them. However it would be an one-sided advantage which complicates the comparison of the respective tactics. As a result the difficulty level 'Expert' was used throughout the tests.

As you can see 'Jagged Alliance 2' is a good example of how games need to apply 'cheating' mechanisms to keep up with the player. Technically the enemy soldiers are equal to the players mercenaries, yet they appear in manifold numbers to make up for their lack of variant tactics. To provide a concrete example:

New players are going to notice quite fast that hiding in a building is a superior tactic. Enemy soldiers will spend their APs in opening the door of the building and walking into line of sight of the players mercenaries. This leads very often to an interrupt for the player, allowing him to take down the enemy agents one by one. The second of the following test scenarios was designed with this knowledge in mind.

As mentioned in chapter 4, the performance of the teams (and therefore the AI controlling the team) is measured with the team-fitness function, which again measures the remaining HPs of all agents in a team. This gives a quite good estimation on how close a match was. However, the outcome of a single match has no real significance. Please keep in mind that the game is strongly randomized, so winning by chance is possible even with an inferior tactic. To make a statement about the general performance of an AI, one should look at the average results over a large number of matches.

Hence I decided to average the results in a similar way as Pieter Spronck did in his tests with 'Neverwinter Nights' **[Spr05, page 90]**. Each of the following scenarios looped through 10 batches à 100 matches, resulting in 1.000 runs per scenario. The rule weights are reset after each batch just as the relative rule priorities, which means that all learned tactics are discarded and the learning restarts at zero. Please note that 100 matches are a very low value for machine learning methods. This number of trials would not be sufficient for most standard algorithms, but 'Dynamic Scripting' was designed to learn from a small number of encounters.

How do all these values combine into a single graph? First, the team-fitness values for each encounter are averaged across the batches. This results in 100 average values, one for each encounter. Then we summarize the outcomes of the last matches in a size-10 window, which means the results of the last ten matches are averaged for each encounter. As a consequence the first possible point can be calculated after the first ten matches, averaging the values from match 1 till 10. The second point is the average of the matches 2 till 11, and so on. This results in 91 final data points.

The result graph for each scenario shows the number of passed encounters on the x-axis. A point on the y-axis is the average size-10 team-fitness value calculated as mentioned above. For reference I decided to plot a non-averaged version of the graph as well, which is colored grey.

Example

Let us assume we want to evaluate 2 batches à 12 encounters (I chose lower numbers because the example would take up too much space otherwise). Here is a fictional result table with some invented team-fitness values.

Encounter	Batch 1	Batch 2
1	0	0
2	0.4	0
3	0.4	0
4	0.6	0.5
5	0.7	0.4
6	0.8	0.8
7	1	0.9
8	1	0.9
9	1	0.8
10	1	1
11	1	1
12	1	1

So first we calculate the average values for each encounter:

Average for encounter 1: $\frac{0+0}{2}=0$

Average for encounter 2: $\frac{0.4+0}{2}=0.2$

Average for encounter 3: $\frac{0.4+0}{2}=0.2$

Average for encounter 4: $\frac{0.6+0.5}{2}=0.55$

... and so on

Doing this for every encounter leads to the following table:

Encounter	Team-fitness average
1	0
2	0.2
3	0.2
4	0.55
5	0.55
6	0.8
7	0.95
8	0.95
9	0.9
10	1
11	1
12	1

Then we construct the 10-size average values for each encounter. Since the first possible point of calculation is the tenth encounter, we get just three final values in this example:

10-size average fitness value, encounter 10:

$$\frac{0+0.2+0.2+0.55+0.55+0.8+0.95+0.95+0.9+1}{2}=0.61$$

10-size average fitness value, encounter 11:

$$\frac{0.2+0.2+0.55+0.55+0.8+0.95+0.95+0.9+1+1}{2}=0.71$$

10-size average fitness value, encounter 12:

$$\frac{0.2+0.55+0.55+0.8+0.95+0.95+0.9+1+1+1}{2}=0.79$$

Our final tables looks like this:

Encounter	10-size average fitness value
10	0.61
11	0.71
12	0.79

This is the table used to create the result graph.

The following values for the 'Dynamic Scripting' learning parameters were used:

$W_{INIT}=100$: The sum of all initial rule weights specifies the available 'pool' of weight points that can be re-distributed.

$W_{MIN}=0$: This allows the elimination of useless rules, because rules with a weight of 0 have a probability of 0 of being inserted in a script.

$W_{MAX}=500$: This means that a rule cannot gain enough weight to leave all other rules at 0. Thus this is a rather explorative approach. Please note that a higher value could even improve the following results, because the agents would more consequently exploit successful tactics. However those tactics tend to be quite specialized and very dependent on one rule. While such a script is optimal for one scenario, it generally does not work in other scenarios. So this setting provides a good balance between exploration and exploitation.

$R_{MAX}=100$

$P_{MAX}=25$: At first the value of P_{MAX} may seem (too) low, but please keep in mind that this value can scale up to 75 if multiple matches are lost in a row (as described in chapter 3.6).

Both the red and the black script of each agent was limited to 4 rules.

One more note about the evaluation: The total runtime for a scenario was about 2 days (48 hours) on average. This is because there was no feasible way to turn off the graphical presentation. Each soldier's action had to be animated along with a delay after each move. Therefore even a fast CPU could not speed up the process, because of the fixed delays. While this is reasonable if a human player plays the game, it proved detrimental during the evaluation.

5.2 Scenario 1

This is the main test scenario, which was designed to provide an objective evaluation environment. The 'Dynamic Scripting' team fights an enemy squad of equal strength. All agents (blue and red) possess the same attributes and equipment. Please refer to Appendix B for more details about the configuration. Both the blue and the red team consist of five soldiers. They are positioned with equal distance between them. Each team has a group of tightly planted trees close-by. These trees are an important part of the scenario, because they are the only cover in the vicinity.

The task for the 'Dynamic Scripting' team is to gain the upper hand against the identical equipped enemy squad. Since both teams are of equal strength such a success could only be caused by a superior tactic.

Figure 5.1: *Layout for scenario 1*

Here are the results:

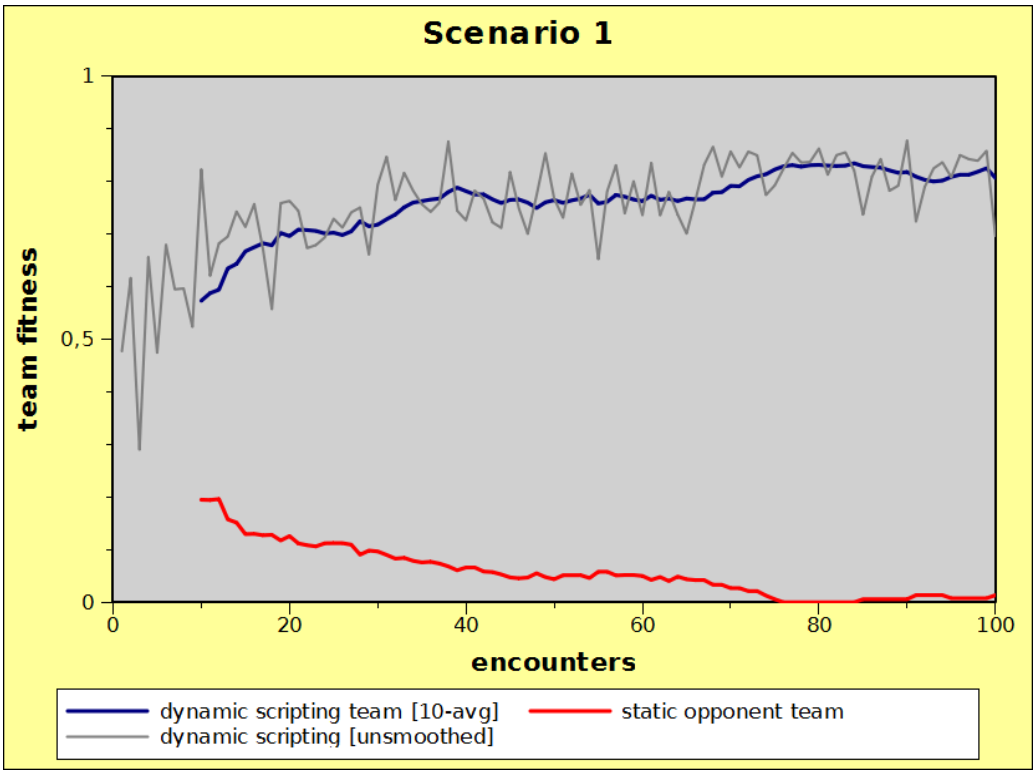

Figure 5.2: *Team-fitness progression for scenario 1*

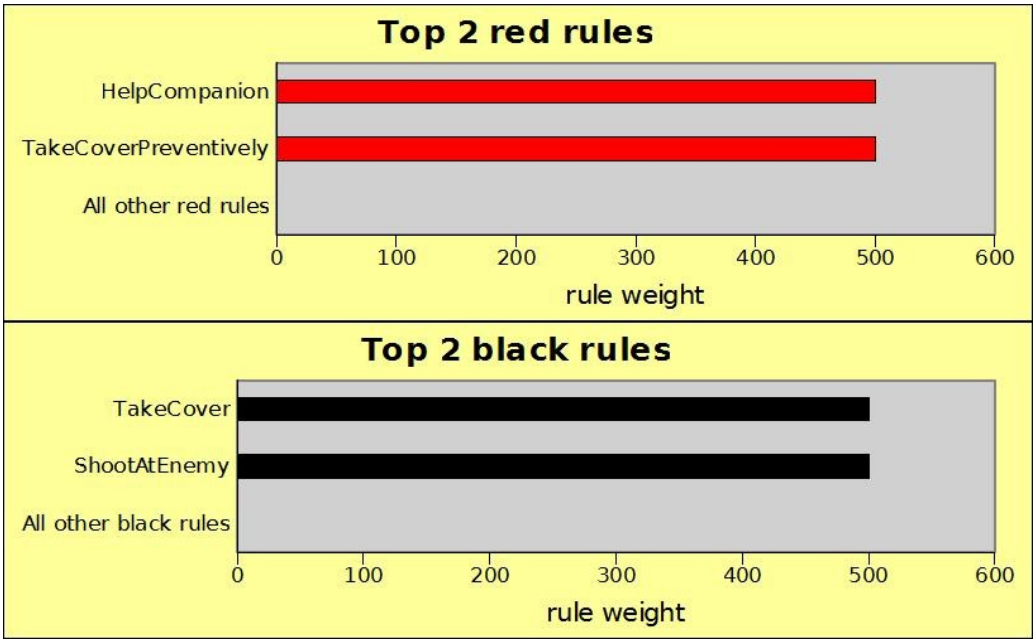

Figure 5.3: *Top rules for scenario 1*

Discussion:

The 'Dynamic Scripting' team consequently exploited the weak points of the static AI. First of all, they didn't use the 'ShootBurstAtEnemy' rule, because the ammunition was fairly limited in this scenario. The enemy however often fired bursts at long range, which is quite ineffective, because the accuracy is heavily decreased and they ended up with no ammunition in long fights. Furthermore, they learned to use the cover provided by the trees and they recognized that staying close together was an advantage in this scenario.

The beginning of each match looked fairly balanced. Both teams stayed in cover and tried to score some long range hits. However, the static agents select an action based on their actual state and on random factors. Basically they decide between attacking and taking cover based on a dice roll. So after some turns (amount varies) one of the red agents always decides that is time to storm the front. Since the 'Dynamic Scripting' agents are standing close together, they all get a free shot at this red agent, most of the time resulting in his death.

The point at which the 'Dynamic Scripting' team was significantly better than the red team is very low. However, the curve itself is not very steep. This is because two of the batches were a lot worse than the others, with turning points of 27 and 66 respectively. The other batches all had turning points in the range of 0-10, most even below 5. Pieter Spronck identified this phenomenon as 'outliers' and proposed solutions for their reduction. An enhancement called 'history fallback' is available to engage this issue, for more information please refer to **[Spr05, page 92-97]**. This approach was not realized in the implementation in this study, skip to chapter 6 for more suggestions of improvement.

Looking at the learned rules, they are straight to the point. Taking cover is very important in both the red and the black alarm state. The enemy is outlasted by hiding behind the trees and scoring long range shots.

5.3 Scenario 2

This test scenario is a little bit different from the first one. The 'Dynamic Scripting' team faces a learning task where only a specific tactic leads to success. Here is the basic setup:

The red team, consisting of four soldiers, is heavily armed and armored. Each member of the red team wears one of the best bullet proof vests in the game and possesses high damage automatic weapons. They are standing close together in a barn guarding a barrel of fuel. The blue team on the other hand only consists of two members, wearing light camouflage equipment. Each blue agent holds a light pistol with a silencer, which is unable to penetrate the red teams armor efficiently. To make up for that lack of firepower both of the agents carry multiple grenades with them.

Due to the camouflage suits the red team gets no initial interrupt when the blue agents first enter their line of sight. The only way of winning for the blue team is to throw as many grenades as possible on the red team, which causes the barrel of fuel to explode. No other tactic works here, because the armament of the red team is way superior. Even headshots would not do enough damage to eliminate more than one or maybe two red soldiers. The red team could still take out both blue soldiers in one round with ease.

Figure 5.4: *Layout for scenario 2*

Here are the results:

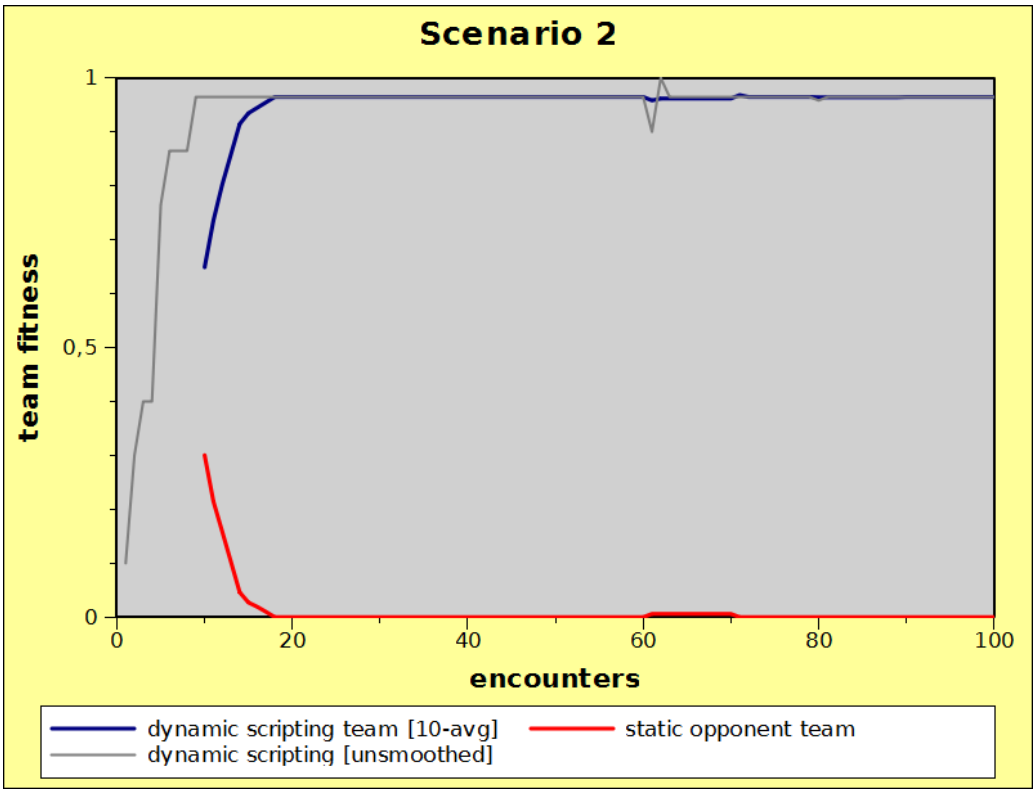

Figure 5.5: *Team-fitness progression for scenario 2*

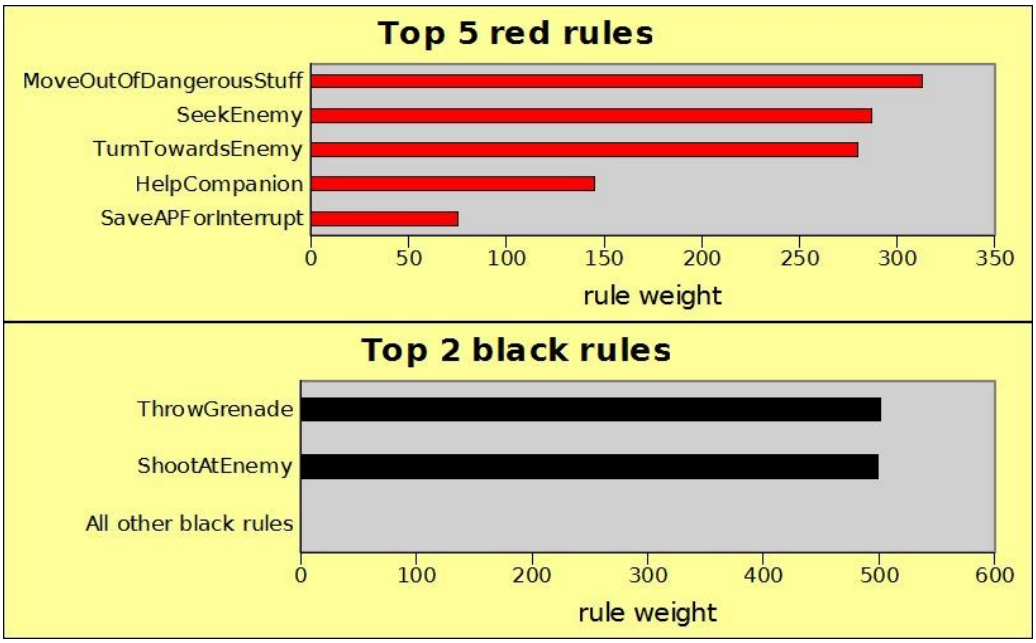

Figure 5.6: *Top rules for scenario 2*

Discussion:

The learning task in scenario 2 obviously posed no problem for the 'Dynamic Scripting' team. It took less than 10 matches on average to learn the proper tactic. Please note that the best possible tactic is to move close enough for a good throw and then use all available grenades on the enemy cluster. Most of the time at least one opponent survives, but since the blue team has also gas grenades in their repertoire, the remaining enemies will be unconscious. The blue agents can then easily finish them off with their pistols[5]. This tactic results in a victory with a team-fitness value of 1.0. The 'Dynamic Scripting' team was able to learn exactly this tactic in 9 out of 10 batches. Even so they managed to learn a successful tactic in the remaining batch, it was slightly inferior. The blue agents moved too close to the enemy cluster so their own grenades did some collateral damage to themselves. This is the reason the blue graph does not reach the 1.0 mark on average.

There is a very small drop between the 60th and 70th match. This is because the 'Dynamic Scripting' team managed to lose one single match in between while applying the same tactic as before. Grenade throws have a small chance to miss, so the loss was probably bad luck caused by two or more grenades missing their target.

Looking at the top rules, there are no major surprises. The red rules are basically unimportant, because with the right tactic the blue agents are exclusively in the black alarm state. But there was one 'sub-optimal' learning batch in which the red rules were relevant. In that situation 'MoveOutOfDangerousStuff' probably allowed the blue agents to move out of their own gas grenades. The black rules are straightforward: 'ThrowGrenade' the most important one, directly followed by 'ShootAtEnemy'.

5 The concept of captives does not exist in 'Jagged Alliance 2', hence the cruelty

5.4 Scenario 3

This test scenario is basically a spin-off of the first test scenario (chapter 5.2). All agents start at the same positions with the same attributes and the same equipment. The only difference is the removal of the trees on both sides of the battlefield. Please note while this may seem like a minor difference, it is actually something major. The analysis of scenario 1 revealed that the static opponent team acted too offensively, so it was possible to defeat them by exploiting the cover provided by the trees. This tactic is not possible anymore.

The goal for the 'Dynamic Scripting' team is to learn an alternative tactic to overcome the enemy squad. This scenario was designed to show the limits of the current implementation of the 'Dynamic Scripting' technology. Actually the learning task is quite difficult, because the key to victory in 'Jagged Alliance 2' lies very often in the intelligent use of the environment. Open field battles are to be avoided at all cost, because the winner of such a fight is determined by attributes and equipment.

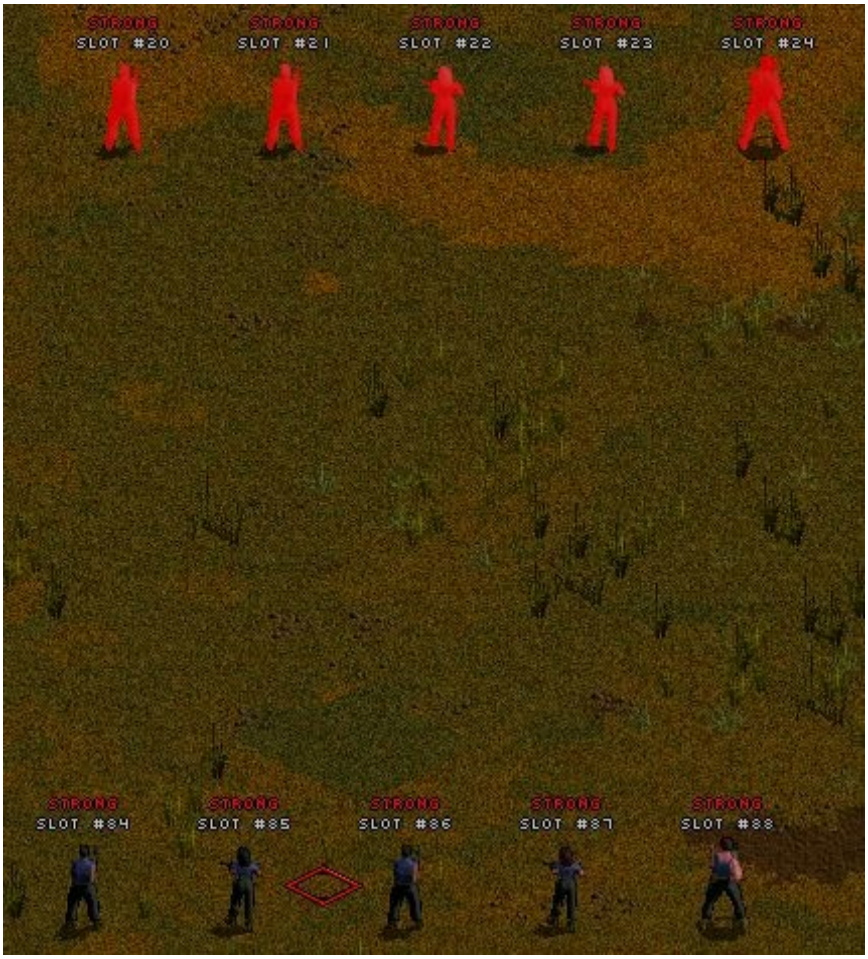

Figure 5.7: *Layout for scenario 3*

Here are the results:

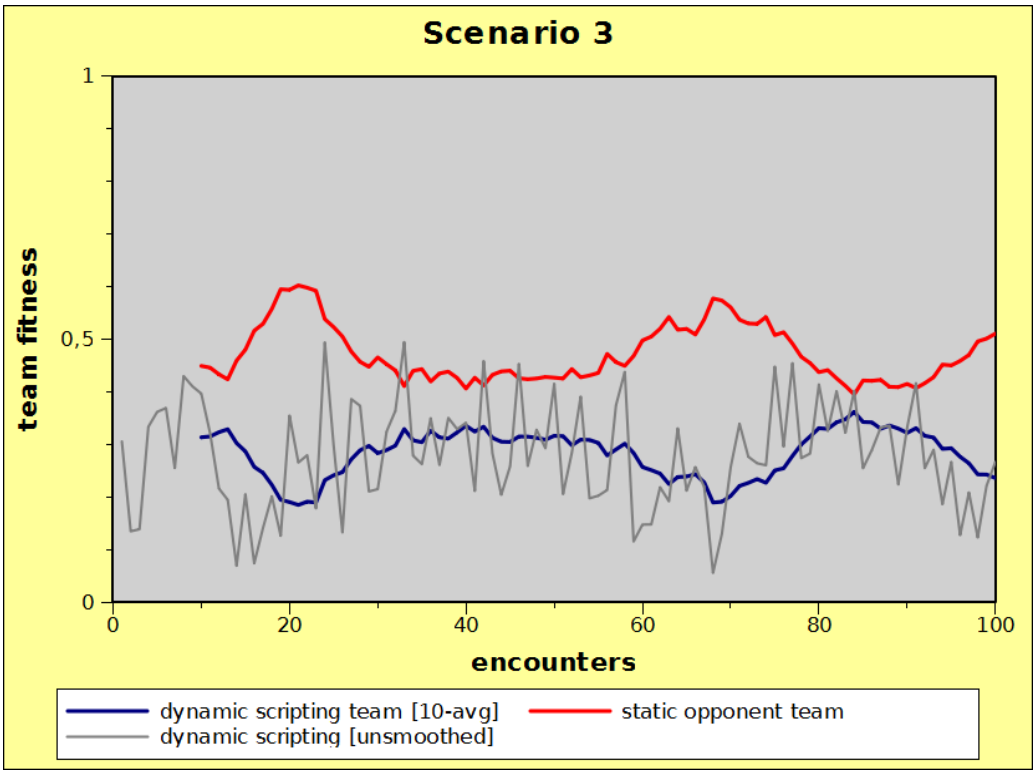

Figure 5.8: *Team-fitness progression for scenario 3*

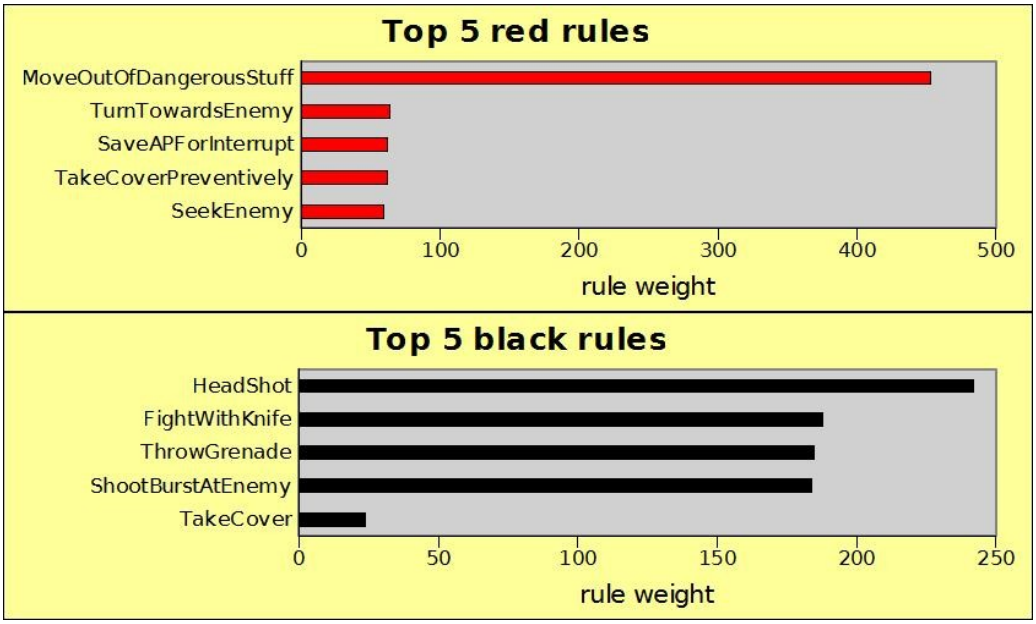

Figure 5.9: *Top rules for scenario 3*

Discussion:

The 'Dynamic Scripting' team was not able to learn a superior tactic for defeating the static game AI. This was probably due several reasons. First of all, as already mentioned, this scenario is an open-field battle, and usually the better equipped team wins in such a case. Since both teams were equally equipped during the evaluation, victory would be decided by the random number generator. However, this only holds true if both teams would always employ the (arguable) “best” tactic, which is to shoot as often and as accurate as possible. This would lead to a 50:50 win/loss ratio over a larger number of matches. By the way, the static AI team generally used that strategy.

The 'Dynamic Scripting' team did not always employ this tactic though. On a loss all activated rules are penalized, which leads to a higher probability for selection of other rules. Different scripts are generated by design to try out new tactics, which may perform better. However, in the case of this scenario, all other tactics failed. So does that mean that 'Dynamic Scripting' cannot cope with such scenarios in general?

Not at all. Some adjustments are necessary to deal with this and most probably other scenarios. The most important question is the following: How could a human player beat this scenario and what prevents the 'Dynamic Scripting' AI to mirror that behavior? Unfortunately there is no data for 'Jagged Alliance 2' to analyze human tactics, but I could think of a promising tactic with my personal domain knowledge. As taking cover is a crucial concept throughout the game, it may be a good idea in this scenario as well. There is no viable cover spot in the vicinity, but the map is much larger than the snippet shown in figure 5.7. An agent would have to walk two or even more rounds to reach a decent cover, but most probably this would be worth it. Even if some agents are wounded while running for defense, the advantage of the protection offered by trees or structures is just too critical.

The implemented rules as presented in chapter 4 do not allow the encoding of such a behavior though. Soldiers only consider locations for cover which are reachable in one round. Most of the time this is the right thing to do, because standing on a open field during the enemies' turn is usually a bad idea. Another aspect is efficiency: Evaluating the cover value of a game tile requires $O(n^2)$ calculations, so increasing the search range may noticeable impact the performance of the game, since these checks are possibly executed every round for every agent.

However, in this special case it probably would be a good idea to look for more distant cover spots. As mentioned before, the implemented rules do not claim to be perfect, optimizing them could increase the performance of the 'Dynamic Scripting' team in this scenario.

Another way of dealing with the issue would be the usage of a static tactic for certain scenarios (like this one). Basically the problem could be solved like outlined in the following pseudo code:

```
if (openFieldBattle == true) then
        ApplyTactic('Shoot');
else
        ApplyDynamicScripting();
```

This approach could be used for other situations as well. The idea is to use static tactics when a situation arises where a well-known best tactic exists. However one should not over-use this approach, because it limits the potential of adaption for the 'Dynamic Scripting' AI. The following chapter 6 provides more suggestions to embed 'Dynamic Scripting' into real games.

6. Conclusion

6.1 Summary

The outcomes of the three test scenarios were discussed in chapter 5. This chapter should provide a short summary along with my personal interpretation.

The results presented in chapter 5 are very promising in my opinion. I use the word 'promising' to indicate that there is of course room for improvement (see scenario 3). Basically a lot of fine-tuning could happen with more manpower, which would result in better (and maybe faster) learning results. But even with the time constraints of this thesis it was possible for the 'Dynamic Scripting' AI to learn tactics that exploited the weaknesses of the static original game intelligence in a very reasonable amount of time (speak: trials). Therefore the goal formulated in chapter 1.3 was reached.

I am very well aware of the fact that time constraints exists in commercial game development too, so there is only a limited amount of optimization that can be done. But my argument is that the time required to implement 'Dynamic Scripting' into a new game and do proper testing is not much more than the time required to write and test all static scripts. Or to put it more bluntly: If a single student who needed to read up on tens of thousands lines of source code could provide a nicely working implementation in three months, imagine what a bunch of professional programmers could do in the same amount of time.

Actually there are very few differences in the development process between a static scripted AI and an adaptive 'Dynamic Scripting' AI. The implemented rules may even be the same, the only difference would be that the static scripts are manually put together while the 'Dynamic Scripting' AI learns them. Both approaches require testing and optimization phases.

Please note that the definiteness of the results may raise the theory that the weaknesses of the static AI were implemented with intent. But this is most probably not the case, as the AI utilizes cheating mechanisms to keep up with the player (as described in chapter 5.1). The application of 'Dynamic Scripting' to 'Neverwinter Nights' had the same initial success and the developers of that game even released a patch with major fixes to the AI after those results were published **[SprNWN]**.

Pieter Spronck designed 'Dynamic Scripting' as a technology for the deployment in real commercial games. I want to stress that point again: This is not academic research just for the sake of it, but a real applicable technique that could greatly improve the 'fun factor' of a upcoming commercial computer game. Assuming a careful implementation there are little risks and many rewards.

The biggest worry regarding learning AIs is the unpredictability of what is learned. While this is a valid concern which is inherent to learning procedures of all kinds, 'Dynamic Scripting' has many characteristics that minimize this issue. A careful implementation of all rules in a rulebase is the first step to ensure that only meaningful behavior is generated. Another big advantage of 'Dynamic Scripting' is how the learned information is stored. The rule weights which are used to generate the scripts can easily interpreted by humans; rules with a low weight value are considered bad and rules with a high weight value are probably important for the success of an agent. If you think about other learning algorithms such as neural networks, this point no longer holds true.

6.2 Deficiencies in the current implementation

The implementation at hand is basically too general for an efficient use throughout the game. Every soldier has just one red and one black script; right now they learn these for each new scenario. Now the problem is that the scenarios (or sectors) are quite different from each other. Which means that a script that worked well in one case may perform very poor in another one.

For example take a look at scenario 2. The blue agents learned to assault the building and plaster the enemies with grenades. A very efficient tactic for conquering buildings; on an open field however this would be a bad idea. Soldiers should rather take cover there and attack from as far as possible, because trying to get in range for a grenade throw would almost certainly mean death for any agent.

There are probably several solutions for this issue. One possibility would be to classify the sectors into distinct categories. The agents could learn tactics (with 'Dynamic Scripting') for each different class of sectors then. This could be realized as static tags by the developers, so a sector with a large building gets a different tag than a sector which is a forest area. Or it could even be implemented as another learning task (with another learning procedure), where the classification of a sector is learned from a feature vector.

The problems regarding scenario 3 are obvious: The current rules did not allow the encoding of a superior tactic for that scenario. Possible solutions are the refinement of the implemented rules or the use of static tactics in situations like this (as described in chapter 5.4).

Another point of improvement would be the introduction of agent classes. Right now every computer controlled soldier is the same. Typical players however tend to specialize their mercenaries. For example they hire a mercenary with a high attribute value in marksmanship and dedicate him as a sniper and use another mercenary with a high value in explosives for placing bombs and such. The computer player does not make such differentiations, all soldiers are treated the same. While the allocation to different agent classes would even make sense without the use of 'Dynamic Scripting' it is all the more a good idea with it. Different agent classes could use distinct rulebases with specialized rules. Possible agent classes could be assault, sniper, grenadier, medic and more.

Also please keep in mind that a game featuring 'Dynamic Scripting' would not be shipped with all rule weights set to W_{INIT}. Naturally the developers would encode pre-learned tactics in the rule weights. The point of an on-line adaptive AI is not to deploy random acting agents who slowly learn meaningful behavior, but to provide the possibility of learning new tactics after the current one has been exploited.

There is one disadvantage regarding the 'Dynamic Scripting' procedure though. The final goal is to make digital games more immersive by providing intelligent acting agents. 'Dynamic Scripting' tries to reach this goal by optimizing the applied tactics in a stochastic manner. As a consequence scripts may be learned which are admittedly successful but contain unnatural looking rule sequences. Especially unnecessary rules are a problem, i.e. a single rule contained in a sequence of rules which provides no benefit at all. For example there was a case in 'Jagged Alliance 2' where an agent controlled by 'Dynamic Scripting' learned to hide behind a tree and then to run to an injured team member. This looked unnatural, because this agent was not under fire and one would expect him to help his companion immediately without a intermediate stop in another direction. Please note that this behavior was unlearned after some time, because the script generation is subject to a certain randomness. So at some point in time a script was generated which did not contain the rule responsible for hiding behind the tree, and this script was as successful as the other one. But since 'Dynamic Scripting' is designed as an on-line learning procedure, events like this may occur while a player is playing the game.

There are some countermeasures though, which provide no guarantee to eliminate the above mentioned incidents, but which should be able to reduce their probability of occurrence.

Firstly, one could limit the size of the learned scripts to a smaller number. This way scripts containing unnecessary rules are most probably not successful, which results in their early rejection. However this only limits the 'living time' of those scripts, it does not avoid their creation. Additionally, reducing the maximum number of rules in a script may prevent certain tactics from being learned.

Secondly, one could add additional conditions to certain rules, to ensure the rule is only applicable when it really makes sense. This is more complicated than it sounds, because foreseeing all possible situations in which a rule would be meaningful can be next to impossible (depending on the specific rule). Some obvious unwanted sequences of rules can be eliminated with that method though. A typical condition could be of the type 'if last action was/was not'.

6.3 Further improvements

This sub-chapter is meant to provide concepts for the general application of 'Dynamic Scripting' (i.e. not specific to 'Jagged Alliance 2').

Basically the most important point is that 'Dynamic Scripting' is very flexible and combines well with many standard approaches. For example it should be mentioned that scripts in computer games are not exclusively used to store tactics and strategies. They may also contain triggers for starting specific events or story sequences. These events are technically rules too, consisting of conditions and actions. However they should not be included in the rulebase for learning purposes, but rather be a static part of certain scripts. This is easy to realize with 'Dynamic Scripting' by just adding those events as default rules to the respective scripts.

It became obvious in the last sub-chapter that learning one general script for an agent for use throughout the game is not the optimal solution. 'Dynamic Scripting' receives no input of the current situation and/or state of the agent, therefore the developer has to make sure that a learned script is useful in the respective situation. As a consequence it makes sense in many cases to learn several scripts for each agent class and select one matching the situation (or game state). The challenge hereby is to find a balance between adaption to a specific situation and re-use of already learned knowledge.

For example take a typical FPS (= first-person-shooter): It could be the best solution to learn a behavior for every map (which is the counterpart to a scenario in 'Jagged Alliance 2') in such a game, since maps are quite often very different from each other. Also, players tend to play the same maps over and over again to improve their movement and learn about the important spots. This provides many opportunities for the 'Dynamic Scripting' procedure to learn as well.

On the other hand a typical RTS (= real-time strategy) game has slightly different characteristics. While there are different maps (or scenarios) as well, they are often not as individual as in FPSs. A more general classification could prove advantageous for those games. For example, since many real-time strategy games feature naval warfare, it may be meaningful to classify maps as 'water' and 'land-only'. This way tactics are not specifically-tailored towards one single map, which may circumvent the AI to explicitly exploit a custom feature of the scenario. However knowledge learned from a match on one map could be transferred to other maps, resulting in a more global adaption.

Please note that these were just examples, of course there may be real-time strategy games with very distinct maps favoring another approach than a rough classification and vice versa. At the end of the day everything depends on the game and all decisions regarding the AI are based on the custom rules and quirks of the game.

There are other ways of customizing 'Dynamic Scripting'. For example modifying the agent-fitness function on a per-agent basis opens up interesting possibilities. That means while two agents (or agent classes) draw the rules of their scripts from the same rulebase, they each got a different fitness function and store their own rule weights. For instance one could decrease the factor of $A(a)$ (survival capability)[6] and increase the factor of $C(g)$ (damage done)[6] for one type of agents to produce more aggressive tactics. That would be a simple yet efficient way of realizing different attitudes or personalities. A disadvantage would be the slower learning process though.

To sum it up: There are numerous ways of embedding 'Dynamic Scripting' in a game with many opportunities to customize the resulting behavior. Looking at the massive research in the fields of artificial intelligence and machine learning one can safely assume that the days of static game AIs are numbered. Basically it is only a matter of time when the first big game studio releases a major title with an adaptive AI, forcing the competitors to follow up.

6 as defined in chapter 4.5

Appendix A – 'Dynamic Scripting' pseudo code

The following pseudo code is taken from **[Spr05, page 82-83]**:

Algorithm 1 Script generation

```
ClearScript()
sumweights = 0
for i = 0 to rulecount - 1 do
    sumweights = sumweights + rule[i].weight
end for
for i = 0 to scriptsize - 1 do
    try = 0
    lineadded = false
    while try < maxtries and not lineadded do
         j = 0
         sum = 0
         selected = -1
         fraction = random(sumweights)
         while selected < 0 do
              sum = sum + rule[j].weight
              if (sum > fraction) then
                   selected = j
              else
                   j = j +1
              end if
         end while
         lineadded = InsertInScript(rule[selected].line)
         try = try + 1
    end while
end for
FinishScript()
```

Algorithm 2 Weight Adjustment

```
active = 0
for i = 0 to rulecount – 1 do
    if rule[i].activated then
         active = active + 1
    end if
end for
if active <= 0 or active >= rulecount then
    return {No updates are needed.}
end if
nonactive = rulecount – active
adjustment = CalculateAdjustment(Fitness)
compensation = -round(active * adjustment / nonactive)
remainder = -active * adjustment -nonactive * compensation
{Awarding rewards and penalties:}
for i = 0 to rulecount – 1 do
    if rule[i].activated then
         rule[i].weight = rule[i].weight + adjustment
    else
         rule[i].weight = rule[i].weight + compensation
    end if
    if rule[i].weight < minweight then
         remainder = remainder + (rule[i].weight – minweight)
         rule[i].weight = minweight
    else if rule[i].weight > maxweight then
         remainder = remainder + (rule[i].weight – maxweight)
         rule[i].weight = maxweight
    end if
end for
{Division of remainder:}
i = 0
while remainder > 0 do
    if rule[i].weight <= maxweight – 1 then
         rule[i].weight = rule[i].weight + 1
         remainder = remainder – 1
    end if
    i = (i + 1) mod rulecount
end while
while remainder < 0 do
    if rule[i].weight => minweight + 1 then
         rule[i].weight = rule[i].weight – 1
         remainder = remainder + 1
    end if
    i = (i + 1) mod rulecount
end while
```

Appendix B – Agent configuration

Szenario 1 + 3

The behavior of all red agents was set to 'Aggressive'. All blue and red agents possess the same equipment and the same attributes, namely:

Attribute	**Value**
Experience level	3
Life	80
LifeMax	80
Marksmanship	80
Strength	80
Agility	80
Dexterity	80
Wisdom	80
Leadership	80
Explosives	80
Medical	80
Mechanical	80
Morale	80

Slot	**Item name**
Head	Kevlar Helmet
Torso	Kevlar Vest
Legs	Kevlar Leggings
Hands	FN FAL
Inventory Slot 1	Combat Knife
Inventory Slot 2	Mag, 7.62x51mm, Mag 5
Inventory Slot 3	-empty-
Inventory Slot 4	-empty-

Scenario 2

Attributes blue agents (1 and 2):

Attribute	Value
Experience level	4
Life	80
LifeMax	80
Marksmanship	80
Strength	80
Agility	90
Dexterity	90
Wisdom	80
Leadership	80
Explosives	90
Medical	80
Mechanical	80
Morale	80

Equipment blue agent 1:

Slot	Item name
Head	Coated Kevlar Helmet
Torso	Coated Kevlar Vest
Legs	Coated Kevlar Leggings
Hands	Beretta 92F
Inventory Slot 1	Mk2 Defensive Grenade
Inventory Slot 2	Mustard Gas Grenade
Inventory Slot 3	9x19mm Pistol Mag 15
Inventory Slot 4	-empty-

Equipment blue agent 2:

Slot	Item name
Head	Coated Kevlar Helmet
Torso	Coated Kevlar Vest
Legs	Coated Kevlar Leggings
Hands	Beretta 92F
Inventory Slot 1	Mustard Gas Grenade
Inventory Slot 2	Stun Grenade
Inventory Slot 3	9x19mm Pistol Mag 15
Inventory Slot 4	Mk2 Defensive Grenade

The behavior of the red agents was set to 'Defensive'. All red agents possess the same attributes and equipment.

Attributes red agents:

Attribute	Value
Experience level	1
Life	60
LifeMax	60
Marksmanship	60
Strength	30
Agility	30
Dexterity	30
Wisdom	30
Leadership	30
Explosives	30
Medical	30
Mechanical	30
Morale	30

Equipment red agents:

Slot	Item name
Head	Treated Spectra Helmet
Torso	Treated Spectra Vest
Legs	Treated Spectra Leggings
Hands	AK-74
Inventory Slot 1	5.45x39mm Magazine
Inventory Slot 2	-empty-
Inventory Slot 3	-empty-
Inventory Slot 4	-empty-

Bibliography

[Alp04]: Ethem Alpaydın (2004). *Introduction to Machine Learning (Adaptive Computation and Machine Learning)*, MIT Press

[Für01]: Johannes Fürnkranz (2001). Machine Learning in Game Playing: A Survey. In J. Fürnkranz and M. Kubat (eds.), *Machines that Learn to Play Games*, pp.11-59, Nova Science Publishers, 2001.

[JA2awards]: Official 'Jagged Alliance 2' homepage – Awards, 2000. URL: http://www.jaggedalliance.com/awards/index.html [retrieved on 09.09.2008]

[JA2v113]: Jagged Alliance 2 v1.13 Mod Wiki, 2008. URL: http://ja2v113.pbwiki.com/ [retrieved on 13.05.2008]

[Mit97]: Tom Mitchell (1997). *Machine Learning*, McGraw-Hill

[PMSA06]: Marc Ponsen, Héctor Muñoz-Avila, Pieter Spronck, and David W. Aha (2006). Automatically Generating Game Tactics with Evolutionary Learning. *AI Magazine*, Vol.27, No. 3, pp.75-84.

[PMSA07]: Marc Ponsen, Pieter Spronck, Héctor Muñoz-Avila, and David W. Aha (2007). Knowledge Acquisition for Adaptive Game AI. *Science of Computer Programming*, Vol. 67, No. 1, pp. 59-75. (Springer DOI 10.1016/j.scico.2007.01.006)

[SBS07]: Frederik Schadd, Sander Bakkes, and Pieter Spronck (2007). Opponent Modeling in Real-Time Strategy Games. *8th International Conference on Intelligent Games and Simulation (GAME-ON 2007)* (ed. Marco Roccetti), pp. 61-68. (Presented at the GAME-ON 07 conference).

[Spr05]: Pieter Spronck (2005). *Adaptive Game AI*. Ph.D. thesis, Maastricht University Press, Maastricht, The Netherlands

[SprNWN]: Pieter Spronck, Neverwinter Nights Modules. URL: http://ticc.uvt.nl/~pspronck/nwn.html [retrieved on 16.09.2008]

[Tes95]: Gerald Tesauro. Temporal Difference Learning and TD-Gammon, originally published in *Communications of the ACM,* March 1995 / Vol. 38, No. 3. URL: http://www.research.ibm.com/massive/tdl.html [retrieved on 10.09.2008]

[TSH07]: Timor Timuri, Pieter Spronck, and Jaap van den Herik (2007). Automatic Rule Ordering for Dynamic Scripting. *Proceedings, The Third Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE07)* (eds. Jonathan Schaeffer and Michael Mateas), pp. 49-54. AAAI Press, Menlo Park, CA. (Presented at the AIIDE07 conference).