# Machine Learning in Real-Time Strategy Games

Author: Rik Vermeulen

Monitor: Dr. G.A.W. Vreeswijk

Table of Contents

1 Introduction

1.1 RTS-games 3

1.2 Goal 3

2 Dynamic scripting 4

2.1 Dynamic Scripting 4

2.2 Dynamic scripting in RTS-games 4

3 Improvements to dynamic scripting in RTS-games 6

3.1 Evolutionary Algorithms 6

3.2 Goal-directed Hierarchical Dynamic Scripting 9

3.3 Case-based reasoning 1

4 Discussion & Conclusion 14

7 References 15

# 1 Introduction

## 1.1 Real-Time strategy games

Real-time strategy games are a type of computer game in which the players train armies and have them fight against each other. To do this, players need to acquire resources, explore the map, and engage the enemy. Typically, as the game progresses, players gain access to stronger units by building specific buildings, or spending resources on certain upgrades. Players all do this simultaneously, there is no concept of a 'turn', hence the 'real-time' in real-time strategy games.

The AI in current real-time strategy games is, in one word, poor. Unlike game AI for more traditional games such as chess, the AI of real-time strategy games is no match for a skilled human player. Many games somewhat get around this by allowing the AI to 'cheat' on the higher difficulty settings – often, the AI has, for example, a higher income or lower unit cost. Even then, however, skilled human players are generally better. Buro (Buro, 2004) identifies the following points as the main reasons the performance of RTS AI is lagging behind the performance of AI in related areas such as classic board games:

- RTS-games feature large numbers of interacting objects, imperfect information, and a need to quickly perform many micro-actions. This is a stark contrast to turn-based, perfect information games, where most moves have global consequences and human planning abilities can simply be overcome by enumeration.
- AI-research costs time and money, and game companies do not (yet) have enough of an incentive to invest in it.
- Multiplayer games do not require world-class AI performance, as long as there are enough human players interested in playing online.
- Due to the complexity of RTS-games, it is difficult to set up an infrastructure to perform experiments on. Combined with the fact that most commercial software is closed and has no AI interface, the result is that there is a lack of AI competition in this area.

## 1.2 Goal

The goal of this paper is to take a look at machine learning in real-time strategy games, and propose a number of possible improvements to previously used algorithms. We focus on the dynamic scripting technique developed by Ponsen & Spronck (Ponsen & Spronck, 2005) in particular because early results with this technique are very promising, and because it is one of the only algorithms that have been developed for machine learning in RTS-games.

We will take a critical look at previous research, and attempt to suggest possible additions or improvements that might have been overlooked. We will also try to give some informed suggestions for future research in this area.

## 2. Dynamic scripting

### 2.1 Dynamic scripting

Dynamic scripting (Spronck et al, 2003) is a direct online learning technique, inspired by reinforcement learning (Russel & Norvig ,1995). Dynamic scripting is used in computer games because normal reinforcement learning techniques are not efficient enough (Manslow, 2002). In dynamic scripting, rules are drawn from a rulebase, essentially a database with script rules, to generate a script that controls the player's behaviour. (Henceforth, the player controlled by the dynamic scripting algorithm will be referred to as the 'dynamic player'.). To each rule is assigned a weight value in the range [0,1]. The higher the value of this weight, the higher the probability the corresponding rule will be selected.
After each game, the weights of the rules that had a positive effect on the outcome are increased, and those that had a negative effect decreased, such that the total weight remains unchanged.

### 2.2 Dynamic scripting for RTS games

Originally developed for computer role playing games (Spronck et.al, 2003), Ponsen & Spronck adapted dynamic scripting for use in RTS games, by introducing 'states' and 'state evaluations'. As typical RTS skirmishes can be divided into several phases, Ponsen & Spronck decided to structure these phases into game states. Whereas the CRPG implementation of dynamic scripting employs different rulebases for different types of opponents, Ponsen & Spronck's RTS implementation has different rulebases for different game states. These states are intended to roughly reflect all phases in an RTS based on what buildings the player has built, and consequently what units there are available.

Additionally, in dynamic scripting for CRPG's , rules have only a single weight. However, this does not suffice for RTS games as the success rate of a rule can vary greatly based on the state the game is in. Therefore in Ponsen & Spronck's implementation, each rule gets several weights attached to it, specifically one weight per state. So a rule would, for example, have a weight of 0 for state 1, 0.6 for state 2, and 0.3 for state 3. Having a weight of 0 for certain rules will occur when it is impossible to use that rule in that state due to, for example, build restrictions in that particular state.

The dynamic script is then created as follows: The game starts in state 1, and the technique will randomly select a rule from the rulebase for state 1 until it selects a rule that triggers a state change, and from then on will select rules from the rulebase for that state. The probability that a rule is chosen depends on its weight – the heavier the weight, the higher the probability a rule will be selected. In order to prevent monotonous behaviour, a limit on the number of times a rule may be selected is implemented.

Ponsen & Spronck based the weight adaptations on two fitness functions, one evaluating the game as a whole ('overall fitness') and one evaluating all states visited during the game ('state fitness'). They defined the 'overall' fitness function as follows :

$$F = \begin{cases} \min(\frac{Sd}{Sd+So}, b)\{d \text{ lost}\} \\ \max(b, \frac{Sd}{Sd+So})\{d \text{ won}\} \end{cases}$$

Where $F$ is the overall fitness for player $d$, $Sd$ is the score of player $d$, So the score of $d$'s opponent, and $b$ is the break even point. The break even point is the point at which the weights remain unchanged. Player d is the player controlled by the dynamic scripting algorithm.

And the 'state fitness' function was defined as

$$Fi = \begin{cases} \frac{Sd,i}{Sd,i + So,i}\{i = 1\} \\ \frac{Sd,i}{Sd,i + So,i} - \frac{Sd,i-1}{Sd,i-1 + So,i-1}\{i > 1\} \end{cases}$$

Where $Fi$ is the fitness of state i for player d, $Sd,x$ is the score of player $d$ after state $x$ and $So,x$ the score of the opponent after state $x$.

The score $Sx$ for a player x was defined as follows:

$Sx = 0{,}7Mx + 0{,}3Bx$, where $Mx$ represents the number of 'military points' for player $x$, i.e. the points awarded for killing enemy units and destroying enemy buildings, and $Bx$ represents the building points for player $x$, the number of points awarded for constructing buildings and training units.

The final weight update function then was

$$W = \begin{cases} \max(Wm, Worg - 0.3\frac{b-F}{b}P - 0.7\frac{b-Fi}{b}P)\{F < b\} \\ \min(Worg + 0.3\frac{F-b}{b}R + 0.7\frac{Fi-b}{1-b}R, Wma)\{F \geq b\} \end{cases}$$

Where $W$ is the new weight value, $Worg$ is the original weight value, $P$ is the maximum penalty, $R$ is the maximum reward, $Wma$ is the maximum weight value, $Wm$ is the minimum weight value, $F$ is the overall fitness, $Fi$ is the state fitness in state $i$,
and b is the break even point.

Ponsen & Spronck chose the numbers for these update functions intuitively. Dahlbom & Niklasson (2006) later investigated how punishment and reward factors affect the learning rate. They investigated three different settings for these factors: Higher rewards, higher punishments, and equal punishments and rewards. They also investigated if adaptation time can be shortened by increasing both factors proportionally at the same time, increasing the learning rate factor. It is however important to remember that too large a learning rate can make the algorithm predictable, removing one of dynamic scripting's benefits – unpredictability (Spronck et al, 2003). Dahlbom & Niklasson also argue that with dynamic

scripting, considering the temporal aspects of results could also be applicable. For example, if a rule receives low fitness for a number of consecutive evaluations, it is likely the rule is no good and it's weight can be drastically reduced. A potential way to realize this would be to track the trend over time of the fitness results, i.e. to introduce the derivative of the fitness results. However, using the derivative of the results is not directly applicable as the fitness results do not constitute a continuous function. An approach could be to use some form of smoothing function such as a non-uniform rational b-spline (NURB). Fitness results could be inserted into a NURB curve which could be used to find the derivative. For more information regarding NURB curves and their derivative, see for example Piegl and Tiller (1995). Dahlblom & Niklasson report significantly shorter adaptation times with a higher learning rate factor, and also when including the derivative.

Ponsen & Spronck report that, while the dynamic player was quick to statistically outperform a standard balanced AI, it failed to outperform optimized strategies (as are often employed by human players). Dahlbom & Niklasson also report that static opponents were relatively quickly outperformed statistically when compared to dynamic opponents.

## 3 Improvements to dynamic scripting

This chapter will consider several approaches to improving or adapting the dynamic scripting algorithm.

### 3.1 Evolutionary Algorithms

One way of improving dynamic scripting is by increasing domain knowledge. This can be done by either adding domain knowledge to the rulebase, or using more input data from the environment. Adding domain knowledge to the rulebase is, in essence, adding more useful rules to the rulebase.
In order to find additional rules for their rulebase to increase dynamic scripting's performance against optimized strategies, Ponsen & Spronck used an Evolutionary Algorithm (EA) to evolve several counter-strategies to these optimized strategies, and then manually derived new rules for the rulebase from these evolved strategies. Ponsen et al. later developed a method to automatically update the rulebase with strategies gained from evolutionary algorithms. When applying an EA to an RTS game, encoding and evaluation are the most critical design issues. Encoding schemes must be able to represent any possible solution, and preferably unable to represent infeasible solutions. Ponsen & Spronck therefore gave the EA maximal freedom in rule selection and rule parameterization but prevented it from inserting illegal rules into the solution. They did this by corresponding game states to a set of rules the EA was allowed to choose from.

Ponsen & Spronck grouped the genes on a chromosome into states. They defined four types of genes: build genes, economy genes, combat genes and research genes. Build genes construct buildings, and start with the letter 'B', followed by a number representing the building to be built. Research genes are similar, but start with the letter 'R'. Economy genes start with the letter 'E' followed by a number indicating the number of workers to be built. Combat genes are somewhat more complex. They start with the letter 'C', and then a number representing the current state. A combat gene in state 1 would start with 'C1', while a combat gene in state 12 would start with 'C12'. The first parameter of a combat gene is an identifier for an army. The last parameter is the role of the army, offensive or defensive. The number of parameters between the first and last, indicating what number of what sort of unit should be built, can vary depending on state. This is because in later states, more kinds of military unit are available. For example, part of a chromosome could look something like this:

S1 C1 2 5 defend B4 S3 E8 R15 B3 S4 C4 2 2 5 5 defend B1 C4 5 7 3 3 attack ….

'S1' indicates the script is in State 1. Next, a combat gene. 'C1' means it is a combat gene for state one. The following 2 is the identifier for the army that will be built, which consists of 5 soldiers with a defensive role. Then, build gene B4 builds the building represented by the number 4, triggering a state change, as indicated by the following S3, showing the script is now in state 3. Then, economy gene E8 trains 8 workers, and then research gene R15 does the research represented by the number 15. Build gene B3 triggers a state change again, bringing the script to state 4. Then a combat gene again, for state 4 this time as indicated by the 4 in C4. It has the same army identifier as the soldiers that were built earlier. However, this gene trains 2 soldiers, 5 shooters and 5 catapults, again with a defensive role. Shooters and catapults were unavailable in state 1, but are in state 4, thus the extra parameters for the combat gene.

The evaluation function was defined as:

$$F = \begin{cases} \min(\frac{GC}{EC} * \frac{Md}{Md + Mo}, b) \{d \text{ lost}\} \\ \max(b, \frac{Md}{Md + Mo}) \{d \text{ won}\} \end{cases}$$

Where *F* is a fitness value in the range [0,1], *Md* are the military points for the player *d*, *Mo* are the military points for the opponent, *GC* is the game cycle, the time it took for one of the players to lose the game, *EC* is the 'end cycle, the maximum amount of time a game is allowed to run, and b is the break even point. This formula is rather similar to the fitness functions for the weight update function discussed in section 2.2. As they are both fitness functions for the same game, their similarity is not surprising. There are, however, two differences: First, the fitness function for the EA here uses only military points rather than the complete score to determine the fitness. Or, put differently, building points are not used to determine fitness for the EA. Second, the GC/EC factor.

Ponsen & Spronck do not explain their reasoning for not including building points for this fitness function. Most likely, they felt that with the way they encoded their chromosomes it would not be necessary and possibly counterproductive. Unnecessary, because chromosomes with 'fit' military genes would also have to have 'fit' build genes. Possibly counterproductive, because increasing 'building points' would not add at all to the military strategy they were hoping to evolve, and might inflate the fitness of a relatively unfit chromosome.

The reason for the GC/EC factor is as follows: If the evolutionary script fights a long battle but eventually loses, it is probable the chromosome is close to finding a solution and as such should still have a reasonable fitness score. The GC/EC factor ensures that losing solutions that play longer games than other losing solutions are granted a higher fitness score.

Ponsen & Spronck had four genetic operators to evolve a solution: state crossover, rule replace mutation, rule biased mutation and randomization.

With state crossover, 2 parents are selected, then checked if they have at least 3 activated states in common. (A state becomes activated when the AI has executed at least 1 gene in that state). If so, state crossover can be used. It must be ensured that the child inherits material from both parent chromosomes to prevent a parent being completely copied onto the child. To prevent illegal state changes from appearing on the chromosome, all genes between 2 matching states are copied onto the child. After the last activated state, the rest of the chromosome is copied from one of the parents.

In rule replacement mutation, select a single parent, and all research, economy or combat genes in it have a 25% chance of being replaced. Build rules are excluded from being replaced as well as being replacements because doing so could trigger state changes and corrupt the chromosome.

Rule biased mutation works by selecting 1 parent and giving each existing combat or economy gene a 50% chance of mutating. These mutations are within predefined minimum and maximum values. Both research and build genes are excluded from this sort of mutation.

Changing the parameters for those rules doesn't really make sense, and could also possibly lead to a corrupt chromosome.

In both rule biased mutation and rule replacement mutation, only genes in activated states are considered. Genes in states that aren't activated are considered 'dead'.

Ponsen & Spronck report that their genetic algorithm was able to defeat optimized strategies their original dynamic scripting algorithm could not. Initially, Ponsen & Spronck manually extracted new rules from the evolved strategies to add to their rulebase, and noted a marked improvement to the dynamic scripting algorithm's performance against optimized opponents, although it was still unable to statistically outperform them. Later, Ponsen et al. developed a method to automatically generate these new rules, called AKADS (Automatic Knowledge Acquisition for Dynamic Scripting). (Ponsen et al, 2005). AKADS has 3 steps in generating adaptive AI opponents:
1. Evolve domain knowledge using an evolutionary algorithm (as described earlier in this chapter)
2. Within-domain knowledge transfer
3. State-based tactics selection (as described in chapter 2 of this paper)
Steps one and three have already been described in this paper. However, AKADS automatically extracts tactics from the evolved chromosomes and adds them to state-specific rulebases. All genes in an activated state are considered to be a single tactic. Dynamic scripts generated with AKADS significantly outperforms the scripts generated with a manually designed rulebase (Ponsen et al, 2005). This is most likely because the evolved knowledge bases are not restricted to the domain knowledge provided by the designer, which may be poor, and because the automatically generated knowledge bases include tactics that consist of multiple atomic game actions. This in contrast to the improved knowledge bases that include tactics that consist of a single atomic action. Having compound tactics in a knowledge base reduces search complexity and allows dynamic scripting to adapt quickly to many static opponents. Although the AKADS-generated scripts performed better than the ones with manually generated rulebases, they were still unable to statistically outperform the optimized scripts.

### 3.2 Goal-directed Hierarchical Dynamic Scripting

Dahlbom & Niklasson (2006) developed another way to improve the rulebase of dynamic scripting, which they call 'goal-directed hierarchical dynamic scripting' (GoHDS).

Unlike the Dynamic Scripting for RTS games proposed by Ponsen & Spronck, GoHDS does not have separate rulebases based on game states, but different rulebases based on what type of player the algorithm is supposed to model, as the dynamic scripting algorithm for CRPG's did (Spronck et al, 2003). However, in GoHDS, rules with the same 'purpose' are grouped together, and it is said these rules have the same goal. Each rule in a rulebase has a purpose, and several rules can have the same purpose. A rule is then seen as a strategy for achieving a certain goal, which can be seen as domain knowledge directing the behaviour.

Dahlbom & Niklasson argue that 2 advantages can be gained by extending dynamic scripting with a goal-based component: 1)The illusion of intelligence can be strengthened, and 2) complex domain knowledge possessed by human designers can easily be translated to individual goals and prerequisites.

GoHDS' learning mechanism operates with the probability that a rule has been selected in order to fulfil a certain goal. Because certain rules may be useful in working towards more than one goal, it is desirable to allow for rule reusability. By removing weights from individual rules and attaching them to relationships between goals and rules, adaptation can occur in a separate learning space for each goal. This can allow for higher flexibility and reusability.

Rules in GoHDS are divided in two distinct states, an 'init' state and an 'active' state. The init state has the purpose of seeing if the global state is suitable for a given rule, meaning that possible preconditions for the rule are checked. If these preconditions are not met, goals are started with the purpose of fulfilling these preconditions. For example, if an army of a certain advanced unit, such as catapults, is to be built, then certain buildings need to have been built. In the case these have not yet been constructed, there is no need to check if there is enough money to build catapults. Instead, a goal to construct these buildings is started.

When a rule's preconditions have been fulfilled, it changes to the 'active' state. A rule's active state has the purpose of executing the main action of rules if their optional condition(s) is (are) true, e.g. to build a number of catapults in the previous example.

In dynamic scripting, rules are designed in a clear and understandable fashion. Usually, this is an advantage. However, if the rules are to give the illusion of intelligence at the tactical/strategic level, this might pose a problem. For example, a rule for ordering a fast assault on an enemy base will not be simple, nor will it be reusable. Therefore, in GoHDS, such rules are broken down in smaller rules and sub-goals. These are then connected to form a hierarchy of rules and goals. By dividing rules in this way, it is easier to maintain simplicity and understandability.

GoHDS on it's own, however might not be enough to create a seemingly intelligent opponent. An AI in an RTS game at the strategic/tactical level faces such tasks as resource allocation, modelling, spatial- and temporal reasoning, planning, and decision making. (Dahlbom & Niklasson, 2006). As the GoHDS method does not contain a communication system, is not a spatial reasoning system nor a temporal reasoning system, collaboration and spatial and temporal reasoning are not possible with it. Dynamic scripting is not a system for making plausible hypotheses considering enemy intent, so explicit modelling is ruled out as well. However, the weights learned by the dynamic scripting algorithm implicitly model the behaviour of the enemies from which these weight allocations were learned. Also, in GoHDS, some degree of planning and resource allocation is contained in the goal hierarchy and rule preconditions.

Dahlbom & Niklasson argue that, for these reasons, GoHDS might need to be complemented with other systems in order to be useful in practice. These systems would include a perception system, a modelling system, a resource management system and a pathfinding system. A perception system is required in order to act. This could, for example, be a 2D map containing all vital objects seen by friendly units. The perception system could also be complemented with influence maps for detecting movement patterns of enemy units. This perception system could then be used by a modelling system which, for instance, could keep a state vector of the world. Each state could be matched against a target state, and for each state not fulfilled a goal could be started in GoHDS. GoHDS should also be able to communicate with the perception system on its own in order to receive state information. The modelling system and GoHDS

should also be able to communicate with some manner of resource management system that prioritizes and performs production scheduling. Finally, a pathfinding system could be used by GoHDS, the modelling system and a pathfinding system. The pathfinding system could also use a terrain analysis system for input. Figure 2 illustrates a simple example of the described system.

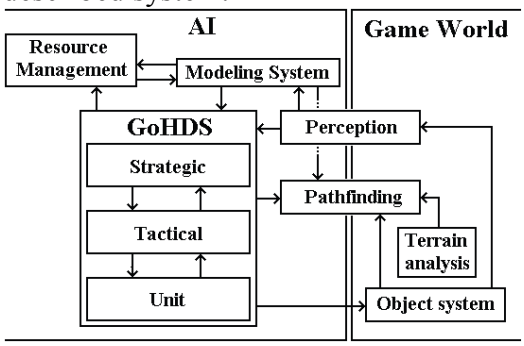


Fig. 2: example of GoHDS combined with other systems.
(Source: Dahlbom & Niklasson, 2006)

### 3.3 Case-based reasoning

So far, we have discussed the dynamic scripting algorithm learning to counter a single strategy. However, we would like to perform well against multiple enemy tactics, rather than assuming a static opponent. Thus, Aha et al. developed the 'Case-based Tactician' (CaT). CaT added another layer of domain knowledge to the algorithm to allow for case-based reasoning. In addition to the state lattice consisting of building states and the set of tactics to select from as used by Ponsen & Spronck (Ponsen & Spronck, 2004), they introduced cases that map game situations to tactics and their performance, significantly increasing performance. Tactics were automatically generated similar to the AKADS method, which has been mentioned in chapter 3.1 of this paper. While CaT is not a dynamic scripting algorithm, it builds on the work done by Ponsen & Spronck for their dynamic scripting so much that it should be included here.

A 'case' C was defined by Aha et al. as a tuple of four objects: $C = \langle BuildingState, Description, Tactics, Performance \rangle$, where *BuildingState* is an integer node index in the state lattice, *Description* is a set of features of the current situation, *Tactic* is a counter-strategy's series of action for that state, and *Performance* is a value in [0,1] that reflects the utility of choosing that tactic for that *BuildingState*.

When a new state in the state lattice is entered, CaT retrieves cases. At those times, it records the following information to form a Description:
- The number of opponent combat & worker units killed minus the same for oneself, in the preceding state
- The number of opponent buildings destroyed minus same for oneself, in the preceding state
- The number of opponent buildings ever created
- The number of opponent combat units ever created
- The number of opponent worker units ever created
- The number of own buildings currently existing
- The number of own combat units currently existing

- The number of own worker units currently existing

These numbers were selected because they are available and intuitively informative, and they balance information on recent game changes, the opponent's situation, and the player's situation. Cases were grouped by BuildingState, and, after each game ends, at most one case is recorded per BuildingState. CaT's function for computing the similarity between the current game description *S* and a stored case *C* is defined as follows:

$$Sim(C,S) = (C_{Performance} / dist(C_{Description}, S)) - dist(C_{Description}, S)$$

Where *dist()* is the Euclidian distance among the eight features.

A k-nearest neighbour function was used to select case Tactics for retrieval. Of the k most similar cases, it retrieves the one with the highest performance. However, in order to gain experience with all tactics in a state, case retrieval was not performed until each available tactic at that state had been selected *e* times, where *e* is called the *exploration* parameter of CaT. While still in this 'exploration' mode, CaT retrieves one of the least frequently used tactics for reuse. This 'exploration' is also done whenever the highest performance of the k nearest neighbours is below 0.5.

CaT's reuse process is given the retrieved case Tactic. While adaptation takes place, this is not controlled by CaT, rather it is controlled at the level of the action primitives in the context of the game engine. This means that, when for example the construction of a building is requested by an action, where it is built, and what workers will construct it, is decided by the game engine. This could differ in each game situation.

In revision, the reused tactics are executed in the game, and the results evaluated. These tactics are not altered themselves. Evaluating these results yields the Performance of a case's tactic. Performance is measured at both a local and global level.

CaT records the game score for both the player and the opponent at the start of each BuildingState and the end of the game. A game ends when all of one player's units and buildings have been destroyed, or when the game is terminated if no victor has emerged after 10 minutes. Performance of a Tactic *t* for a Case *C* in a BuildingState *b* was defined as a function of it's 'global' score ($\Delta Score_i$) and it's 'local' score ($\Delta Score_{i,b}$). The 'global' score focuses on relative changes between the time that *t* begins executing in *b* and the end of the game. The 'local' score focuses only on changes during state *b*.

This function was defined as follows:

$$C_{Performance} = \sum_{i=1n} \frac{C_{Performance\,i\,,}}{n}$$

$$C_{Performance\ i\,,} = \frac{1}{2}(\Delta Score_i + \Delta Score_{i\ b,})$$

$$\Delta Score_i = \frac{Score_{i,\,p} - Score_{i,\,p,\,b}}{(Score_{i,\,p} - Score_{i,\,p,\,b}) + (Score_{i,\,o} - Score_{i,\,o,\,b})}$$

$$\Delta Score_{i,b} = \frac{Score_{i,p,b+1} - Score_{i,p,b}}{(Score_{i,p,b+1} - Score_{i,p,b}) + (Score_{i,o,b+1} - Score_{i,o,b})}$$

where $n$ is the number of games in which $C$ was selected, $Score_{i,p}$ is the player's score at the end of the $i^{th}$ game in which $C$ is used, $Score_{i,p,b}$ is player $p$'s score before $C$'s Tactic is executed in game $i$, and $Score_{i,p,b+1}$ is $p$'s score after $C$'s Tactic executes (and the next state begins). Similarly, $Score_{i,o}$ is the opponent's score at the end of the $i^{th}$ game in which $C$ is used, etc.

During a game, CaT records a Description when it enters a BuildingState, along with the score and selected Tactic. It also records the score of both players at the end of the game, and also who won. In case of a tie, neither player wins. For each BuildingState visited in a game, CaT checks if there exists a case with the same Description and Tactic. If so, the Performance of that case is updated. If not, a new case is created with the Performance calculated as during revision. There is no case deletion policy in CaT.

Aha et al. report that CaT learns to perform significantly better than the best performing evolved counter-strategies against opponents in WARGUS.. Specifically, it wins over 80% of it's games after 100 games. Additionally, CaT's algorithm has not yet been tailored to this application, thus its performance can probably be improved further (Aha et al, 2005).

## 4. Discussion & Conclusion

We have taken a look at ways for computers to learn to win a real-time strategy game. Specifically, we have looked at the dynamic scripting algorithm for RTS-games and offshoots of it. In Ponsen & Spronck's implementation of dynamic scripting in WARGUS, they used game states based on what buildings there are available, and learned weights for each state. With 20 states, weights can be reasonably learned for each state. However, as the number of states increases, learning weights for all states will start taking considerably more time. It is unclear whether this is a real cause for concern; it is unknown what would be an average number of states for an RTS-game.

Each of the described methods has only been tested in one-versus-one scenario's. However, it is not uncommon in RTS-games for a game to consist of more than 2 players. Theoretically, there are no great obstacles preventing dynamic scripting from performing well in free-for-all scenario's with 3 or more players. (Ponsen & Spronck mention that, if they want to test the AI against multiple opponents, they would have to alter the fitness function). However, in an X versus Y scenario (where X and Y are >1), communication between allied players becomes an important factor. As dynamic scripting does not have any method to communicate, it would most likely perform poorly in such a situation, and would need to be extended with some sort of communication system. The case based reasoning approach described suffers from a similar problem, however in addition, the information within a Description would increase with each opponent added, as Descriptions contain data relating an opponent's status to the player's status. In free for all games, only recording information of how each opponent's situation relates to the player's own would probably be enough. However, in 2 versus 2 or larger games, it might also be needed to record how each opponent's situation relates to each allied player's situation. Recording this amount of information would quickly become impractical.

The GoHDS structure proposed by Dahlbom & Niklasson seems very interesting on paper. However, it has not really been tested. One of the more interesting features of GoHDS is that it allows for the reuse of rules, as it is not unlikely that applying a certain rule multiple times may be beneficial. However, it is somewhat unclear how GoHDS should determine when a goal should be started. Most likely, this would involve some manner of opponent modelling.

The dynamic scripting algorithm learns how to deal with a static opponent, however when faced with a different opponent who deploys a different tactic, it must re-adapt itself. This is rather undesirable, no matter how few games are necessary for re-adaptation. In a commercial game, this is mostly undesirable because a human player will quickly become bored with an opponent he has to fight with the same tactic several times before it is a challenge, and again when he changes his tactic somewhat. For researchers, this is undesirable because the (ultimate) goal of RTS-AI research is to make as strong an AI as possible, one that can consistently defeat human players without 'cheating'.(Buro, 2004). With an algorithm that can only perform well against one specific tactic, that will not happen. In contrast, CaT was specifically designed to be able to deal with a number of different opponents without having to re-adapt each time it faced a different tactic.

The CaT implementation discussed here however is a bit of a cheater. Usually, the opponent in an RTS games cannot be freely observed, thus some of the information CaT uses to form a game Description would normally be unavailable. (Aha et al, 2005).

So far, dynamic scripting has been unable to defeat highly optimized scripts. However, we have seen that improving the rulebase of the dynamic script did increase its performance against such scripts. Possibly, further optimization of the rulebase could further increase the dynamic scripting algorithm's strength. This might be done by evolving a larger number of different solutions with a genetic algorithm. Expert knowledge of human players however could also prove to be useful in this endeavour.

Finally, while the algorithms described have performed well against other artificial intelligence opponents, they have not been tested against human players. Even though the AI's were only trained against certain tactics, setting them against a human player could give some insight in the general strength of the AI. It may be the generated scripts have some sort of weakness that is obvious to a human, but not a machine.

Taking these things into account, interesting areas for future research are:

- The efficiency of machine learned tactics against human opponents.

- Improved opponent modelling for RTS-games, and effectively using such opponent models. Specifically, a way of opponent modelling that uses only information that would also be available to a human player, that can also comfortably model multiple opponents.

- Communication between allied dynamic scripts to allow for coordinated attacking and defending, against multiple opponents.

- Further development and testing of the GoHDS method, and the systems it is required to work with.

- Further optimization of the dynamic scripting rulebase.

## 6. Summary

We set out to take a look at machine learning in RTS-games, with an emphasis on dynamic scripting, and suggest some additions or improvements to the methods used up until now, and give some direction to future research in this area. Several additions and changes to the dynamic scripting algorithm will also be discussed, including improving the knowledge base with the aid of an evolutionary algorithm and altering it into a case-based reasoning system. These and other methods will be discussed in-depth.
We see that dynamic scripting on its own, although performing decently against balanced scripts, does not deal well with optimized scripts. We also see that the additions and alterations to the algorithm improve it's performance against these types of opponents.
We conclude to suggest a number of avenues for future research, including improvements to opponent modelling for RTS-games, and communication systems between allied dynamic scripts.

## 7 References


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

Spronck, P., Sprinkhuizen-Kuyper, I. and Postma, E.: Online Adaptation of Game Opponent AI in Simulation and in Practice. In: Mehdi, Q., Gough, N. and Natkin, S. (eds.): Proceedings of the 4th International Conference on Intelligent Games and Simulation (2003) 93–100

Ponsen, M. and Spronck, P. (2004). "Improving Adaptive Game AI with Evolutionary Learning." Computer Games: Artificial Intelligence, Design and Education (CGAIDE 2004). pp. 389-396. University of Wolverhampton.

Russel, S., J., & Norvig, P. (1995) Artificial intelligence: a modern approach, Prentice Hall, Inc, Upper Saddle River, NJ.

Dahlbom, A., and Niklasson, L. 2006. Goal-directed hierarchical dynamic scripting for RTS games. In *Proceedings of the Second Artificial Intelligence and Interactive Digital Entertainment Conference*, 21–28. Menlo Park, CA: AAAI Press.

Manslow, J. (2002). Learning and Adaptation. AI Game Programming Wisdom (ed. S. Rabin), Charles River Media, 2002, pp. 557-566.

Ponsen, M.J.V., Muñoz-Avila, H., Spronck, P., & Aha, D.W. (2005). Automatically acquiring domain knowledge for adaptive game AI using evolutionary learning. To appear in *Proceedings of the Seventeenth Conference on Innovative Applications of Artificial Intelligence*. Pittsburgh, PA: AAAI Press.

Piegl, L., and Tiller, W. 1995. *The nurbs book, 2nd edition*. Springer.

David Aha, Matthew Molineaux, and Marc Ponsen. Learning to win: Case-based plan selection in a real-time strategy game. In ICCBR, pages 5–20, 2005.