Learning RTS AI

by

Tom Sverre Hageland, Gjermund K. Lie
Supervisor: Ole Christoffer Granmo

Project report for IKT407 in Autumn 07

based on report template version 3.0 (2006)

Agder University

Faculty of Engineering and Science

Grimstad, 20 November 2007

Status: Final

Keywords: Artificial Intelligence, Computer controlled player, Learning automaton

Abstract:

During this project, we intended to achieve something uncommon in real time strategy games - an AI that learns from playing. In order to reach a level of abstraction where we could actually achieve results, we had to create a simulated environment. Here we could, with some accuracy, predict what changing various variables would result in when it came to the result of the game. We have then tested our AI in its various completition stages against simulated opponents with specified difficulty settings. Finally, we have implemented a basic system for long-term learning. Despite our best efforts, achieving proper learning in the system the simulator creates seems impossible, and our results are thus, naturally, not entirely conclusive.

Table of Contents

1 Introduction. 3

2 Problem description. 4

3 Background. 4

4 Solution. 5

4.1 Requirements. 5

4.2 Design Specification. 5

4.3 Implementation. 6

4.4 Validation and Testing. 13

5 Discussion. 15

6 Conclusion. 16

Appendices. 16

App: Glossary & Abbreviations. 16

App: Attatchments. 16

1 Introduction

Our task as it was originally given:

> *Resource allocation scheduling for ORTS. - When to devote time to get more resources. This could be a so called need driven AI.*
>
> *1. Observe and evaluate the environment*
> *2. Find the most urgent need, based on the environment and the self*
> *3. Perform the need*

A required clarification: Plans for implementing the AI into ORTS was discarded early in the project planning, for various reasons. The need for repeated test-runs for statistic purposes was one of the most pressing concerns, something we could not do efficiently using the ORTS engine. The need for simplicity in order to be able to achieve anything at all was another concern.

Thus, our task became - construct an AI that, depending on the game state (environment) and its own state, would determine its most urgent need and then take steps towards this need. Upon completion of this step, we intended to upgrade the AI using a learning automaton, ideally allowing the AI to win every game with the same starting strategy.

As there is very little prior work in this specific area, we would have to start more or less from scratch. Hopefully through this report you can see how we decided to implement a learning AI, and see how we came to our conclusion.

2 Problem Description

For this project we intended to create a need-driven AI for a simulated real-time strategy game, controlling how the computer will react to "The big picture" - an AI that reacts to the information (concerning available resources, game state, relative military strength, etc.) it has available, attempting to alter the game state into something more favorable to itself. The variables available for altering for a "player" in the simulator includes percentage ratings of resource gathering and unit production, as well as aggression (how likely the player is to perform an attack any given round) and base/expansion relative attack rates (in plain english, the odds to attack the home base or expansions of the enemy). The computer should be able to react reasonably well to most scenarios, including running out of resources (need to expand) and any composition of enemy troops (anti-units).

In order to develop, test and improve this AI, we needed a heavily simplified game simulator. The simulator had to deliver benchmark results with two presets computer players pitted against each other, which we would then use to implement and test our AI against.

Finally, we intended to build a basic learning algorithm that allows the AI to improve from game to game. Eventually the AI should then be able to determine whether there is any strategy that will lead to winning in most cases - and then perfect it.

3 Background

Although almost modern strategy game contains some sort of AI - mostly a collection of "if x, do y" statements and predefined strategies - there have been done very little in the direction of "learning" AIs. This has various reasons, like preserving processing power and reducing development time, but in general it's viewed as less important: Most strategy games focus on two areas - the heavily scripted single-player campaigns, where only AI is sparingly used, and multiplayer, where you don't need a central AI at all.

4 Solution

4.1 Requirements

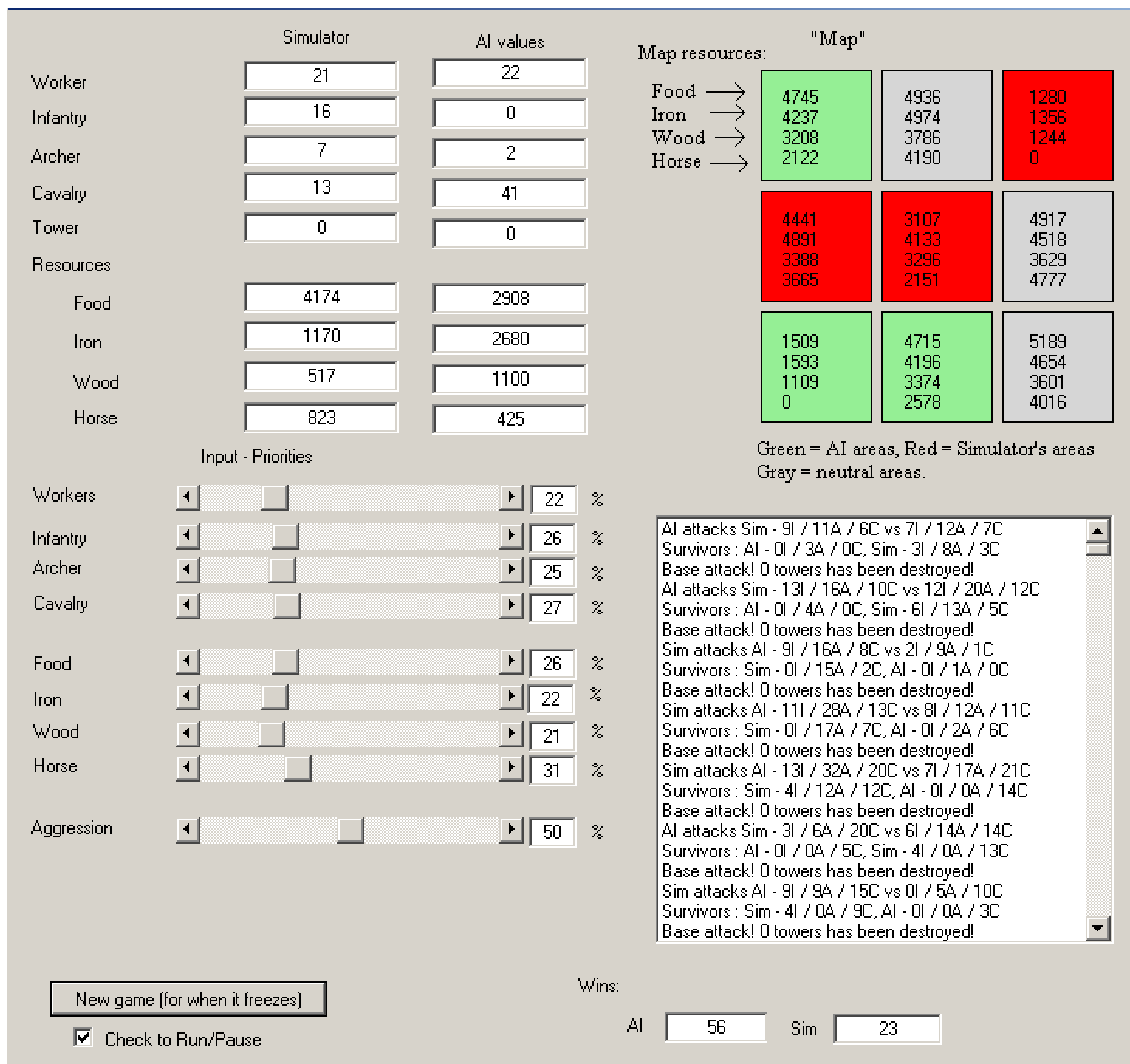
Requirements for the AI: We've divided the requirements into steps, where we moved from one to the next when the AI performed satisfactory in the current step. First, the learning AI should be able to beat a simulated opponent using the same tactic every game. Next it should be able to beat a simulated opponent that uses different tactics between games. The final step was to learn to beat an opponent which not only changes tactics between games, but also while the game is running. When all this was accomplished, the AI should be able to determine any unbalances in the simulated game in order to find a winning strategy able to beat any opponent.

4.2 Specification

The design consists of two parts: The simulator and the AI. The simulator will simply be a tool to update information, both to the human controller and to the AI, every "tick" of gametime. It will also take input, again both from the human controller and the AI, about, for example, what units is produced and destroyed, and in what amounts the simulated opponent should collect various resources, produce various units, etc. The AI, on the other hand, is reading the input from the simulator and, by mathematical formulas, attempting to decide the best adaptation to the current situation. Over time, the AI will learn what decisions are good and bad, influencing the results of the formulas to improve efficiency. Due to problems with finding proper means of "learning" how to deal with aggression and base attacking parameters, implementing AI reactions to these factors, as well as tests with the various extremes of these values had to be canceled. It was simply too difficult to predict the results of changing these variables, even with huge amounts of testing.

4.3 *Implementation*

The graphical interface of our simulator:



(fig. 1)

How the game works:
A map is generated with a semi-random amount of each of the resources Food (used to produce worker units), Iron (used to produce Infantry units), Wood (used to produce Archer units) and Horse (used to produce Cavalry units). Each player is then given one map section, identified as its "base", 5 worker units and 200 food. The amount of units each player can produce each "tick" of game-time is determined by the amount of expansions (extra map-pieces controlled by the player). How much resources a player can gather depends on its amount of worker units. Expanding is handled automatically when the home base starts to run out of resources.

When one player decide to initiate combat, a method is called calculating the result according to the following constants: Infantry does double damage to Cavalry, Cavalry does double damage to Archers,

and Archers do double damage to Infantry. Archers are cheaper to produce, but have a reduced relative attack strength. Cavalry, on the other hand, are more expensive to produce, but they have increased relative attack strength. The attacker brings all his units to the attack, while the defender gets a reduced percentage of his units, based on his amount of areas controlled. This is to simulate how the attacker has the benefit of initiative. When attacking the "base" square, the defender gets the benefit of 4 defensive structures that requires a large force to destroy. This is to make sure that one player cannot defeat the other at the very start of the game.

Implementation of the simulator: Using a timer to control the flow-speed of the game, we have created a simulator that does the following:

- ***Update resource stockpiles based on the gathering rates of each player and his number of worker units.***

Example: For the "Food" resource:

```
int foodGoal = workers * gatherRate * foodRate / 100;
for (int i = 0; i < expansions.Count; i++)
    tempRes = (int[])resourceMap[((int)aiExpsI[i]),((int)aiExpsJ[i])].Clone();
    if (tempRes[0] > 0){
        if (tempRes[0] > (foodGoal / foodBases)){
            currentFood += (foodGoal / foodBases);
            tempRes[0] -= (foodGoal / foodBases);
        } else {
            currentFood += tempRes[0];
            tempRes[0] = 0;
        }
    }
```

Where "foodBases" is the amount of map squares controlled by the player containing food resource.

- Update unit counts based on production rates, production capacity available and current resources available.

Example: Producing Infantry (simplified):

```
double countInfantry = (totalUnits * infantryRate / 100) - currentIinfantry;
double probInfantry = countInfantry / totals * 100;
int maxProduction = (productionPerBase * bases) + (productionPerExp * expansions);
while (maxProduction > 0 ) {
    double calc = random.Next(100);
    if (calc < probInfantry) {
        if (currentIron >= costInfantry) {
            currentIron -= costInfantry;
            currentInfantry++;
            maxProduction--;
        }
    }
}
```

- Dependant on current available resources, determine if a player requires an expansion, and if needed, expand to uncontrolled map pieces.

Example: Expanding to a random unoccupied map square (simplified):

```
expansionNeed = 1 - (((double)availableFood - (((double)resFoodPerSquare - 500) *
(bases + expansions))) / (double)simAvailableFood * (( Same for the other 3
resources )) ;

bool done;
if (random.Next(100) > expansionNeed*100) {
    done = false;
    while (!done) {
        rndI = random.Next(3);
        rndJ = random.Next(3);
        if (controlMap[rndI, rndJ] == 0) {
            controlMap[rndI, rndJ] = -2;
            expansions++;
            updateUnitMap();
            updateColorMap();
            done = true;
        }
    }
}
```

- Determine whom, if any, of the players gets to perform an attack, and then call methods executing the attack.

How it works, pseudo-code because a proper example would be several pages long:

```
f = random.Next(90) + 10;
X = player 1 attack level calculated on number of units and aggression;
Y = same for player 2;
q = coin flip, who gets to attack if both players "want" to;
if (f < X && f > Y || (f < X && (f < Y && q == 1))) {
    decide whether the player want to attack a base or expansion;
    calculate amount of defending units;
    call the "fight" method (see next paragraph);
} else if (f < Y) {
    same as above for the other player;
}
```

- Calculate the winner of an attack as described above, and if applicable change the map to give control of conquered areas to the attacker.

The "fight" algorithm (simplified):

Input: attacking and defending units, defensive structures.

For each unit:

```
tempKills = (int)(restAttackers[0] * targetFactor * relStrThisUnit);
restAttackers[0] = (int)(Math.Floor((tempKills - tempDefenders[2]) / (targetFactor *
relStrThisUnit)));
if (restAttackers[0] < 0) restAttackers[0] = 0;
tempDefenders[2] -= tempKills;
if (tempDefenders[2] < 0) tempDefenders[2] = 0;
```

Where targetFactor is decided by the current unit's relative strength towards particular units (higher value towards its target unit, lower towards its anti-unit), and relStrThisUnit is the relative strength towards all units, being higher for the more expensive cavalry, and lower for the less expensive archer.

This is then repeated for each unit until either it "runs out of steam" and all units have attacked, or the base is taken.

Implementation of the AI: Various methods being called during the game flow does the following:

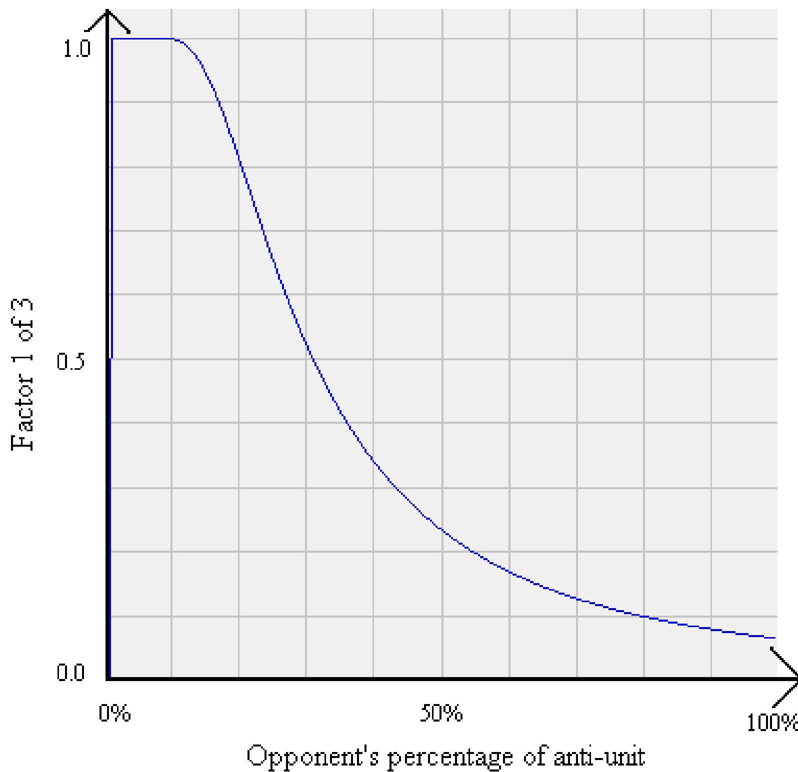
- Update resource gathering rates for the AI player based on current stockpiles and available resources

- Update unit production rates for the AI player based on the composition of enemy units and current stockpiles of the various resources needed to construct these units.

- Change the "lines" determining the outcome of the above calculations to improve performance over time. How the "lines" work is explained below.

Example of the calculations: AI formula for updating unit production rates:

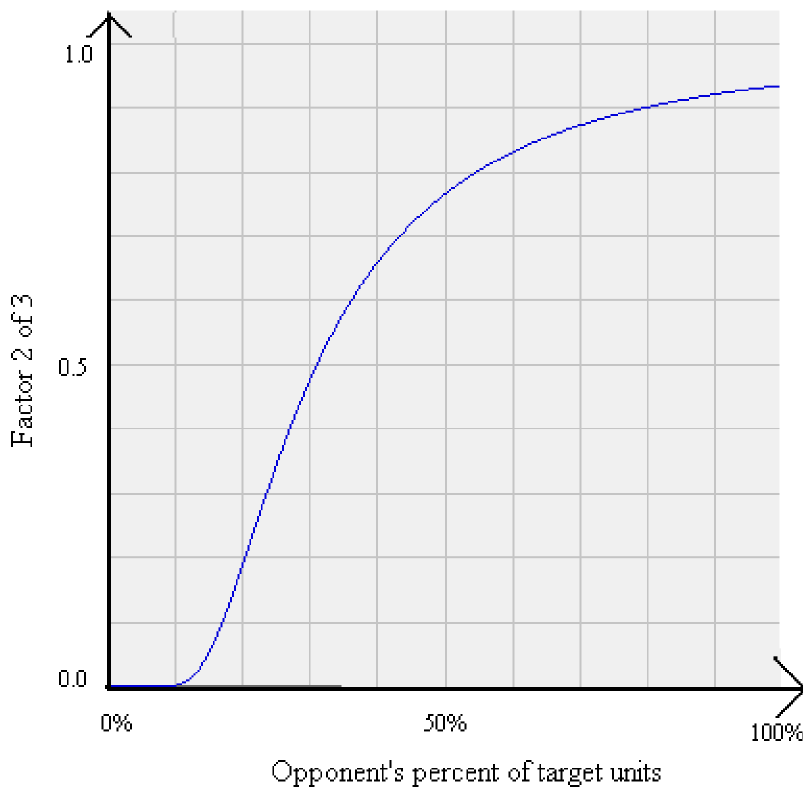
To calculate the factor at which the AI changes its unit production rates, the following factors are calculated from the opponent's current unit combination:



Formula:

$$1 - e^{\left(\frac{-1}{15*(\%\ \text{anti-unit})^2}\right)}$$

(fig. 2)

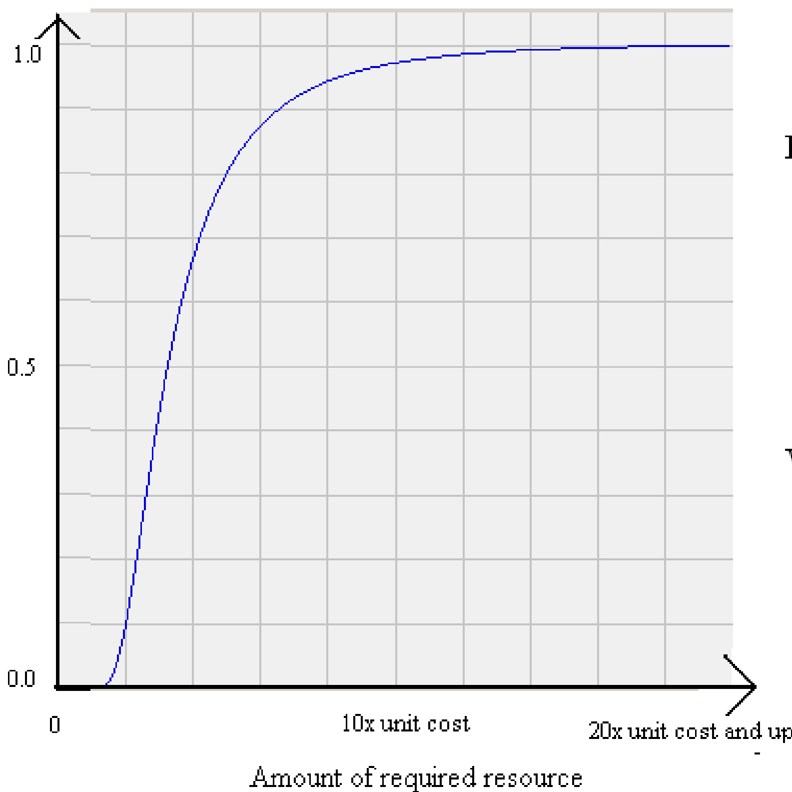


Formula:

$$e^{\left(\frac{-1}{15 * (\%\ \text{tar. unit})^2}\right)}$$

(fig. 3)

The average of these two are then multiplied by a factor based on the AI's current resource required to build the unit in question, to avoid building units the AI currently does not have resources for:



Formula:

$$X^{\left(\frac{1}{100 * X^2}\right)}$$

Where:

$$X = \frac{\text{amt. of req. resource}}{\text{unit cost} * 20}$$

(fig. 4)

The result is then compared to a "line", dividing the possible results into "increase" and "decrease" portions. If the result is above the "line", the rate will be increased by an amount relative to the

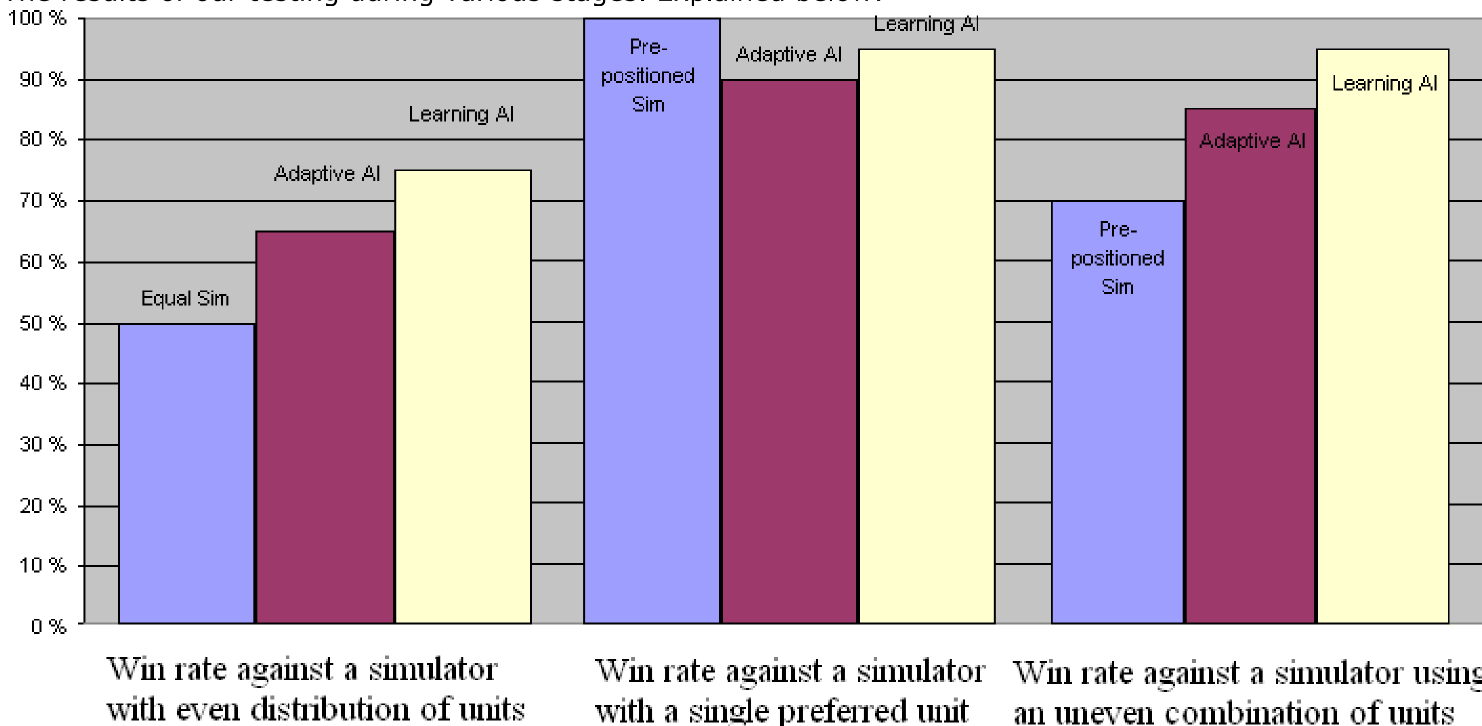
distance from the "line". If it is below the "line", it will naturally be reduced by the same relative amount. Initially the "lines" are set to 50% - giving equal chances of a given result being on either side of it.

This "line" is what is changed by the learning portion of the AI: When the AI decides that a particular action was beneficial throughout a "game", the "line" will be moved towards zero. This increases the size of the area that gives a positive change, and reduces the area that gives a negative change. It also increases the possible strength of the change, since the maximum distance from the result of the above calculation and the "line" is increased, whether it results in bigger increases or decreases in production rates.

The above example shows the calculations for a military unit. Worker unit rate calculations are based on stockpiled resources instead, to allow the AI to keep an even flow of the required resources for as long as possible. The rates for resource gathering is calculated by a combination of the current available resources (what the AI has left in the squares is currently controls) and how much resources of each type is required to keep producing the units according to the current unit production rates.

4.4 Validation and Testing

The results of our testing during various stages. Explained below:



(fig. 5)

The graphic shows the results of hundreds and hundreds of "games" played during various stages of AI completion. The blue column is pre-AI testing, showing the result of preset values on both players in the simulator. The purple column is the Adaptive AI - no learning, only direct reactions to the state of the game. The white column shows the results of the learning AI after some training sessions. All of these tests will be explained in detail below.

Early testing - no AI, only simulators playing each other:

Slight alterations / equal settings: In this test, two simulators are once again pitted against each other, this time with only minute or no differences in unit-production priorities. The results are bland, and depend heavily on what kind of unit the priority difference influence (which is an indication of a game balance problem), and the results also depend on what resources are in the expansions that the simulators capture. Building slightly more cavalry, for example, usually results in a higher win rate, indicating a slight imbalance towards cavalry being a superior unit.

Clear case: In this test, two simulators are pitted against each other, where one will focus on producing only one particular unit type (i.e: archer), while the other will build only the “anti-unit” (i.e: cavalry). The result shows that the simulator works as planned in such extreme cases: The simulator with the anti-units will win 100% of the simulated games. Even as such - again, going all out cavalry still gives the opponent a harder time, even when the appropriate anti-unit is produced. Again, a game-balancing problem.

Combinations: In this test, two simulators are again pitted against each other, where one will focus on producing two different unit types (i.e: archer & infantry), the other will produce anti-units to both these (here: cavalry and archer). Again the results show that the simulator works as planned with a win rate of around 70% in favor of the simulator with the anti-units. Because the first simulator also has the anti-unit to half of the second simulators army - the assumed "underdog" in this test has a slightly better chance at winning than in the clear cases. And as before, combinations with cavalry has a slightly larger chance of winning.

Continued testing - Adaptive AI playing the simulator:

Slight alterations / equal settings: Using the same setup as above, the AI now tried to adapt to the situation, finding that when its opponent built about equal amounts of units, there was not much to gain from just adapting. The results became more reliant on luck in the selection of attack targets and who ran out of resources first, but since the AI could adapt to running out of a particular resource, or to the fact that the opponent could only build certain units with its own resources, the AI scored more wins than the earlier tests.

Clear case: One would think that an adaptive AI would be perfect for taking on the clear case - single unit opponents should be easy to adapt to. And while the win rate is high, the AI has some troubles with a particular unit - again, cavalry shows that it is indeed too powerful. With 100% win rates against only infantry or only archers, only cavalry reduces the average win rate to around 90%, simply because the AI does not adapt fast enough to counter the cavalry-only army.

Combinations: In this case, the AI does a lot better compared to the preset simulator. Mostly due to the fact that these games last longer, and once the resources used by the opponent starts to run out, the AI gets the upper hand by adapting in order to use the resources that are left. But still - winning every game will not happen even here - the randomness in who takes what areas and the presence of cavalry is still messing with the test results.

Near the end - the Learning AI plays the simulator:

Slight alterations / equal settings: Finally the AI is starting to give some results. After a brief training sessions to get some starter values into the system, the learning AI starts shining. And why? Because it has figured out that cavalry is indeed too powerful. And uses this to great results. While it will not give top results in every game, due to possible resource shortage, focusing heavily on cavalry is allowing the AI to win more games even against the hardest opponent - the evenly balanced simulator.
Clear case: There is no real contest here. Even against opponents with heavy focus on cavalry the AI will win most games. Not quite a perfect record, but very close indeed. The fact that the AI also has learned that faster adaption is a good thing is another factor that increases win rate.

Combinations: Just like the clear cases - the AI knows what works and goes with it. Unless it runs out of the "horse" resource, the AI will use cavalry to good effect against most combinations and win most games.

Final tests - the Learning AI faces the challenge of an in-game adapting opponent:

For this test we had to slow the simulator down far enough for us to be able to adapt the AI's opponent as best we could during games. Because of this, we have not been able to test this part as thoroughly as we would prefer. That being said, a good dozen games gives a pretty good estimate. The learning AI, fully trained after the sessions above, achieve mostly the same win rates as in the "combinations" case above. No matter how quickly we alter the settings, the AI reacts fast enough to counter without losing an excessive amount of units. In addition, we are in fact adapting to the values we can see in the simulator, while the AI only adapts to the values it observes during combat. All in all, the results are surprisingly good in the AI's favor.

5 Discussion

First impressions: Testing the simulator.

According to the early testing - our simulator actually does a pretty good job at giving us the results we expect from two static opponents. In other words - the simulator is working as intended, or close to it. Some minor balancing changes, some of whom are already ready for implementation, and we have a solid base for testing our AI as it comes along. The main "problem" with the static simulators are that they don't work well with lacking a particular resource, or having little of a particular resource available for gathering - leading to reduced unit production.

Moving on: The adaptive AI takes on the simulator.

As we move on to testing out the adaptive AI, we quickly noticed that simple adaption to the current state of the game brings us vastly increased performance. Our AI actually performed so well during certain stages of this test phase we started to wonder if the "learning" part of the project was even needed at all. The AI responds quite well to most cases, mostly finding the evenly balanced enemy one of the harder ones to compete with, probably because it has problems finding a proper counter to an opponent who strives to have the same amount of all units.

The final touch: The AI gives learning a try.

After extensive testing of the learning algorithm using various setups, we finally received something resembling conclusive data. After several hundred games, the values the AI ended up with can be translated as follows: Cavalry is seriously overpowered, and favoring Cavalry will give decent results in almost every case. Archers are a good second choice when you run out of the "horse" resource, while Infantry is the worst choice in most situations. There is also a slight tendency towards favoring lowering the "lines" overall - leading to increased adaptation / faster changes. While we're not entirely sure these choices are the best ones, we can see how we have, unintentionally, made Cavalry overpowered, and it is a relief that the AI is much more able to detect it than us.

Sadly, the real effects on the gameplay from our attempts at learning is not showing as well as we would like. Ideally we would like to see a major increase in win rates for the AI after a long period of learning. The increase IS there, but it's not quite as strong as we would prefer. Why? Possibly because deciding what is a good or bad action in a constantly changing environment - where you cannot properly identify the cause of these changes from one point to another - is an nearly impossible task. It was therefore necessary to simplify this process by averaging out the AI's impression of what action was good or bad throughout the course of the game. This "cumulative benefit" variable is then used to determine how the rate lines change. Finally - what should we do with the difference between losses and wins? While it might seem a simple question the implementation is shaky at best.

6 Conclusion

It can be seen from our testing that have, indeed, accomplished the goal we were initially given - the AI observes and evaluates the game, determines what need is most pressing, and then acts on it. Our main goal has thus been reached. The secondary goal - achieving proper learning for the AI, has had some problems. While getting the AI to measure what it did well and what it did poorly is in itself a huge task, having it act accordingly has been found much harder than expected. Some goals, like adapting the aggression and base-attack variables, had to be scrapped due to problems with predicting results through statistical analysis. Relying on guesswork alone was not on our agenda for this project. Overall, however, we have seen a good deal of improvements in the performance of both our simulator and AI over the course of development. Both the game engine itself and the AI has grown more solid and more stable during the project period, and while the simulator is not a proper RTS game, the approximation is not that much off. Our greatest success is actually seeing how the AI decided that a particular unit was too good compared to the others, and then started leaning towards producing this unit more often - showing that the learning is indeed working as it should. This despite the fact that in our testing the AI is not as superior as we would like.

Appendices

Glossary & Abbreviations

AI	Artificial Intelligence
RTS	Real Time Strategy (game)
ORTS	Open-Source Real Time Strategy (game)

Attachments

The entire Visual Studio Project containing our project code. Actual classes and methods in the files "Form1.cs" and "AI.cs".