AI for real-time strategy games

Development of an AI for a real-time strategy game
capable of learning new strategies

Master thesis
By Anders Walther
June 2006

IT-University of Copenhagen
Design, Communication and Media

Supervisor:
Kim Steenstrup Pedersen
and Marco Loog

Table of contents

1 Introduction

When playing a real-time strategy game (RTS game) one usually has the possibility to play the single-player option, where you play against the computer or alternatively, one can play the multiplayer option, where you plays against other players, for example over the internet. In the multiplayer option of RTS games new strategies are continuously being developed and change the way the games are being played. Because of this, players keep finding this game option interesting and challenging. However, in the single-player part the artificial intelligence (AI) in these games doesn't learn these new strategies. It only draws from hard coded strategies provided by the programmer. Therefore, a human player quickly becomes superior to the computer AI and looses interest in the single player part of the game.

If the AI in these RTS games could adapt itself to new strategies and was able to learn new strategies on its own, the player would continue to find this part of the game interesting and would be able to try out new tactics before for example playing online against a “human opponent”.

1.1 Problem

This thesis will explore the possibilities for developing an AI for a RTS game that can learn to adapt itself to new strategies used by an opponent. In this way it is hoped that the single player mode of RTS games can continue to be as interesting and challenging as the multiplayer mode of RTS is.

Aim

The aim of this project is to develop an AI for an RTS game that can learn to counter new stratgies used by an opponent.

Goals

- Compare different AI theories and find the one best suited for developing an AI for a RTS game
- Design and implement the AI for a RTS game
- Integrate the developed AI with a RTS game
- Design and implement learning algorithms that can be used for training the AI
- Evaluate the AI and the learning algorithms by letting the AI play against other AI opponents

1.2 Planned Approach

First a general introduction to RTS games will be given. This will be followed by an analysis of the problem domain describing the different types of AIs in RTS games and some general AI theories. From this it will be chosen which type of AI to focus on and which theory to use for developing this AI. Furthermore it will be discussed whether to integrated the AI with an already existing RTS game or whether a new RTS should be made from scratch.

The next part of the thesis will be to get an understanding of the chosen theory and to prepare the RTS game into which the AI is to be integrated.

With this in place the development of the AI will begin. First, a model for the AI will be designed and implemented and secondly, algorithms for training the AI will be developed. All implementations will be done in Microsoft Visual Studio 2005.

In the last part of the thesis the AI will be evaluated through various experiments which are to show both how well the learning algorithms are at training the AI but also how well the chosen theory was for developing the model for the AI.

2 Background - The RTS genre

This section describes the RTS genre and will explain the elements, strategies and playing modes of a typical RTS game.

Elements

There are a lot of different elements in the game. First of all there is the map. Each game is played on a limited sized map that can be made of lots of different types of terrain. This can be grass, water, mountains, bridges etc. Some of these types of terrain will be possible for units in the game to move on. Others, for example a mountain or water, will be impossible for some unit types to move on. Often the terrain on a map will call for special types of units and special strategies to be used.

The second element in RTS games is resources which are located various places on the map. Depending on the game, resources can be minerals, gas, oil, trees, gold etc. Resources can normally be collected by certain unit types called workers and are used to build workers, buildings, military units or research technology.

The third element is units. In every RTS game there are a lot of different units which the player can choose to use in his army. Each unit type has different strengths and weaknesses and therefore, different game situations call for different unit types. For example a soldier might be strong against other soldiers but weak against air units which he cannot fire upon.

The last element in RTS games is buildings. There are often a lot of different buildings in RTS games which all have their own functionality, for example to train new military units, turn resources into money, supply your army with food or to research new technology.

Game play

A RTS game is seen from either an isometric or a 3D perspective.

In the game, players strive to eliminate their opponents from the map. This is done by building an army of various units and using this army in the battle against the opponent army.

The player can select units and buildings by clicking on them. Once selected, he can for example order a unit to move to a given position by clicking on that position on the map or make the unit attack another unit by clicking on that other unit. Once a certain building is selected the player can for example order this building to start producing various soldiers or to start researching new technology that will give the player the possibility of building new types of units and buildings.

Throughout the game, the player is in charge of setting up the economy (building for example workers), he decides which factories and which units should be constructed and he lays the overall strategy in the battle against the opponent. Also he has to control all the individual units in the game, where they go, which enemy units they should attack, which special abilities they should use and so on.

Strategies

Having a good strategy for how you will engage your opponent is a very central aspect of a RTS game. However, it is equally important to be able to realize when your strategy will not work against your opponent and to be able to change your strategy in such a situation.

In almost all RTS games there are three main strategies and counter strategies. One strategy is fast attack (rushing), where you try to take out your opponent after only few minutes of playing. Another strategy is to research new technology very fast which will allow you to build units that are much stronger than your opponents units. The third strategy is based on getting a huge economy up and running very fast which will allow you to build more units than your opponent later in the game. There is of course an endless number of variations and combinations of these general strategies and new strategies are continuously being developed.

Real-time aspect

Another central part of RTS games is the real-time aspect. Everything happens in real-time which means, that the player must control all his units, build factories and units etc. all at the same time. This means that he must be able to think and react really fast.

Fog of war

Fog of war is a very common characteristic of RTS games and means, that a player only can see the part of the map where he currently has either units or buildings placed. The rest of the map will be covered by a fog that makes it impossible for him to see what happens at those positions. Therefore it is important for a player to continue to scout the map to see what to enemy is doing.

Playing modes

There are normally three different modes of playing RTS games: a campaign mode, a single player skirmish mode and a multiplayer mode.

The campaign mode is only for single player gaming and contains a number of levels that the player must complete. These levels often differ a lot from each other. The mission on one level might for example be to defend a pre-build base for some amount of time and on another level, the mission could be the escort a special unit from one part of the map to another.

The levels in the campaign mode are built around a story that is told to the player through cut scenes between the levels. When playing a level the player is placed inside this story and needs to complete a level before he can move on to the next chapter of the story. Especially Starcaft and Warcraft by Blizzard are very famous for the story in the campaign mode of the games [32].

In the single player skirmish mode, a player can play against an AI controlled opponent. When playing a skirmish game, there are no special missions like in the campaign mode. The human player and the AI controlled opponent play under the same conditions; they start with the same amount of money, the same number of units and buildings and can during the game build the same type of units. A skirmish game can normally be 1 vs. 1, 2 vs. 2, 3 vs. 3 or 4 vs. 4 where all allies and opponents are controlled by an AI.

A multiplayer game is a lot like the skirmish game. The only difference is that the opponents are other human players.

There are lots of professional RTS players playing the multiplayer mode of RTS games. There are also a lot of different leagues and tournaments for RTS games and in some countries there are even TV-channels showing only RTS matches between human players [7].

3 Analysis

In RTS games the AI needs to control the individual units, decide what to build, lay the overall strategy etc. In this chapter different types of AIs for RTS games as well as theories for developing AIs are described. From both the different types of AIs and the different theories one is chosen for this project. Furthermore it is discussed whether to integrate the developed AI in an already existing RTS game or whether to construct a new RTS from scratch.

3.1 The AI in RTS games

As described in the previous chapter there are so many different elements in a RTS game that need to be controlled, that it is perhaps a bit vague to simply talk about an 'AI for an RTS game'. Brian Schwarb [11] argues for a number of areas in RTS games which need to be controlled by an AI. In the following section I will shortly describe and exemplify these different types of AI's and decide on which of these types of AI's this project is going to focus on.

Individual units behaviour AI

One of the basic roles of the AI is to control the behaviour of individual units. This for one thing includes path finding but it also includes the behaviour of units in a battle, for example make a unit fight back when attacked by enemy units, make a unit attack enemy units when ordered to attack enemy units in a given area and so on. How intelligent the individual units are varies from game to game. Also players can overwrite the individual unit behaviour and take control of all the units themselves.

Some players prefer micromanagement while others prefer macromanagement. Micromanagement includes among other things moving units to avoid enemy fire, making all units focus their fire on selected enemy units or surrounding enemy units so they cannot escape.

Players who prefer macromanagement build up huge armies and then simply order the entire army to attack enemy units in a given area, without for example specifying specific targets.

Therefore some balance of the individual units behaviour is needed that will be useful for both players preferring micromanagement and those preferring macromanagement.

AI for worker unit behaviour

The worker unit type is special because it normally is very poor at fighting. Therefore an AI is needed that will make the workers run away from battle when attacked or at least try to fight back the best they can though their normal job is to harvest resources [11].

However players who prefer micromanagement will have different desires with regards to worker units than players preferring macromanagement. In Starcraft and Warcraft [15] for example, workers can be used to block the enemies when they attack your base. When attacking your base, the enemy units will first of all try to destroy your military units and guard towers. However, if you place your workers between the enemy units and your military units and guard towers, the enemy units will not be able to attack your units but can be shot upon while trying to find a way to your units. This is especially useful when the enemy units are melee units (units that prefer close combat to ranged combat).

However, if you are not fast enough to do that kind of micromanagement, your workers will simply continue to harvest and will be an easy target for the enemy units, once they have destroyed your defence, which of course, is not desirable. So, where players doing micromanagement need full control of their workers, other players might call for some automated battle avoidance from the workers.

Medium-level strategic AI

This AI is in control of 5-30 units [11] and is created when the higher level AI decides to for instance attack the enemy base. The primary role of this AI is to make sure that the attack takes all units strengths and weaknesses into consideration (both own and enemies) to make sure that the army is not wasted by for example letting 20 footmen walk into a base defended by snipers.

Where the individual unit behaviour is the same for both human players and computer players, the medium-level strategic AI is normally only needed by the computer player, since the human player will want to direct the attack himself. However, in some games like Total Annihilation [12] or Dawn of War [13] a special commander unit type can be used to bolster groups of units [11].

High-level strategic AI

Similar to the medium-level strategic AI this AI is only needed by the computer player and is in charge of the overall strategy; should we attack fast, grow a strong defence, go for fast research and so on. It is also responsible for deciding how the enemy should be attacked (by air units, ground units, stealth attack etc.). In most parts of the game this high-level strategic AI determines a specific playing-style. Some might value offence and a strong economy while others might be cautious and studious [11]. By changing the high-level strategic AI the computer controlled opponent can be varied, thereby giving the human player different playing experiences from game to game.

Town building AI

The AI also needs to know where to build new buildings, which in some RTS games actually is a very important part of the game. On many maps in Starcraft for example, a player can choose to block the entrance to his base right from the start of the game by placing buildings on the path leading to his base. This will give him time to build up a strong economy or research new technology very fast, without needing to use money on defensive units. However, this will give his opponent(s) the opportunity to take control of large areas of the map. In other games such as Generals [16] the preferred base design changes depending on the length of the game. In the beginning of a game, it is an advantage to place the buildings close together since this will make it easier to defend it from enemy attack. Later in the game however, closely placed buildings will be very vulnerable to enemy mass destruction weapons which are capable of destroying buildings and units in a large area.

Another example where a town building AI is needed is with placement of economy buildings. They need to be placed close to the resources on the map [11] so that workers harvesting the resources will not have to go a long way to deliver the collected resources.

Terrain analysis AI

Maps in RTS games can differ a lot from each other. Some might be open deserts or consist of islands and some might have lots of resources scattered around the map while others have only few resources located at some central places on the map.

Therefore the AI needs to make an analysis of the terrain to be able to determine how to defend its base, how to attack the enemy, how to gather resources and so on. This analysis is normally done by sending cheap units out to explore the map and should be done continuously since all players affect the map when moving units, building towns, harvesting resources etc.

Opponent modelling AI

Opponent modelling is the process of keeping track of what units the enemy builds, when he builds them, when he normally attacks, how he normally attacks and so on. This will give the computer information about its opponent that it can use to adapt itself to the playing style of its opponent.

Resource management AI

Resource management is a vital aspect of RTS games and is about gathering resources and choosing whether to spend the collected resources on units, buildings or technology.

In every game the player must carefully decide what to build in a way that will give him both a stronger economy and a stronger army than the opponent has. Furthermore he must make sure to produce the best type of units for a given situation. If he builds units that are weak when fighting against the unit types that his opponent is producing, his units will not last long in battle.

AI controlled players often have specific build orders [11]. This will make sure that the computer continues to be a threat since this avoids, that the computer builds some buildings or units that are useless at a given time of the game. This build order however also makes the computer player somewhat predictable and easy to defeat if a weakness in the build order can be found.

Human players also use specific build orders, which they have found to be optimal in a given situation, for example when doing a fast attack. However, human players will have a lot more build orders to choose from, since new variations of build orders often will be discovered that tune the strategy a bit. At the same time, the human player is capable of changing the build order when discovering that the strategy being followed is no good against the strategy chosen by the opponent. This is something the computer cannot do when blindly following a predefined build order.

Instead of blindly following a build order, a need-based system could be used so that AI opponents would bias heavily towards certain units or resources and would rely much more on map type and personality [11].

Reconnaissance AI

Reconnaissance is about finding out what is happening on the map. As mentioned earlier there is normally what is called ‘fog of war’ in RTS games. Fog of war means, that a player can only see the areas of the map where he currently has either buildings or units positioned. Since the rest of the map is invisible for him, enemy units can move around the map without being detected and therefore reconnaissance is essential for finding out where the enemy units are positioned and if possible what the enemies plans are.

Cheating AI

A point that has not been discussed yet is the possibility for the AI to be cheating, which is already happening in a lot of RTS games. For instance, it could be chosen to let the AI be unaffected by the fog of war mentioned earlier. This would mean that the AI always was aware of what the opponents were doing and would therefore not need the reconnaissance component.

The same is possible with the terrain analysis AI where the AI could know the entire map as soon as the game starts and with the town building AI where the AI could have pre-planned building sites for each map.

The main reason for choosing the possibility of letting the AI cheat is that it makes the development of the AI less complicated since a lot of the components can be removed from it. Furthermore this is a way of making the AI put up a good fight against even very good human players.

The problem with this solution of course is that the player might feel that he is playing an unfair match when the AI opponent always seems to know exactly what his plans are.

3.1.1 Sub-conclusion for AI in RTS games

First of all it is important to note, that the different AI components for RTS games described in this chapter are not a finite list of AI components that all RTS games developed so far have been following. For instance Bob Scott [17] describes a slightly different AI structure that was used for the game Empire Earth [18]. However, though the different components might overlap and have slightly different responsibilities from architecture to architecture, the components presented in this section give a good overall picture of the responsibilities a general AI has in RTS games.

Due to the timeframe of this thesis, it will not be possible to develop all of these different AI components. Instead it has been chosen to focus only on the AI for resource management. The reason for this is first of all that resource management is a very important foundation for almost every other part of the game. If you have a strong economy and build the right units for a given situation, you will stand a good chance of winning the game, even though you might not be very good at micromanagement. In spite of this, AI in many RTS games simply rely on the build order as mentioned earlier. This results in very predictable computer-controlled opponents who never learn new build orders and which can easily be beaten by a human player once he has found some weaknesses in the AIs build orders.

The second reason for choosing resource management AI is that the theory about this subject is not very well documented in literature. There are several examples, where implementation of a resource management AI is mentioned but not explained in detail.

For instance though Brian Schwarb [l1] argues for a resource management AI based on a need-based system, he never starts discussing how this system could be implemented and has no references to games where this system has actually been implemented.

As mentioned, Bob Scott [17] describes the architecture of the AI that was made for Empire Earth. In this architecture there are a unit and a build manager responsible for producing new units and buildings, but the decision of what to build is hidden in a combat-manager and is not further discussed.

The book Strategy Game Programming by Todd Barron [19] which promises to include “everything necessary for you to create a strategy game” only discusses the path finding part of the RTS AI.

The third reason for choosing resource management AI is that it can be implemented and tested in a closed environment. The AI will only rely on some input from the game situation or from the developer and will, as a result, return information about which unit or building to build. It doesn’t require a full RTS game up and running before it can proof its worth as for example the medium-strategic level AI would do.

3.2 Theories

In this section different theories for developing the resource management AI will be described and later compared to determine, which of them is most suitable for developing the AI in this project.

3.2.1 Neural Networks

Neural Networks is a well known method for doing AI and is described in many books dealing with AI for games. Neural Networks are build in a way that simulates the way a human brain is constructed by using many small building blocks called artificial neurons [20].

An artificial neuron has input and output. Inputs normally reflect the state of the problem domain and the number of inputs to a neuron can vary a lot from network to network. Each input has a certain weight added to it. This weight specifies how much a particular input should count for inside the neuron. Inside the neuron is an activation function that is used to determine the output value for the neuron. In this activation function, all the inputs are multiplied with their respective weights and are then added together. If the total value is larger than some activation value, the neuron will set its output to a value showing that it was activated, otherwise it will be set to some value showing that the neuron was not activated (this could be a 1 or a 0). This value is then sent on to other neurons or sent out of the network as one of the total outputs of the network.

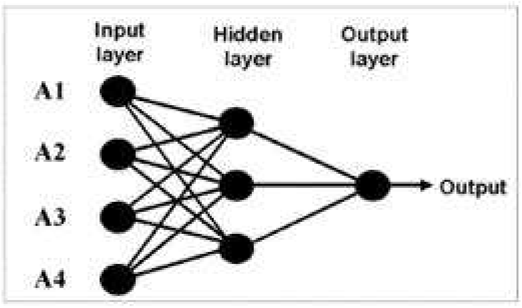


Figure 3-1 A feedforward network

The most widely used way to connect neurons is to use layers. The network in **Fejl! Henvisningskilde ikke fundet.** is called a 'feedforward network' since each layer of neurons feeds its output into the next layer of neurons until an output is given [20]. As can be seen from the figure the feedforward network has a layer of neurons called the hidden layer. This layer is simply a layer of neurons lying between the inputs and the outputs. In fact, there can be as many hidden layers as one may feel is necessary, including not having hidden layers at all.

A neural network can be trained for example to recognize a pattern. This is done by adjusting the weights in the network. A method for doing this is the 'back propagation' method. In this method, the output value from the network is compared to the target output value. The difference between these two is called the error value. This error value is passed on to the output layer where the weights are adjusted according to the error value. This process then continues through the hidden layers by calculating the difference between the hidden layer next to the output layer and the adjusted output layer. The weights in this layer are then adjusted and so on.

A resource management AI based on Neural Networks could be made by letting the input values be information about the current game state and the output values be either building or unit requests.

One problem with using this method is that it might seem like a black box. We give the box some inputs and it returns some outputs. However, to track down why a given unit or building request was made becomes difficult because of all the different weights and connections between neurons in the network, and therefore tweaking of the AI becomes a difficult task.

3.2.2 Genetic algorithms

Genetic algorithms exploit the way evolution takes place in nature: Only the fittest members of a species survive. This ensures that only the DNA from the fittest members of a generation is passed on to the next generation, which results in the strongest possible generation being created. Furthermore, small errors will occur from time to time when passing DNA from a parent to a child. This little error might have some negative effect on the child, which will then have a smaller chance for passing its DNA on to the next generation. However, in some rare situations, these errors might result in positive effects for the child, making it fitter than all the other individuals from its generation and this might increase its chance for passing its DNA to a new generation.

In genetic algorithms different solutions to a problem are encoded for example in form of a sequence of binary numbers. These solutions are then put to solve a problem and those that perform best are selected for producing the next generation of solutions. This next generation is made by taking data from one solution and combining it with data from another solution. For example, if the solution is given by an array of ten binary numbers, a new child solution could be generated by taking the first 5 binary numbers from one solution and the last 5 from another. To ensure evolution, small changes will be made from time to time in this child. This could be by changing one bit from being a ‘0’ to being a ‘1’, or it could be to change the position of some bits.

After a lot of generations, hopefully evolution will have come up with a good solution to the problem.

Genetic algorithms were used with great success in the game Nero [33] where soldiers can be trained for war by letting them do specific tasks a large number of times and continuously producing new soldiers based on those who performed best from the previous generation. However, a problem with using genetic algorithms for the resource management AI is that we would need entire populations of resource management AIs from which the best could be selected for producing a new population. While it is not a problem when training individual soldiers, who could all be put in the same game it would require that many games were run simultaneously when dealing with resource management AIs. Also a fitness function would need to be created for determining the quality of each resource management AI. But what exactly is the fitness of a resource management AI? Is it the number of wins in a row? The number of units lost during a game? The time it takes to win over a given opponent? Etc.

3.2.3 State machines

A finite state machine is a commonly used technique for letting an object behave differently in different situations and can, if wanted, be implemented with a simple switch statement. A state machine consists of four main elements [5]:

- states which define behaviour,
- state transitions which are movement from one state to another,
- rules or conditions which must be met to allow a state transition,
- input events, which are either externally or internally generated, and which may possibly trigger rules.

For example, a soldier can have the states `healthy`, `wounded` and `dead`. If he is in the state `healthy` he will engage the enemy soldiers in battle without thinking to much about avoiding fire. When his health drops below 50%, the rule for changing to state `wounded` will be met and the transition will occur. Now he will not blindly attack the enemy but will try to find cover to avoid enemy fire. If his health drops to 0% a transition to state `dead` will occur and he will no longer be able to shoot at the enemy.

In this way the behaviour of a soldier can be controlled. A problem with this technique is that it might result in very predictable behaviour [5]. A solution could be to use nondeterministic state machines instead where transitions between states are less predictable. For example, the soldier can change to state `wounded` if his health drops below 50%, but if it drops below 60% there can be a small change that he might already change state at this time. This could be determined by either throwing a dice (random number) or by for example using fuzzy logic.

The state machine could as such be used as the method for doing the resource management. For example could one state could be `BarrackPresent` which would allow for producing soldiers. However, since there are normally lots of different buildings and units in RTS games, the number of states needed would be enormous.

3.2.4 Fuzzy logic

A rule in normal logic is written in the form '`if A then B else C`'. Here the condition `A` must be exactly met if `B` is to be carried out. This means that condition `A` must be specified in a way that makes it possible to clearly answer yes or now. If we for example want to know whether a man is tall or not, we will clearly have to specify what tall implies. If the limit is set to being 180 cm, a person who is 179.99 cm tall will not be seen as being tall.

In fuzzy logic every statement can be thought of as being true to some degree. If something is said to be true to a degree of 1 it is absolutely true while 0 means absolutely false [6]. Anything in between is true to some extend. For example a person that is 160 cm could be around 0.80 degree tall while a person that was 210 cm tall would be tall to a degree of 1.

Fuzzy logic works by first converting an input value into a degree of truth for a number of sets. These sets could be short, medium and tall and will probably overlap a bit, which means that the input value can belong to one or more of them. There are functions for turning the input value into degrees of truth for each set. These functions are called member functions and can vary a lot from each other, which will affect the resulting degree of truth. Examples of member functions are the triangular membership function, the reverse grade membership function and the trapezoid membership function [6].

Once the degree of truth has been found for each of the sets, these degrees of truth are used to make the premise for some fuzzy rules. Such a premise could be

‘if(thinANDtall)'.

In normal logic we use if-else statements to choose which rule to fire and only one rule will be fired.

In fuzzy logic however, all rules fire, but the result of a rule will have a certain degree of truth. If for example the result of the ‘if(thinANDtall)’ rule is an eat-more variable, then this variable will be set to some value between 0 and 1 depending on the operands thin and tall in the premise for the rule.

The actual value for the eat-more variable will be found in some function evaluating or comparing the two operands.

Last the results from the rules are used to find the output value. For example the eat-more variable might have the value 0.6 while an eat-less variable has the value 0.3. These values are sent into some output member function which determines the actual output. In our example this could be 0.45 meaning that the person should start eating a little more.

3.2.5 Bayesian Networks

Bayesian networks are graphs that compactly represent the relationship between random variables for a given problem [8]. Using these graphs can help us make decisions when too much uncertainty makes it difficult to find the optimal solution to a problem. Bayesian networks use conditional probability and Baye’s rule [9] to find the best solution to a problem given what we might know about the problem. For example as a football coach, we might know which tactic our opponents´ coach normally sets for his team and which players he has available. Based on this information we can find the best possible counter tactic for our team.

Bayesian networks use nodes to represent random variables and arcs to represent the relationship between these variables. A random variable represents some feature of the entity being modelled [10] and can be in some number of states. The aim of Bayesian networks is to find the most likely state for each random variable. For example a random variable could be opponentStrategy which has the states 3-4-3, 4-4-2 and 2-4-4 telling us the likely formation of the opponent team. A random variable can depend on the value of other random variables. This is represented by an arch pointing from the variable causing the effect and to the variable which is affected. This means that the state of the parent variable will affect the state of the child variable.

We can use Baye’s rule to find the probability of a variable being in a certain state given what we know about its parent variables. The definition of the rule is

$$P(A|B) = P(A)*P(B|A) / P(B)$$

Which means that the probability of the variable A given the variable B is equal to the probability of A times the probability of B given A divided by the probability of B. In this way we can find the probability of some random variable being in a particular state even though we can not measure this directly. As mentioned, this is useful in for example games where the AI has to make decisions based on incomplete information about the current game situation.

3.2.6 Sub-Conclusion on choosing the theory

The AI theories described in this section were neural networks, generic algorithms, the state machine, fuzzy logic and Bayesian networks. As described in 3.1, the project will focus on developing an AI for resource management, where the AI shall be used to determine which units to build during the game. For this purpose Bayesian Networks seem most suitable for several reasons. For one thing in RTS games players always have to make decision based on a lot of uncertainty, which is exactly what Bayesian Networks are good at and which therefore makes Bayesian networks an obvious choice. Another reason is that Bayesian Networks are interesting from an academic point of view. Where neural networks, generic algorithms, fuzzy logic and state machines all seem to be well established AI methods in (RTS) game development, Bayesian Networks seem to be a less used and less documented method. Whether this is because Bayesian Networks, in spite of the otherwise obvious benefits from using Bayesian Networks in RTS games, isn't such a suitable method for AI in RTS games after all, or whether game developers simply rely on more well documented and established methods is a very interesting question that will be tried answered through this project.

3.3 Learning

This section will describe different approaches to AI-learning.

3.3.1 Supervised learning

Supervised learning works just like learning in schools: Students are taught by their teacher – they are given knowledge. When given a question they can use this knowledge (input) to come from that question to a specific answer (output). The teacher is capable of informing the students whether they answered right or wrong, since he knows the correct output to a given input.

In machine learning in general a function must be learned that can find the correct output to certain input. A method for supervised learning is the back propagation method for learning in Neural Networks. This could for instance be to get the network to recognize a specific pattern. Here the output from the network is compared with the desired output and if they do not match, corrections are made to the network.

3.3.2 Unsupervised learning

In unsupervised learning no predefined output is available for specific input and therefore the output coming from e.g. a Neural Network cannot be categorized as being either correct or false. Instead patterns in the input must be learned so that each input can be determined to belong to a specific class. Unsupervised learning can therefore be used in situations where limited information about a given problem is available.

3.3.3 Reinforcement learning

Another type of learning is reinforcement learning. In reinforcement learning there is no predefined correct output to a specific input. Instead an agent must learn the best possible output by using trial and error. However, for each action chosen the agent will be informed whether this action led to something positive or something negative. This information will be represented as a reward where high values mean that the action led to something positive and small rewards mean that the action led to something negative. In this way the agent will learn which actions give the highest reward and can in this way learn the best solution to a problem, i.e. the optimal correspondence between input and output.

3.3.4 Online vs. offline learning

The above described learning approaches can be applied as online learning or offline learning. The two terms, online and offline learning can be defined differently:

- In one situation, online and offline learning refers to whether the learning takes place *while* the game is being played (online) or *after* the game has finished (offline).
- The other definition of the two terms refers to whether learning takes place *before* the game has been published (e.g. whilst developing the game; offline) or whether learning takes place *after* it has been published, meaning when the game is being played (online).

For this project the later definition is used.

The difference between online and offline learning using this definition affects the requirements to the AI. Though it is important that learning happens fast in offline learning, it is a very essential aspect in online learning. The players of RTS games develop new strategies and playing styles rapidly and therefore it will not be a useful solution if the AI needs a thousand games to adapt itself to the playing style of a new opponent. The AI should be able to do so in relatively few games. Even though there probably are strong time limitations in the development phase of a game, a slower convergence rate will be more acceptable in an offline learning situation.

Another important aspect is the needed computation power when learning. In an online learning environment the learning needs to be done without the player taking notice of it and the computational resources are limited in games. In contrast, a more heavy computational process can be allowed in an offline learning situation.

3.3.5 Sub-conclusion on learning

This section described different learning approaches: supervised learning, unsupervised learning and reinforcement learning, as well as the two possibilities of how to apply them – online or offline learning.

Since the aim of this project is to develop an AI that can adapt itself to new playing styles and strategies, learning methods are needed that will be fast and effective enough to be used as online learning. Whether the developed learning methods in fact will be that effective, or whether too much computational resources will be needed before a satisfying result can be reached will be discussed in later chapters.

Choosing online learning will affect the choice of learning approach. It will be difficult to use supervised learning since the learning cannot be controlled by a supervisor, who knows the correct input-output pairs. Unsupervised learning could be used to e.g. let the AI determine whether the current combination of enemy units is an indication for whether he is doing a fast attack or trying to get a strong economy etc. However, as the strategies develop, these patterns (e.g. fast attack vs. strong economy) might start to overlap and new patterns may need to be found before the AI can determine what strategy the opponent is using. Because of this, the unsupervised learning paradigm does not seem like the optimal solution.

In reinforcement learning the AI can learn to adapt to new playing styles through the feedback it receives on its actions, for example informing the AI whether it won or lost the game. Therefore this learning approach seems ideal for an online environment and shall be used for this project.

3.4 To use an existing RTS game or to implement a new RTS game?

To be able to test the capabilities of the AI implemented in this project, the AI needs to be integrated in some RTS game. This RTS game could either be an existing game with an accessible source code, or it could be a game implemented for this project only.

Working with an existing game would remove some of the work that implementing an entire game requires. However, when using an existing game, the AI developed in this thesis would need to work together with all other AIs in that game (path finding, high-level strategy etc. etc.) and this could result in some difficulty when evaluating the AIs capabilities. Integrating the AI in an existing game would also require time to get to understand the game engine and to integrate the AI with all necessary components of the game.

It has therefore been chosen to design and implement a small RTS game with the specific purpose of testing the resource management AI. This includes several variations from the normal definition of an RTS game.

First of all RTS games normally have what could be called a space-aspect. The player can explore the map of the game and can use different tactics that best take advantage of the terrain. Furthermore the player has to make decisions regarding placement of new buildings and needs to control his units to make them avoid enemy fire, attack the enemy units etc. Implementing this space-aspect would require a lot of work and the question therefore is whether the resource management AI could be developed and evaluation for a RTS game where this space-aspect had not been implement.

The space-aspect is the area of RTS games where the terrain-analysis AI, the medium and high-level strategic AI and the individual behaviour AI would be needed. However this space-aspect does not directly influence the resource management AI. Instead the resource management AI is influenced by the unit and building requests coming from the medium- and high-level strategic AI. These analysis AI components normally depend a lot on the space-aspect and the resource management AI is therefore indirectly affected by the space-aspect. However, if the analysis-components were exchanged with analysis components not depending on the space-aspect, the resource management would not be particularly effected if the space-aspect was removed from the game. It has therefore been chosen to implement the RTS game without the space-aspect and instead simulate the medium and high-level AIs by directly providing the resource management AI with game information.

Removing the space-aspect will change the interface of the game. As mentioned in the description of RTS games, the user can normally select units and buildings and order them to perform specific tasks. Removing the space-aspect of course also removes this form for interaction. Instead the interface is going to consists of buttons that the user can press in order to build more units or buildings and some text areas displaying the current number of the different units and buildings and their individual health.

These changes of course raise the question whether the game in fact will continue to be an RTS game. The game will still be real-time and will still be a strategic game and could therefore still be called and RTS game - though it might not be exactly what people normally would consider to be an RTS game. However, this question is not really important. The important part is that the resource management AI will have exactly the same functionality as in a normal RTS game and hopefully the lessons learned in this project will be usable for ‘real’ RTS game projects.

Chapter 5 will focus on the design of a small RTS game with no space-aspect.

3.5 Analysis conclusion

In this chapter the different types of AI in standard RTS game were described and, based on that, it was chosen to focus this project on the resource management AI.

To be able to test the developed resource management AI it was chosen to implement a small RTS game only focusing on the elements of an RTS game important for the resource management AI.

Last it was chosen to use probability theory, namely Bayesian Networks, for developing the AI and to use reinforcement learning as the method for training the network (AI).

4 Theory

The following sections will cover some of the theory concerning probabilities and Bayesian Networks - which is described in ‘AI for Game Developers’ [8, 9] and Learning Bayesian Networks [2] - as well as theory concerning reinforcement learning described in ‘AI Game development’ [26] and ‘Reinforcement Learning: A Tutorial [25]. This theory will be needed for developing the resource management AI using Bayesian Networks and Reinforcement Learning.

4.1 Probability Theory

We know that two and two makes four and we know that if we throw an apple, the gravity will eventually pull it back down to earth again. There are however other things that we cannot know for sure. For example will it be raining tomorrow or will it just be cloudy, will the trains be delayed today and will there be a ticket control in the trains? Though we might have a feeling or a strong belief that one of these situations might occur we cannot know for sure. Instead we must rely on probability theory to get an idea of what is most likely to happen.

We use probability theory in general and in games when there is something we are uncertain about. For instance a player that stays in his base in an RTS game and does not explore the map might be going for fast technology while a player who explores the map early might be going for a fast attack. However, we cannot be sure of either. If we for example have discovered that our opponent is exploring the map very early in the game we might conclude that it is very likely that he will do a fast attack and therefore start building defences. On the other hand he might just be checking whether we are going for technology or a fast attack and if we then use resources on our defences he might get an advantage by going for fast technology.

In these situations we have to use probability. Is the probability of our opponent going for a fast attack larger than the probability of him going for fast technology? The distribution showing these probabilities can then be used when deciding whether to prepare for the attack or whether we should use the money on something else than defence.

4.1.1 The sample space and the probability function

A sample space is the set of elements representing all the possible values for a given situation and is represented by the notation Ω. For example, when dealing with probabilities for drawing certain cards from a normal deck of cards, the sample space will be the 52 cards in the deck. Any subset of the sample space is called an event and the elements in an event are the possible outcome of that event. An example of this could be the event of drawing a card of type hearts. Here the events are those of the 52 cards that are of type hearts, and the elements in this event are the individual cards of type hearts.

Furthermore, the set of all possible events including the empty set Ø and all complementary events are called the sigma-algebra or σ-algebra of Ω. For our card game σ would include the hearts, spades, diamonds, clubs, all aces, all but hearts, all but diamonds, all red cards etc. etc.

A **probability function** P() is a function of the events in the sample space and determines the probability of a given event occurring. So if we have an event *Hearts* consisting of the elements

$$\mathit{Heart} = \{h1, h2, h3...h13\}$$

then

$$P(\mathit{Hearts}) = 1/4 \text{ and}$$

$$P(\{h1\})+P(\{h2\})+P(\{h3\}).... P(\{h13\}) = 1/4$$

That is, a probability function assigns a number to all events.

Furthermore we also talk about a **probability space**. A probability space is the triple{Ω, σ, P} where Ω as mentioned is the sample space, σ is the set of all possible events in Ω including the empty set Ø and all complementary events and last P is a probability assigned to each of the events in σ.

In probability theory in general we have that the probability of some event occurring must be between 0 and 1 and the sum of the probabilities of all the elements in the sample space must be 1.

For example to draw the 10 of hearts from a normal deck of cards is 1/52. We write this as

$$P(E) = n/N = 1/52$$

since there is only one 10 of hearts out of the 52 possible cards.

If we instead want the possibility of drawing all cards but the 10 of hearts from a normal deck of cards we will get

$$P(F) = n/N = 51/52$$

This event is actually the same as the probability of the first event not occurring and therefore:

$$P(F) = 1-P(E)$$

For the event of drawing the 10 of hearts we therefore have that

$$p_{success} = P(E)$$

$$p_{failure} = 1- P(E)$$

$$p_{success} + p_{failure} = 1$$

.

4.1.2 Frequency interpretation

Probabilities can be interpreted in different ways. One way is the frequency interpretation where the probability of an event getting the value n out of N possible outcomes is determined by repeating the same experiment a large number of times.

If we for example draw a card from a normal deck of cards 100 times we could get that the probability of drawing the 10 of hearts was 5/52 or 0. However, if the experiment was repeated a many times, the probability of drawing the 10 of hearts would get closer and closer to 1/52. Or said in another way, if the experiment is performed N times and the 10 of hearts is drawn n out of these N times, the probability P(E) of drawing the ten of hearts would be:

$$P(E) = n/N = 1/52,\ N \to \infty$$

4.1.3 Subjective interpretation

Subjective interpretation is a person's degree of belief that a particular event will occur given their knowledge, experience or judgement [1]. For example if it snows we might have a strong belief that the trains are going to be delayed or that the children will be out playing.

We use subjective interpretation when we cannot repeat the exact same situation. That is, though we might have experienced snow before, something might be different on this day of snowing. There might be more train drivers on duty, there might be fewer passengers or the train company might have installed some anti-snow system that enables the trains to drive as fast as normal in spite of the snow. We therefore cannot use frequency interpretation, since we can not repeat the exact same experiment a large number of times.

However, through experience we have learned that though the trains might be on schedule today, it is very likely that the trains will be delayed.

On the other hand, one of our friends might have been lucky with the trains on previous snowy days and therefore will have a stronger belief in the trains being on schedule than we have.

In this way the interpretation is subjective and might differ from person to person.

4.1.4 Probability rules

This section will shortly describe some probability rules that we need to know before moving on to conditional probability.

If A and B are two mutually exclusive events, only one of them can occur at a given time. The probability of one of these events occurring is:

$$P(AUB) = P(A) + P(B)$$

If the two events are not mutually exclusive, it is possible for event A and B to occur at the same time. This means that they share some of the elements from the sample space. We therefore have

$$P(A \cup B) = P(A) + P(B) - P(A \cap B)$$

Since the elements that event A and B share are both found in $P(A)$ and $P(B)$ we have $P(A \cap B)$ twice and therefore must withdraw one of them.

4.1.5 Independence

Event A and B are said to be independent if the occurrence of one of them does not depend on the occurrence of the other. To illustrate this lets first look at drawing the balls in the Lotto game [35].

Here every ball drawn slightly effects the possibility of a given ball being drawn the next time since the sample space has been decreased by one ball.

However, in the Joker part of the Lotto game this is not the case. Here the numbers are drawn from 7 different containers, each containing 7 balls, and therefore, drawing a number 7 from the first container does not depend on or effect the probability of drawing a number 7 from one of the other containers. The event of drawing a certain ball from one container is therefore independent of the event of drawing a certain ball from one of the other containers.

The probability of the two independent events A and B both occurring is written as:

$$P(A \cap B) = P(A)P(B)$$

4.1.6 Conditional probability

When the two events are not independent we have a situation like the drawing of balls in the Lotto game. That is, knowing event B has occurred changes the possibility of event A also occurring.

We write the probability of event A occurring given that we know event B has occurred as

$$P(A|B)$$

The probability of both event A and B occurring is

$$P(A \cap B) = P(A)P(B|A) = P(B)P(A|B)$$

Using this rule we can determine the probability of A occurring when we know that B has occurred as:

$$P(A|B)= P(A \cap B) / P(B) = P(A)P(B|A) / P(B)$$

This rule is called **Bayes' theorem** and lets us find the probability of P(A|B) when we know the prior probability P(A), the likelihood P(B|A) and the evidence P(B). The process of finding P(A|B) is called Bayesian inference and is a very central aspect of Bayesian networks.

4.1.7 Conditional independence

Now that we know about conditional probability we can find a way to check whether two events A and B are in fact independent. We know that they are independent if either

P(A) = 0 or P(B) = 0 or

P(A|B) = P(A) and P(A)!=0 or P(B)!=0

In fact we can also show whether two events are conditional independent. That is, two events A and B might not be independent, but when introducing a third event C, knowing that C has occurred makes A and B independent.

For example, if we have a game where a tank can only be build if we have a factory, and a factory can only be present if our main base has been upgraded to command center. Here the event A of building a tank is not independent of the event B of having a command center. However, if we also have the event C which means that we do in fact have a factory then A and B becomes independent. Knowing that we have a command center does not make it more or less possible that we are building a tank when we also know that we do have a factory. We have conditional independence when either

P(A|B∩C) = P(A|C) and P(A)!=0 or P(B)!=0

or

P(A|C) = 0 or P(B|C) = 0

4.1.8 Random variables

A random variable is a function that maps events to numbers [23]. This number does not have to lie between 0 and 1 as is the case for probability functions. Instead the number is a representation of the outcome of the event. For example, the event of drawing a card from a deck of cards can have the outcomes {heart, diamond, spades and club}. Here a random variable could be made such that if the outcome of the event was heart the random variable would be 1, if the outcome was diamond the variable would be 2 and so on.

A way to explain the relationship between random variables and events is to look at the inverse random variable function. If we look at the event of getting an even number when throwing a dice, a random variable could be made such that if an even number was thrown the random variable would be 1 and otherwise 0. If we inverse this random variable function and feed it with the number 1 this would give us the subset of the sample space containing the events {2, 4, 6}.

The reason to why we would prefer the notion of random variables instead of the notion of events is that random variables in some situations seem more intuitive. For example if we consider the event X = “number of heads when tossing a coin 20 times in a row”. In this example random variables let us specify the probability of getting heads at least 10 time by writing $P(X>10)$. This cannot directly be done with events where we instead must define a completely new event for getting heads at least 10 times.

When dealing with a specific problem domain random variables are used to represent some feature of an entity in our domain and we are interested in knowing which state these random variables are in [2]. In our RTS game for example a player may or may not have a factory. Here the random variable HaveFactory can be in either the state true or the state false.

When we define our random variables it is important that they are precise enough to pass the clarity test [3], which is a test used for removing any ambiguity from the possible states of a random variable. In the test we imagine that there is a clairvoyant who knows everything about the current state of the world. This clairvoyant must then be able to unequivocally determine the current state of our random variables. So if we have a random variable ArmySize which can take the values small, medium or large, our variable won’t pass the test since we won’t know exactly which army sizes belong to which of the states. Do 300 soldiers belong to medium or large for example?

If instead the values were BiggerThanEnemyArmy, SmallerThanEnemyArmy or equalToEnemyArmy the clairvoyant would indeed be able to unequivocally tell which state our variable is currently in.

4.1.9 Joint probability distribution

The joint probability distribution is the probability that all the variables in the problem domain have a particular value at a given time. If the variables are independent we write this as

$$P(X=x, Y=y) = P(X=x)P(Y=y) = P(x)P(y)$$

and if they are not independent we write it as

$$P(X=x, Y=y) = P(x|y)P(y) = P(y|x)P(x)$$

where X and Y are two random variables and x and y are specific values for X and Y.

The full joint probability distribution is all combinations of variables in a problem domain. So for each variable X we would need to multiply the probability for each of the states X can take with probabilities of all possible states of the other variables in the joint probability distribution. Therefore we would need to do m^n calculations where m is the number of states in the variables and n is the number of variables.

4.1.10 Marginal distribution

A way to calculate the probability of a random variable from a joint probability distribution taking a certain value is by summing up all values of the other variables [2].

As an example, consider again the joint probability distribution of the two variables X and Y. If we let x and y (the set of values for X and Y) be {true, false} and {true, false}, we can use the marginal distribution of X to find the probability of X being true:

$$P(X = true) = \sum_{y} P(X = true, Y = y) = P(X = true, Y = true) + P(X = true, Y = false)$$

Even though this might look like an easy way to find the probability of a variable being in a certain state, there is a problem with the method, namely that the number of values, which we need to add together to find the marginal distribution of X, increase exponential with the number of variables in our joint probability distribution.

In our little example, there were only the two variables X and Y, so to find the marginal distribution of X we only had to go through the two values of Y. But if there had also been the variables A, B, C, D and E we would have needed the sum of the 2^6 possible values of A, B, C, D, E and Y.

Also our variables only had two possible values each. This is not always the case and therefore the number of values we need to add together might increase even further.

There are other methods for finding the probability of a variable being in a certain state, but before these are described it is first descrined what a Bayesian Network is and how we can use a Bayesian Network to find a joint probability distribution.

4.2 Bayesian Network

A Bayesian Network is a directed acyclic graph (DAG) where the nodes represent random variables and the edges between the nodes represent relationships between the random variables. Directed means that the edges must point from one node (the parent) to another node (the child), and an edge between two nodes means that the state of the parent node affects the state of the child node. Or, said in another way: a node X_i has a conditional probability distribution $P(X_i|Parent(X_i))$ that quantifies the effect of the parents on the node [31].

One thing that can be learned from a Bayesian Network is the dependencies among the variables in the problem domain. Another thing is that when there is no edge between two nodes in a Bayesian Network this represents independence between these two nodes; nothing about the state of one variable can be inferred by the state of the other [4].

A node can also be conditional independent of other nodes in the network. In fact a node is conditional independent of all other nodes in the network, given its parents, children and children's parents [31].

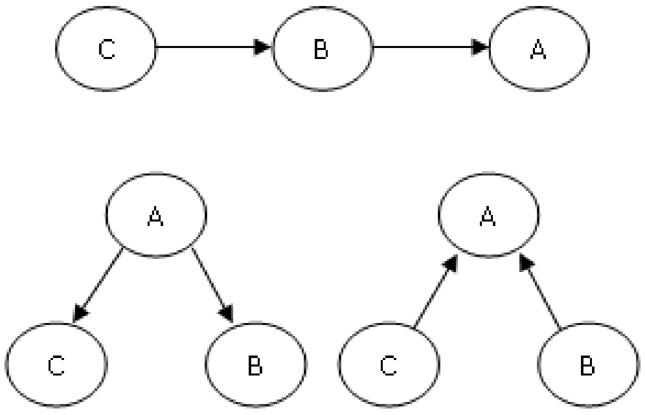


Figure 4-1 Bayesian Networks

The upper network in Figure 4-1 is an example of conditional independence, since nothing about C can change A given we know B. In the bottom left Bayesian Network C is conditional independent of B given A, but in the bottom right Bayesian Network C is not conditional independent of B given A since B is neither a child, a parent og parent of a child of C.

So one advantage of using a Bayesian Network to model our problem domain is that we get a better understanding of what random variables the problem domain contains and also what the relationship between these variables is, which can aid us when doing bayesian inference. It is also easy to see how modifications of the network affect the relationships among the variables in the network. However, this can also be seen as a problem since we always need to consider the whole problem and design the entire network before we can get this understanding.

Another advantage is that the conditional dependency and conditional independence that exist in a network make it possible to write the joint probability distribution P as the conditional distribution of all nodes given values of their parents. For example, the bottom left network in Figure 4-1 can be written as

$$P(a, b, c) = P(c|a)P(b|a)P(a)$$

If we compare this way of stating the joint probability distribution with specifying it directly, we see that the method using conditional tables requires fewer calculations than stating it directly. For example, finding the joint probability distribution for the bottom left graph directly would require

$$P(c, a, b) = n * n * n = n3 \text{ calculations}$$

Whereas finding it using conditional dependencies requires

$$P(c|a)P(b|a)P(a) = n*n + n*n + n = 2n^2+n \text{ calculations}$$

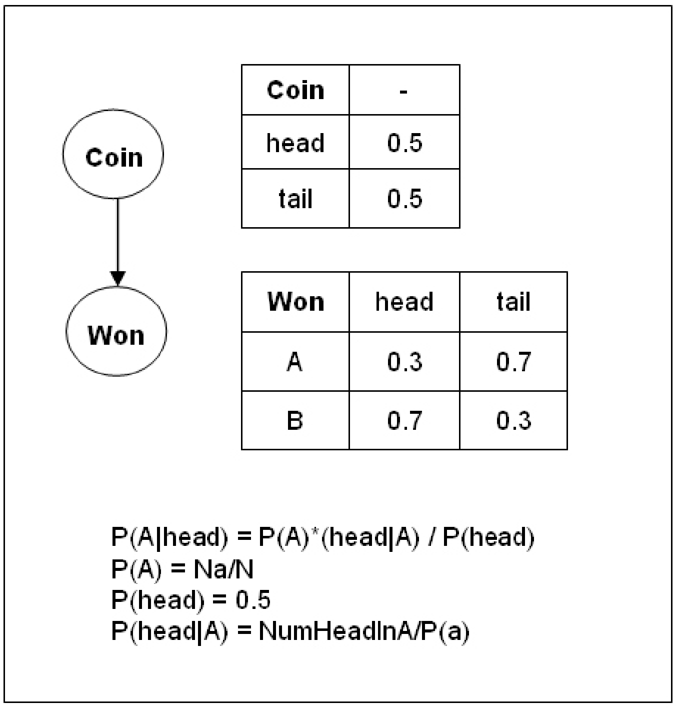
Besides of giving a good understanding of the relationships between the variables in the problem domain we use Bayesian Networks to find the state of query variables given the state of some evidence variables. For example we might know whether our opponent in a RTS game currently has ground units or air units in his army, and from this information we can find the probability of our opponent attacking via the air or ground. This could be illustrated by the lower right network in Figure 4-1 were the variable C could show whether he has ground units in his army, B could show whether he has air units and A – the query variable - could show the probability of the opponent preparing an air or a ground attack given the state of the evidence variables B and C.

The following sections will first describe how the conditional tables belonging to the nodes in the network can be given values and hereafter, methods to determine the state of some query variables are described

4.2.1 Setting the tables

Before we can retrieve information from a Bayesian Network, the table relating to the nodes in the graph of course needs to be given some values. There are two ways of doing this. One way is to do measurements of the problem domain being modelled and then to use the collected data to set the values. This also allows for the inference methods mentioned earlier to construct tables which we cannot directly measure from the problem domain.

However in some situations these data might not be available. When this is the case we cannot use prior probabilities and inference rules to build up the tables in our network. The following example clarifies this:


Coin
Won
P(A|head) = P(A)*(head|A) / P(head)
P(A) = Na/N
P(head) = 0.5
P(head|A) = NumHeadInA/P(a)

Coin	-
head	0.5
tail	0.5

Won	head	tail
A	0.3	0.7
B	0.7	0.3

Figure 4-2 Inference

In this example (Figure 4-2) two persons are betting on the outcome of flipping a coin and we imagine that we only know the probability of the coin landing on head or tail. We don't know anything about the number of games played, how many times the players play on head and tail or about how many times each player has won the bet.

In the example we are looking for the distributions of the table P(Won|Coin). However, to find these distributions we would need the information about previous bets which we do not have. For example we would need information about the number of games played and number of games won by player A to find P(A) and so on. Since we cannot calculate the distribution of Won given Coin we must set these values manually.

This situation is the case for this thesis, since no data is available from which the tables could have been constructed.

For example there is no data which can tell us the probability of building a tank in a given situation since there is no way that we can get this data by doing some measurement. Had the situation been that we were trying to model the way a human player plays, we could have measured how often he was building a tank compared to building other units, which could then have been used for setting the probability for building a tank in our table.

4.2.2 Making queries in Bayesian networks

Once all the tables have been set, either based on measuring or by setting the values manually, another question arises: how do we actually find the probability we are looking for? For instance, in the previous example: how do we get the probability of A or B winning? All we know is that *if* Coin takes the value of head then the probability of A winning is 0.4, otherwise it is 0.7. However, we don't know the state of Coin.

The following section will go through different methods that can be used for this.

The joint probability distribution

One way of finding the probability for a given event is by using the joint probability distribution. The joint probability is, as mentioned previously, the probabilities of all combinations of values of the random variables in the network. So if we know the distribution for the states of Won given all combinations of possible values for the parents of Won (Coin) we would have the joint probability distribution for our network and thereby also the probability of A winning and of B winning. The following example illustrates this.

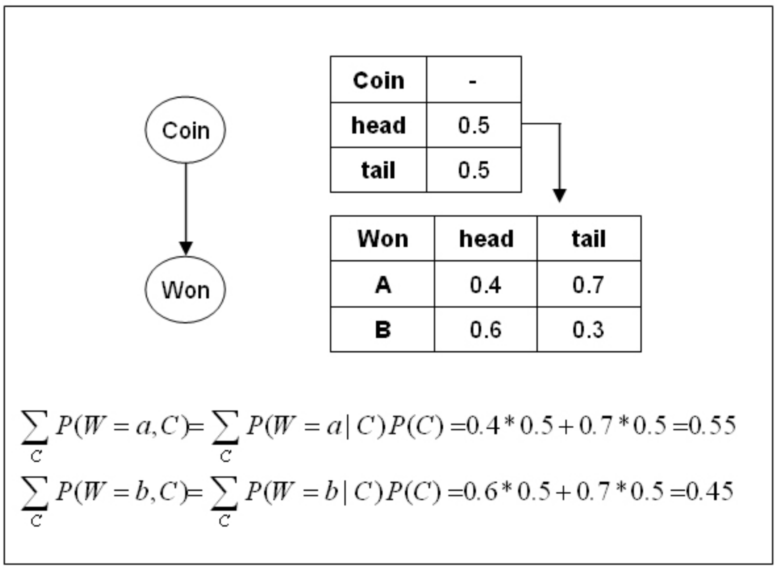
> Lets again say that we have the following distributions for our tables:
>
> P(Coin=head) = 0.5
> P(Coin=tails) = 0.5
>
> P(Won=A|Coin=head) = 0.4   P(Won=B|Coin=head) = 0.6
> P(Won=A|Coin=tails) = 0.7   P(Won=B|Coin=tails) = 0.3
>
> The joint probability distribution is then the following
>
> | | | | | |
> |---|---|---|---|---|
> | P(head, A) | = | P(A\|head)*P(head) | = | 0.20 |
> | P(head, B) | = | P(B\|head)*P(head) | = | 0.30 |
> | P(tails, A) | = | P(A\|tails)*P(tails) | = | 0.35 |
> | P(tails, B) | = | P(B\|tails)*P(tails) | = | 0.15 |
>
> From this we see that
>
> P(Won=A) = 0.20 + 0.35 = 0.55
> P(Won=B) = 0.30 + 0.15 = 0.45

If we were to bet on whom of them would actually win the next bet, we could make a random choice based on these distributions. The problem with using the joint probability distribution is that as the number of nodes in the network increases, the number of entries in the joint probability distribution increases exponential and therefore quickly becomes very large. Since this often makes it impractical to use the joint probability distribution to find the probability of a given event, or to find the probability for all possible events, other methods will be described next.

Exact inference

Exact inference means that once we have found the probability distribution we are looking for, this distribution will be the true probability distribution.

A method for doing exact inference is to send the entire table from a parent to a child so that we in the end will have the probability distributions for all possible events in the network. This is illustrated in Figure 4-3.



Coin	-
head	0.5
tail	0.5

Won	head	tail
A	0.4	0.7
B	0.6	0.3

$$\sum_{C} P(W=a,C) = \sum_{C} P(W=a \mid C)P(C) = 0.4*0.5+0.7*0.5 = 0.55$$

$$\sum_{C} P(W=b,C) = \sum_{C} P(W=b \mid C)P(C) = 0.6*0.5+0.7*0.5 = 0.45$$

Figure 4-3 Exact Inference

The important thing in this example is that we do not know the state of Coin and therefore have to send the entire Coin table down to the Won node. If we had known the state of Coin it would be easy to figure out the probability of A or B winning.

There is one disadvantage with this method, namely that in larger networks it becomes almost impossible to send the tables from a node to another and also to do the calculations at each node, just as the case was with the joint probability distribution. If we imagine that Won also had another parent SpinBottle which had 4 states/attributes, then the probability of A given Coin and SpinBottle would be calculated by summing 2^4 numbers. And since a table often has lots of parents which have lots of states this could turn into a very large calculation.

Therefore methods that only approximate the true probability distributions are often used instead of the exact inference method. In the following section 3 approximation methods are described.

Approximation methods

When we cannot use exact inference methods, we can use approximation methods instead. These methods let us learn realistic models from large data sets and in general trade off computation time for accuracy [21]. This means that the more samples produced, the more accurate the result will be.

Getting an answer – the one shot method

If the distributions for all the tables in the networks are available, the simplest way to get an answer to a query is to make one sample only. This is done by making a random choice at each variable based on the probability distribution for the variable, to find out which state the variable should take.

If we return to the Coin-Won example, let's say we have the following distributions:

P(Coin=head) = 0.5

P(Coin=tails) = 0.5

P(Won=A|Coin=head) = 0.4   P(Won=B|Coin=head) = 0.6

P(Won=A|Coin=tails) = 0.7   P(Won=B|Coin=tails) = 0.3

To find out whether A or B should win, we do the following:

P(Coin) = [0.5, 0.5] - make random choice – let's say it takes the state head

P(Won|Coin=head) = [0.4, 0.6] - make random choice – let's say that it takes the state A

The winner is A.

The question with this method is whether it actually is useful. Does it reflect the true distribution from the network? It could seem that when we only do one sample this is not the case, since the probability of getting A as the winner is 1.0 after this sample.

However, if we did a lot of samples, the probability of getting A as the winner in fact does not change. Let's say that we did 100 samples. The result of this would be approximately the following:

Number of Coin=head : 50

Number of Won=A given Coin=head : 20

Number of Won=B given Coin=head : 30

Number of Coin=head : 50

Number of Won=A given Coin=head : 35

Number of Won=B given Coin=head : 15

This means that the chance of getting A as the winner when the coin lands on head is 20%. But this is in fact the same as the probability of getting A by doing just one sample, namely 0.5 * 0.4 = 0.20.

Of course this one-shot method only works when we are interested in getting a single answer from the network and accepts the probability distribution specified for this answer. If we instead are interested in finding the actual distribution for a given event, for instance finding P(Coind=head|Won=A), the one-shot method will be to imprecise to be usefull. Instead we will need to do more than just one sample if we want an accurate result.

The following describes different methods for doing this.

Rejection sampling in Bayesian networks

When we are interested in distributions that are hard to sample directly we can use rejection sampling which is a simple way of producing these samples.

When the algorithm is run a sufficient amount of times it will produce a consistent estimate of the true probability [31].

The algorithm takes as input a Bayesian Network, some evidence variables and a query variable and works as follows:

- Produce a random event from the network.
    - If the event does not match the evidence variables rejected it.
    - Else determine if the query variable has the wanted value. If this is the case increase the total amount of times this event occurred by one.
- Run the previous steps until we have enough samples.
- Calculate the conditional distribution by dividing the number of times our wanted event occurred with the total amount of samples.

This method is a lot like sampling some event from the real world, for instance measuring how many nights in April the moon is visible.

A problem is that this method rejects a lot of samples. And the more evidence variables we send into the algorithm, the more samples will be rejected by it.

Likelihood weighting

Rejection sampling has the problem of throwing away those of the produced samples that were not consistent with the event of interest. A method that does not have this problem is the likelihood weighted method. In this method, the values for the evidence variables are fixed and therefore each event generated will be consistent with the evidence [31]. However, though all events and therefore all samples will be taken into account, they will not be equally important. When we have enough samples from the network and are about to determine the actual distributions, each sample is weighted depending on the evidence variables and based only on these weighted samples are the distributions determined. The algorithm works the following way:

- For each node in the network determine its probability conditioned on its parents.
    - If the node is an evidence node multiply the probability for the value of this evidence variable to a variable W, which initially is set to 1.
    - Else make a random choice based on the probability for the variable given its parents.
- Add the value of W for the generated event to the already existing W value for this type of event.
- When enough samples have been produced normalize the W values so they add up to one. This will be the probability distribution we are looking for.

Markov chain simulation

The rejection sampling method and the likelihood weighting method both generated each event from scratch. The Markov chain simulation method is different in that it at each step only makes random changes to the preceding event instead of always producing new events.

In Markov's chain simulation some evidence variables stay fixed. Whenever a sample is to be produced, one of the non-evidence variables is randomly changed and is done so conditioned on the values of the variables in the Markov blanket of X_i [31], which means the values of the variables of the parents, children and children's parents of X_i.

The algorithm therefore works as follows:

- For each variable not in the evidence
    - Change a non-evidence variable conditioned on its Markov blanket,
    - Increase the number of occurrences for the resulting event with one.

When we have enough samples, normalize the number of occurrences for each event so that they add up to one. This will be the probability distribution we are looking for.

4.2.3 Conclusion on Bayesian Networks

Section 4.2 discussed Bayesian Networks and methods for getting an answer from a Bayesian Network. That is, how does the resource management AI actually decide what to build once the Bayesian Network has been constructed?

The method chosen for this thesis is the One-Shot method. The reason for this choice is that the resource management AI simply needs a method that will inform it which unit or building to build. It does not need the probability distribution of all possible units and buildings. This would of course also have been usefull for the AI. Once the probability distribution had been received it could choose which unit to build based on the probability distribution.

The One-Shot method on the other hand simply returns an answer on which unit or building to build. It does this by doing one sample of the network only and returns this answer. This means that the true probability distribution cannot be found by using this method. However, if the true probability distribution had shown a probability of 0.2 for building a given unit, the chance for getting that unit type as an answer from the One-Shot method will also be 20

%. Therefore, the One-Shot method is a lot faster than finding the true probability distribution and is still suitable for this thesis.

4.3 Reinforcement Learning

In reinforcement learning an agent learns the best actions for a given situation by trying different actions and being rewarded for those leading to the best result. This section will describe the most important parts of reinforcement learning.

States, actions and the reward signal

In reinforcement learning the environment can be in a number of different states and in each of these states, a number of actions, which all lead to other states, are available for an agent.

When an action has been executed, the agent is informed whether the action leads to something positive or negative. This information is given to the agent as a real number representing a reward where actions leading towards success result in large rewards and actions leading towards failure result in small rewards The goal of the agent is to learn which actions to take from an initial state and until a terminal state is reached, for instance the end of the game, in order to get the highest possible accumulated reward. The accumulated sum of rewards is also called the return.

In some situations, the agent will not get an immediate feedback on the action chosen in a given state but will instead receive delayed reward. For instance, an environment could consist of states which all gave a reward of zero except for a terminal state which gave either a reward of one or minus one. In this situation the agent will not know whether a chosen action leads to something positive or negative until a terminal state has been reached and therefore the agent needs to keep track of which actions were chosen in which states so that it can learn the state-action sequence that leads to success.

Policy

As said the aim of an agent in Reinforcement Learning is to find the action-state sequence that leads to the highest return. A policy expresses which actions should be taken in each state and is denoted with the symbol π. That is, a policy is a set of actions $\pi = \{a_1, a_2, \dots, a_n\}$ that an agent can choose to perform. The goal of the agent therefore is to find the policy leading to the highest return, also called the optimal policy.

The state value and the value function

The value of a state expresses the expected return from being in that state and following the policy π afterwards. The value of a state therefore is an indication of how attractive it is to be in that state.

The value function is a mapping from states to state values and shows the expected return when starting in state s and following π thereafter [2].

$$V^{\pi}(s)$$

To illustrate the difference between the reward an agent can get from a state and the value a state can take, let's look at the problem of finding the shortest path through a graph, where each edge in the graph has some length assigned to it.

We let an agent try to find the shortest path through the graph. The way the agent does this is by always taking the edge with the shortest length assigned to it from a node, since this will give the highest immediate reward. Once the terminal node has been reached the agent will know one path through the graph. And, for what it knows, this path will be the shortest path since the shortest possible edge was always chosen.

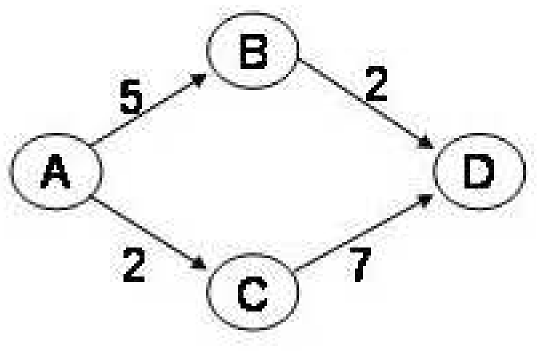


Figure 4-4 Finding the shortest path

However, shorter paths through the graph might actually exist. If instead each node had a value stating the shortest possible path from that node on and until the terminal node, the agent could choose an edge(action) leading to a node with higher state value though this edge might give a low immediate reward.

The action values

Where the state value showed the expected return when being in state s and following π thereafter, the action value, or the Q-value, shows the expected return when taking the action a in state s and following π thereafter. We write this as

$$Q^{\pi}(s, a)$$

This means that we have an estimate of how high the return will be when choosing a in s and hereafter choosing the action a' in states s' (the state that followed by taking a in s) as specified in the policy π.

So where the state value shows how attractive it is to go into a specific state the action value shows how attractive it is to perform a specific action when already being in a specific state.

Action selection policies

Based on the action values, we can choose which action to take in a given state. A way to do this would be to always choose the action which gave the highest estimate for future rewards. However, as we saw in the shortest path example, this might not result in the shortest path. In the following two different action selection policies are described which both guarantee at some point to find the highest total reward.

ε-greedy

This method takes the action with the highest estimated reward almost all the time. However, in each state there will be a small probability of ε for choosing to explore instead of exploiting. That is, it will choose randomly between the other possible actions in that state, independently of the action-value estimates for these other actions [23]. This ensures that if it has run enough times, eventually all possible actions will be tried and therefore the optimal actions will be discovered. Adjusting the ε value will determine how fast all the possible actions have been tried.

Softmax

A problem with the ε-greedy method is that when it chooses to explore, it makes a random choice between the remaining possible actions from the state, without taking into account the estimated rewards to be received from each of the remaining actions. In the softmax method, a weight is assigned to each of the actions according to their action-value estimates [22]. When the agent explores instead of exploiting, a choice is made between the remaining actions taking into account the weights assigned to each action. This means that the actions with the lowest action-value estimate are very unlikely to be chosen and the method is therefore useful when some actions are known to be very unfavourable [22].

4.3.1 Learning algorithms

The actual learning in Reinforcement Learning systems is about approximating the value function. That is, finding the true value for each state. When these have been found, the optimal policy can be determined. In the following different algorithms for finding the state values are described.

Value iteration

In value iteration it is assumed that all states and their corresponding state values can be represented for instance as a lookup table. If this is the case, the algorithm determines the true state value of all states by examining the return from all actions in all states. The state value is then set to be the highest return value possible from the state. When this process no longer results in updates to state values the algorithm is terminated.

For each action in each state the following calculation is performed

$$\Delta w_t = \max_u(r(x_t, u) + \gamma V(x_{t+1}))-V(x_t)$$

- where x_t is the current state, u is the action taken in that state leading to a state transition to state $x_{t+1}$and $r(x_t, u)$ is the reinforcement received when performing u in x [25]. The symbol γ is a discount factor between 0 and 1 and determines how much future rewards should count compared to immediate rewards.

So for each action in a state it is examined what the immediate reward is and what the value of the resulting state is (taking into account the discount factor) from taking this action. From this information the value of the state is updated.

The reason to why the value iteration method can only be used when all states and values can be represented as for example a lookup table is that the algorithm needs to go through all possible states to be able to update the state values. This means, that an agent learning to control a plane or a car would need to go to all possible states even though some of them would result in an accidence occurring. When this is the case, the agent cannot go back to the initial state and examine other actions since the vehicle now is destroyed. Instead the agent must in simulation perform the actions and observe the result [25].

Monte Carlo

In the value iteration method a model of the problem was needed to be able to learn the value of a state. That is, the possible actions of a state had to be known, the rewards from performing these actions had to be known and information about the states transitioned to when performing these actions had to be known. However, in some situations such information might not be available. If this is the case, the state values can instead be learned by collecting statistics about the rewards received when following the policy π a given number of times. This type of method works as follow:

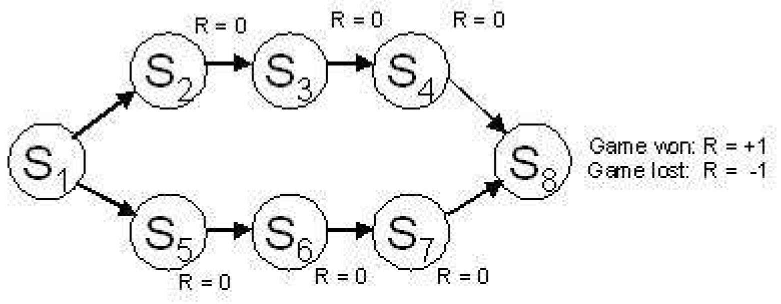


Figure 4-5 Example for Monte Carlo

The figure above (Figure 4-5) illustrates a small game where a player can be in 8 different states. After each iteration or round of playing the game, it is remembered which states the player was in during the game and whether the player in the end won or lost the game. As can be seen, the only state which will inform the player whether he is performing well or bad is the terminal state and therefore nothing can be concluded in the game before it is has been played to the end.

As an example of how to calculate the value function, let's say that the player in 10 rounds of the game was in state S_3 and of those 10 rounds the player lost the 8. The state value of S_3 would therefore be

$$V^{\pi}(S_3) = (2-8) / 10 = -0.6$$

From this the player will know that state S_3 should be avoided.

Besides of being different from the value iteration method by being able to calculate the value functions without knowing the model of the problem, there is one other big difference between these methods. The value iteration method calculates the value functions for the complete state space whereas the Monte Carlo method only calculates the state values of the states visited during a game. This is an advantage if the state space is very large, which often is the case in games because of the combinatoric explosion of the number of possible states [25]. However, this also leads to one disadvantage namely that the optimal policy might not be found which is guaranteed in the value iteration method.

Temporal difference learning

Another method for learning the value function is the temporal difference learning method. In this method the value of the present state is expressed by the immediate reinforcement and the value of the next state [27]. So for iteration $I+1$ we update the state value of S_t by the following calculation

$$V_{I+1}(S_t) = V_I(S_t) + \alpha(r_{t+1} + \gamma V_I(S_{t+1}) - V_I(S_t))$$

Where

- $V_I(s_t)$ is the value of the state s_t at iteration I
- $V_I(s_{t+1})$ is the value of the state s_{t+1}
- r_{t+1} is the reinforcement reward received from going to state s_{t+1}
- α is a learning rate
- γ is the discount factor.

The value $V_{I+1}(S_t)$ will only be changed if it differs from the right side of the equation and it can only be calculated when the value of the next state and the reward received from going to that state is known.

This method might seem a lot like the value iteration method described previously. However, where that method calculates the value functions for the complete state space the temporal difference method, just like the Monte Carlo method, only updates the states visited.

The Q-learning and the Sarsa algorithms

In the temporal difference method there is no policy for choosing which actions to perform. The Q-Learning method and the Sarsa method are two methods that both use the idea of the temporal difference method, but which instead learn the optimal policy by updating the action-value estimates instead of the state value.

The Q-learning algorithm works as follows:

- First choose an action from the state s (for instance by using softmax or greedy).
- Observe the reward r and the state s' that follows by taking action a in s.

- Update the action-value estimate by adding to it r plus the difference between the maximum action-value estimate from s′ and the current value of Q(s, a), taking into account the learning rate and the discount factor.
- Set the current state to s′ and start from the beginning of the algorithm until a terminal state has been reached.

The actual update of the action-value for the selected action can be written as

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \mathrm{Max}\ Q(s', a') - Q(s, a)]$$

Where s′ is the state transitioned to when taking the action a in s and a′ is the possible actions in s.

The action-selection policy used for choosing which action to perform could be for example the previously described soft-max or greedy methods and the Q-learning algorithm is guaranteed to eventually learn the optimal policy. This is also the case when actions are selected according to a more exploratory or even random policy [28].

The Sarsa algorithm is almost identical to the Q-learning algorithm. The only difference is that the Q-Learning algorithm always chooses the action with the highest action-value estimate in the state s', whilst the Sarsa algorithm chooses this action based on the same action-selection policy used to select an action in the state s (for example softmax, e-greedy).

4.3.2 Conclusion on reinforcement learning

This section has described some concepts of reinforcement learning including different methods for learning the state values and methods for learning the action-value estimates.

Which learning algorithm to choose for this thesis depends on what actually needs to be learnt and also how the environment is going to be represented. Therefore, only general conclusions can be at this point

One of them is the fact that games tend to have a huge amount of states which makes the method Value Iteration less attractive since it calculates the value functions for the complete state space.

Another point is that many of the algorithms described need specific actions taken in a state to result in state transition to another specific state. Such a model would have been available if the game had been a board game or a turn based game. However, for this project the game is a real-time game, meaning that the game environment is very dynamic. Therefore, an action taken in the current state might not always lead to the same specific state but will instead lead to different states depending on what happens in the game at that time. This makes it difficult to use the methods Q-learning and Sarsa since they expect to know the reward received when performing an action leading to a specific state.

Therefore the Monte Carlo method seems to be the best choice for this project, but how this method will be designed and what the method is to learn will be discussed later.

4.4 Conclusion on theory

This chapter has described the theory of probaility, of Bayesian Networks and of Reinforcement Learning. This has given the foundation needed for later developing the resource mangement AI.

Furthermore it has been chosen to use the One-Shot method for getting answers from the Bayesian Network and to at least focus on the Monte Carlo method for training the network (AI).

5 Game Design

In the analysis part of this project it was chosen to build a small RTS game from scratch into which the AI could be integrated and tested. This chapter explains what this game actually is going to be about, what types of units and buildings there will be in the game and how the interface of the game is going to look like.

Game play

The overall mission of the game is to destroy the enemy main base. When a player looses his main base the game is over and must be started again.

In the game, the player can first of all build workers who will gather resources. These resources can be spent on building either military units or buildings. The more income the more units and buildings can be build.

The different military units in the game will each have some advantages and some disadvantages and the player must therefore choose wisely when deciding which unit to build.

When the game starts the player will have one main base and four workers at his disposal. To be able to build military units, the player must build special buildings from where the military units can be constructed. It takes time to build both units and buildings. However, the more buildings you have of a particular type, the faster the construction of the military units produced from that building will be.

The units

When designing an RTS game it is important that the units in the game differ from each other in functionality and that the units' capabilities are well balanced.

If the only difference between unit A and B is that unit B costs a little more, is a bit harder to kill and has a bit more firepower than unit A, then the choice of whether to build A or B becomes somewhat irrelevant since the power of your army will be exactly the same whether you build unit A or B. At the same time it is important, that a unit is not made that powerful, that the players always choose to build that type of unit and never considers building other types of units. To avoid this Harvey Smith [34] suggests what he calls an orthogonal unit differentiation which basically means that all choices / unit types must have some advantages and some disadvantages (like the stone, scissor, paper game). Following this principle will encourage (intentional) strategic play and expand the games possibility space [34].

To achieve this, each unit will have six characteristics which each will differ from unit to unit. These characteristics are price, build time, health, firepower, weapon type and amortype. The weapon type and amortize are important to remember. The armor type will give the attacked unit a defence bonus, which can either be small or large depending on the weapon type. For example a light armor will give the attacked unit a large defence bonus if the attacker uses a machine gun while a bazooka only gives a small defence bonus to a light armor.

Table 5-1 shows the characteristics of the different units available in the game.

Table 5-1 Units characterists

Name	**Price**	**Build time**	**Health**	**Firepower**	**Weapon type**	**Amortize**	**Build From**
Worker	50	15 sec	-	-	-	-	Main base
Lightsoldier	60	10 sec	150	30	Machine gun	Light	Barrack
Heavysoldier	70	12 sec	150	45	Bazooka	Light	Barrack
Lighttank	80	15 sec	300	55	Heavy Machine gun	Medium	Factory
Heavytank	100	20 sec	400	60	Grenades	Heavy	Factory

The buildings

Before the player can build military units in the game, he will need buildings from where these units can be constructed. Buildings can be constructed whenever the player has enough money, but can also be destroyed again by the enemy forces. The following table shows the characteristics of the buildings in the game (see Table 5-2).

Table 5-2 Buildings charactersitics

Name	**Price**	**Build time**	**Health**	**Firepower**	**Weapon type**	**Amortize**	**Builds**
Barrack	150	30 sec	1000	-	-	Heavy	Soldiers
Factory	150	30 sec	1300	-	-	Heavy	Tanks

The defence bonus

Table 5-3 shows the different defence bonuses for the different types of armor. The defence bonus is a number smaller than 1 which is multiplied with the total firepower for the attacking unit type.

Table 5-3 Defence bonus for differente armor types

Amortize **Weapon type**	Light armor	Medium Armor	Heavy armor
Machine Gun	0.70	0.50	0.15
Heavy Machine Gun	0.90	0.80	0.50
Bazooka	0.35	0.90	0.70
Grenades	0.20	0.70	0.90

The interface

In the game, each player will have various buttons for creating units and buildings and text areas displaying the amount of units and the unit health (Figure 5-1).

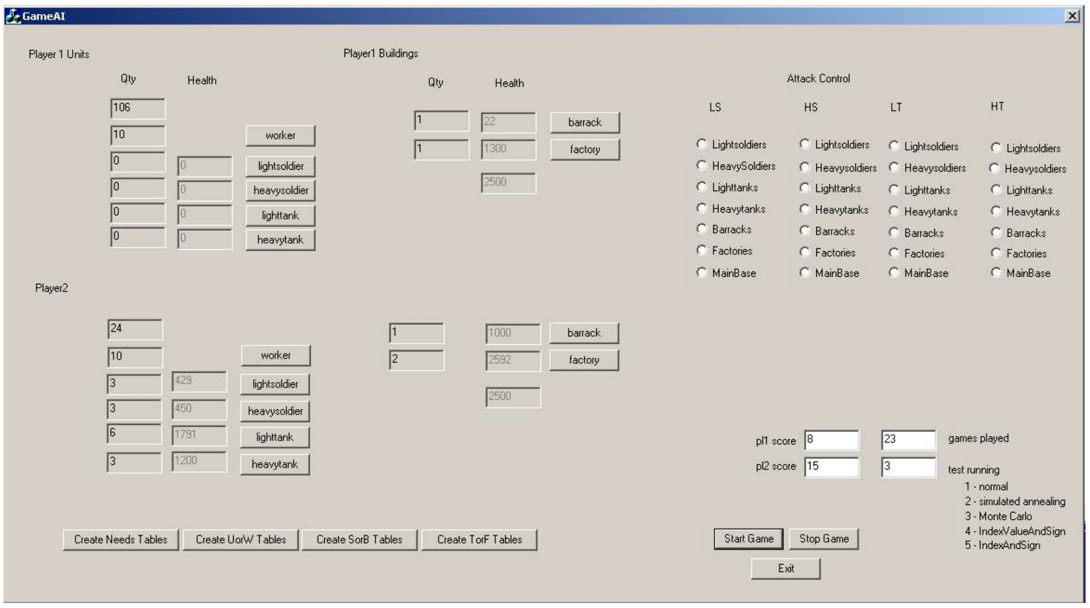
For example, a player has a button for constructing new lightsoldiers. When pressing this button – provided the player has enough money – the price for a lightsoldier will be withdrawn from the players account, and the construction of the unit begins. When the construction is complete, the unit is added to the player's army.

Next to a build button is a text field displaying the amount of a particular unit and next to it is another text field displaying the total health of this unit class. That is, the health of all units of a particular type is shown in this text field. This also counts for building information.

There are two different ways of playing the game. The player can either order his units to attack specific enemy units (or groups of units) or buildings. This is done via the radio buttons in the top-right part of the screen by selecting a given unit type as the target for one of your unit types.

Or the selection of targets is done automatically by the computer by following some rules laid for each unit type. These rules make sure, that a unit type will attack that unit type on which they will deal most damage and avoids, that for example a lightsoldier attacks a building on which he will cause next to no damage.The automatic selection of targets will be used when two AIs are playing against each other.

At the bottom of the screen are buttons for creating random tables for the network (7.5), to start and stop a game and text fields displaying the current score, the total number of games played and also which test or experiment (8.2) is currently running. These informations and functionalities are primarily used when performing experiments.



	Qty	Health
Player 1 Units		
	106	
worker	10	
lightsoldier	0	0
heavysoldier	0	0
lighttank	0	0
heavytank	0	0
Player1 Buildings		
barrack	1	22
factory	1	1300
		2500
Player2		
	24	
worker	10	
lightsoldier	3	429
heavysoldier	3	450
lighttank	6	1791
heavytank	3	1200
barrack	1	1000
factory	2	2592
		2500

pl1 score 8
pl2 score 15
23 games played
3 test running
1 - normal
2 - simulated annealing
3 - Monte Carlo
4 - IndexValueAndSign
5 - IndexAndSign

Figure 5-1 The game interface

6 Game implementation design

This chapter shortly describes the classes needed for implementing the RTS game and also describes the communication between these classes. The game class diagram is shown in Figure 6-1

Player

The player class is responsible for storing all the information concerning a player. That is the number of workers, lightsoldiers, heavysoldiers, lighttanks, heavytanks, barracks and factories belonging to the player. This also includes the amount of money that the player is in possession of. The player class is also responsible for changing these informations throughout the game.

Units and buildings

An obvious way to design the units and building classes would be to have one super unit class and one super building class and then to let the individual unit and building types be generalizations of theses classes. However, since all units share exactly the same characteristic; build time, price, health, firepower, weapon type and amortize and all buildings share exactly the same characteristics; price, build time, health, amortize, the design wont gain any benefits by doing these generalizations – it will only lead to redundant code.

Instead there will only be only one unit class and one building class. To distinguish between the different types of units, the player class will have a container for each type. This will give a fast access to information of the different units and buildings possessed by the player.

TheFactory

TheFactory is a factory-pattern [24] and is available for all other classes which need to construct new units or buildings. Also this class will contain values for the different characteristics of units and buildings which will be used when constructing new units and buildings. This gives a very easy way to tweak the game by adjusting these values.

DamageConverter

The DamageConverter is used to calculate the defence bonus for an attacked unit or building.

TheBattle

TheBattle is responsible for calculating the result of a battle. It does this by taking the firepower of all units, running this value through the damageconverter and then delivering the final damage on a specified target unit or building. This is done by calling a specific method on the player class which will then determine the end-result of the attack.

GraphAI

The GraphAI class contains all the nodes that build up the Bayesian Network. It has methods for either doing exact or approximate inference in the network. Furthermore it has methods for storing, restoring and modifying the tables for the nodes in the network. However, the main logic for doing these operations are located in the utility classes Serialize and TableModifier (see section 7.5) and therefore the primary reason for having the methods in the GraphAI class is to make the class work as a layer between the GameCore class and the two utility classes.

MoneyProvider

This class is used to give the players money by checking which of the workers belonging to a player are ready to deliver money.

IddleChecker

The responsibility of IddleChecker objects is to make sure that no buildings are left iddle when there are unit requests waiting to be taken care of. Therefore, if a building is iddle and a unit requests is recieved the IddleChecker object will make sure that the building will start producing the requested unit.

PlaceOrder

This class produces BuildCommand objects. When a player buys a worker, military unit or a building, this class is in charge of removing the price for this unit or building from the player's amount of money. It is also in charge of producing the specific build order and for placing this build order on either the players list of main base, barrack or factories build orders.

TheConstructor

This class is responsible for turning the players build commands into actual units and buildings. Additionaly it removes the corresponding build command from the player's queue of build orders and, if the player has more build orders waiting, activates the next build command in line.

GameCore

The GameCore class is as the name depicts the core of the game. It works as a layer between the use interface and timers controlling the game and classes who can perform the actual game logic. Also from this class it can be chosen which test or learning method to run.

GameAIDlg

This class is the interface for the user. Here, all the buttons and text fields are created and updated. Furthermore methods for dealing with building request (when a button is pressed) are found here. Also timers which make sure, that the GraphAI, theBattle, the MoneyProvider and the IddleChecker are called periodically are made here.

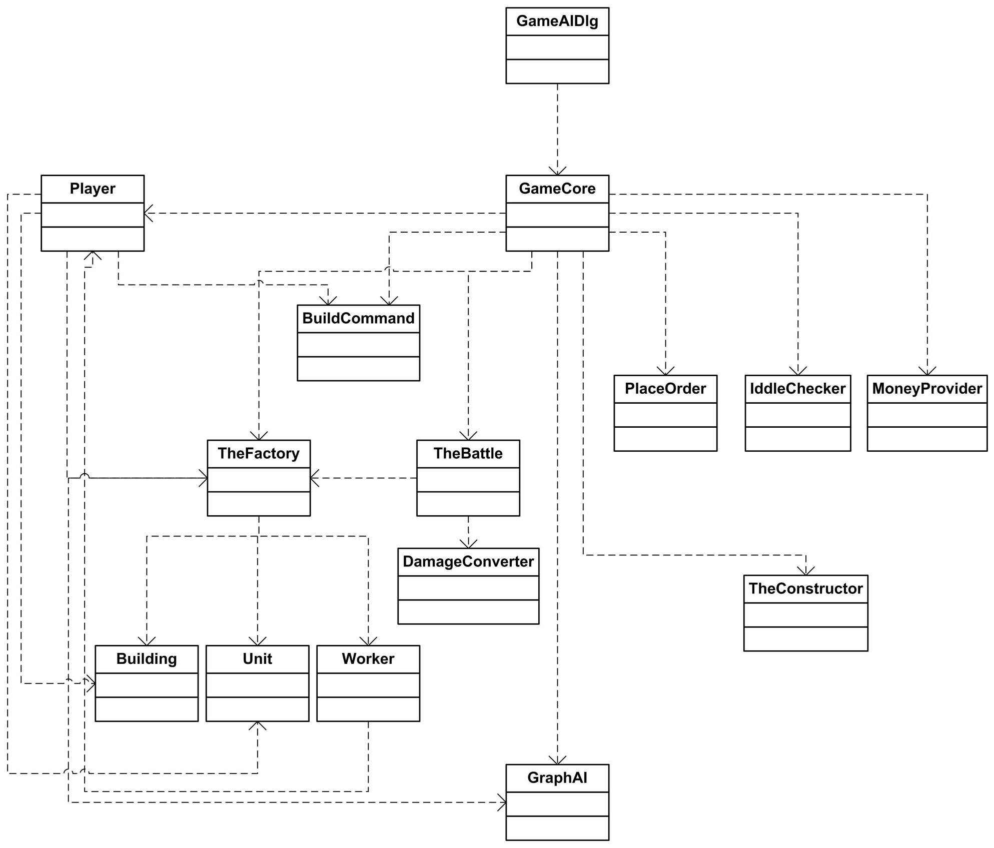


Figure 6-1 Game Class Diagram

6.1 Game Functionality

Apart from the AI, which decides what to build in a given situation, there are basically two main functionalities in the game. The player can build a unit or building and the units can attack other units and buildings. In this section the implementation of these two functionalities is shortly described.

The build order

In the game, the barracks and factories can only build one unit at a time, and the main base can only build one worker, barrack or factory at a time. However, a player, or the AI, can still make more than one request at a time. This request is just queued in the responsible buildings list of waiting build orders and the building will start making the requested unit or building as soon as all previous requests have been taken care of. This functionality has been implemented based on the command pattern [24].

Whenever a unit or building is requested, a BuildOrder object is created and placed in the responsible buildings list of build orders. This BuildOrder object is created with a value indicating what type of unit or building has been requested. When the building (for example a barrack) is ready to start building the unit, the execute method is called on the BuildOrder object. Depending on the value which the object was initialised with, it will in this method create a timer with a name for the unit type and live time for the timer.

When the timer expires, it is received in the GameAIDlg class. From here, a method for building the requested unit or building is called on the GameCore class. However the GameCore class is only responsible for calling the classes capable of performing the functionality requested by the GameAIDlg class. It therefore calls one of the methods in the TheConstructor class. This class is the one responsible for the process of constructing new units and buildings. When called it will remove the build order object either from the barrack, factory or main base and asks the player object to add the unit or building to its lists of units and buildings. The player object does this by asking the TheFactory for a new instance of this unit or building. TheFactory knows every specific detail about the requested object and can therefore return it immediately.

If more build orders are waiting the TheConstructor object will call the execute method on the next in line.

If no more build orders are waiting the building is set to be idle. The CheckForIddle object continuously checks whether buildings are idle and, if this is the case, checks whether they have received new build orders. If this is also the case, it calls the execute method on the first build order object in the list.

The Battle

When two AIs play against each other, it is important that differences in their performance are only caused by the units and buildings possessed by the players, and not on how well they are at controlling their units in a battle. Therefore, a battle function has been implemented which is used by both AIs and which makes sure that they fight in exactly the same way. If a human is playing against an AI opponent it can be chosen to only let the battle function control the AIs units.

The battle function is called every fifth second from the GameCore class. If one of the main bases is destroyed, the battle function will return this information to the GameCore which will then end the game.

Since all units are strong versus specific unit types and weak against others, the battle function makes sure, that units always attempt to fire at the unit type they are strongest against. If the opponent has no units of this type they will attempt to fire at the unit types that they are second strongest against and so on.

At each call, the total firepower of a unit type is calculated. This is done by multiplying the number of units of that unit type with the firepower for the unit type. The result makes up the total firepower of this unit type. However, depending on the armor type of the target, only a certain percentage of this firepower will be dealt as damage on the target. The final firepower is found by calling the object DamageConverter with information about the weapon type of the attacking unit and the armor type of the defending unit.

Once the actual firepower of a unit type has been found the damage is dealt on the attacked units. These units are stored in a list in the player object. It is then determined how many units should be removed from this list and in the end, when the remaining damage is not large enough to kill a unit, the health of the first unit in the list is decreased with the remaining damage.

7 AI – The Bayesian Network

This chapter will describe the resource management AI in the game, what this AI is supposed to do and how it will accomplish this.

7.1 What do we build?

The AI in the game is going to have one basic functionality, namely to figure out what unit will be smartest to build in a given situation. The AI is to be called periodically from the game loop and at each call return its answer to classes in charge of actually building the unit or building requested.

The AI will of course need some information about the current game situation to be able to figure out what unit or building to build. This information could come from a high-level strategic AI informing the resource management whether it was planning a stealth attack, whether there was a threat of enemy air attack etc. However, such a high-level strategic AI is not going to be implemented for this game since the focus needs to remain on the resource management AI. Instead the resource management AI will get all of the following information at each call:

- How much money does the player (computer) have?
- How many of each unit type does the player (computer) have?
- How many of each unit type does the enemy have?
- How many of each building type does the player (computer) have?

Based on this information and on knowledge concerning the advantages and disadvantages of each unit and building, the AI is to be able to decide which unit or building to build at a particular time.

7.2 Identifying the needed random variables

In the analysis it was decided to develop the AI using Bayesian Networks. This section will identify the random variables needed in the network.

The output from the network will be a choice between either building a worker, military unit, a building or nothing at all. Therefore, a random variable is needed from which it will be possible to make this choice once we know the distribution of the random variable. This random variable is called `WhatDoWeBuild`.

However the table for this random variable would be very large if all information from the domain was to be represented in it. Therefore other random variables are needed that will affect which state the `WhatDoWeBuild` variable takes. These variables can be divided into two groups: external random variables, which reflect the current game situation and which cannot be altered unless the entity they represent changes state; and internal random variables which can be altered as the result of other random variables taking on particular values. Table 7-1 shows the external random variables.

Table 7-1 External random variables

Name	Description	Possible values
NumberOfWorkers	Shows the number of workers owned by the computer	0 → 10
NumberOfOwnLightSoldiers	Shows the number of lightsoldiers in the computers army	0 → ∞
NumberOfEnemyLightSoldiers	Shows the number of lightsoldiers in the enemys army	0 → ∞
NumberOfOwnHeavySoldiers	Shows the number of heavysoldiers in the computers army	0 → ∞
NumberOfEnemyHeavtSoldiers	Shows the number of lightsoldiers in the computers army	0 → ∞
NumberOfOwnLightTanks	Shows the number of lighttanks in the computers army	0 → ∞
NumberOfEnemyLightTanks	Shows the number of lighttanks in the computers army	0 → ∞
NumberOfOwnHeavyTanks	Shows the number of heavytanks in the computers army	0 → ∞
NumberOfEnemyHeavyTanks	Shows the number of heavytanks in the enemies army	0 → ∞
NumberOfBarracks	Shows the number of Barracks owned by the computer	0 → 4 (values above 4 will be considered as equal to 4)
NumberOfFactories	Shows the number of Factories owned by the computer	0 → 4 (values above 4 will be considered as equal to 4)
Money (M)	Shows the amount of money possessed by the computer	0=0 <= M< LS price 1=LSPrice<= M < HS price 2=HSPrice<= M < LT price 3=LTPrice <= M < HT price 4=HTPrice <= M < BarPrice 5=BarPrice<= M < FactPrice 6= FactoryPrice <= M (Bar=Barrack,Fac= Factory)

With the external random variables in place, the needed internal random variables can be determined. First of all, the number of units belonging to both the AI units and the opponent needs to be compared to give a clear picture of the current strength status for each unit type. Since this will give an indication of which units are needed and which are not particularly needed, this will strongly influence the final choice of what to build.

This leads to five random variables. Four for comparing the individual unit types and one which, based on the value from the previous four, will determine the overall status of unit strengths, and also how much a unit from each unit type is needed (Table 7-2).

Table 7-2 Five random variables expressing unit status

Name	**Description**	**Possible values**
LSstatus	Shows whether the computer has more or as many lightsoldiers as the enemy or whether the enemy has more	1 (more or equal) 2 (less)
HSstatus	Shows whether the computer has more or as many heavysoldiers as the enemy or whether the enemy has more	1 (more or equal) 2 (less)
LTstatus	Shows whether the computer has more or as many lighttanks as the enemy or whether the enemy has more	1 (more or equal) 2 (less)
HTstatus	Shows whether the computer has more or as many heavytanks as the enemy or whether the enemy has more	1 (more or equal) 2 (less)
WhichUnitIsNeeded	Determines which unit the computer needs the most at a specific time	0 = lighsoldier 1 = heavysoldier 2 = lighttanks 3 = heavytank

The variables shown above give a picture of which units are needed. However, before deciding to build a specific unit, it needs to be decided whether to build a military unit or a worker. This decision could have been made by letting WhichUnitIsNeeded depend on the amount of workers available. However since WhichUnitIsNeeded shows the status and needs for military units only, it has been chosen to let the choice between a worker and a military unit be made by its own random variable called UnitOrWorker (Table 7-3).

Table 7-3 A random variable, deciding whether to construct a unit or a worker

Name	**Description**	**Possible values**
UnitOrWorker	Shows the needs for workers and military units	0 = worker 1 = lightsoldier 2 = heavysoldier 3 = lighttank 4 = heavytank

Next it will be determined whether the unit needed really is what is needed most, or whether it would be better to construct a building, which can build this unit type or, if a building already exists, will make it faster to build units of that type in the future.

This decision will be made in three nodes; SoldierOrBarrack, TankOrFactory and BuildWorker (Table 7-4). These three will all have the UnitOrWorker node as parent and beside of that

– the SoldierOrBarrack will have the external node NumberOfBarracks as parent,

– the TankOrFactory will have the NumberOfFactories node as parent

– and BuildWorker will have NumberOfWorkers as parent.

Table 7-4 Random variables, deciding whether to build units, buildings or workers

Name	Description	Possible values
SoldierOrBarrack	Tells whether a lightsoldier, heavysoldier, barrack or nothing is needed	0 = nothing 1 = lightsoldier 2 = heavysoldier 3 = barrack
TankOrFactory	Tells whether a lighttank, heavytank, factory or nothing is needed	0 = nothing 1 = lighttank 2 = heavytank 3 = factory
BuildWorker	Tell whether a worker or nothing is needed	0 = nothing 1 = worker

At last it will now be possible to decide what to build. The last node WhatDoWeBuild (Table 7-5) will have four parents; the three before mentioned nodes SoldierOrBarrack, TankOrFactory and BuildWorker, and the node Money.

Table 7-5 Random variable, deciding what to build

Name	Description	Possible values
WhatDoWeBuild	The final decision point of what to actually build.	0 = Nothing 1 = worker 2 = lightsoldier 3 = heavysoldier 4 = lighttank 5 = heavytank 6 = Barrack 7 = Factory

In this node, it will first be determined what the build request is. After this, the price for that build request will be compared to the current money situation. If there is enough money the requested unit or building will be constructed and otherwise nothing will be constructed at all.

The full Bayesian network can be seen in Figure 7-1.

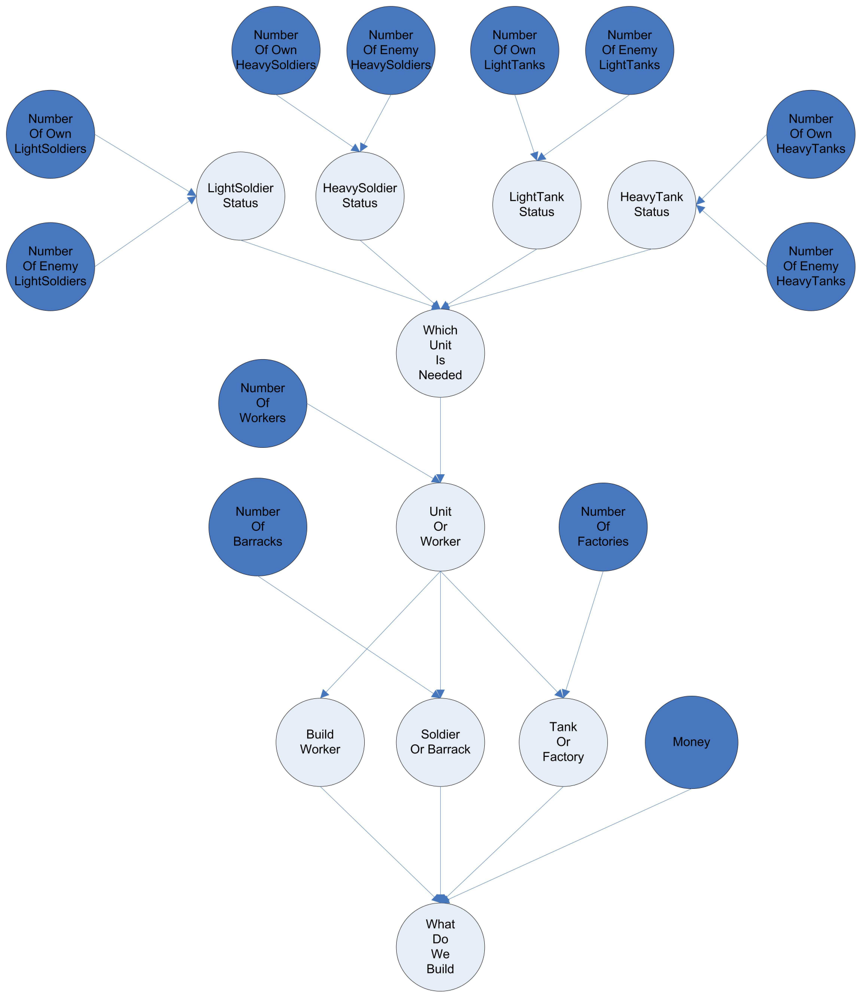


Figure 7-1 The Bayesian Network

7.3 Design decisions

The design of the Bayesian network has undergone many changes through its development. In this section some of the major design issues are explained.

One design issue was to let SoldierOrBarrack, TankOrFactory and BuildWorker be three individual nodes, where instead there could have been one single node having a larger table. The reason for choosing this design is that, depending on the value from the UnitOrWorker node, almost 2/3 of the rows in the table for the combined node would be excluded from becoming the chosen value of the table. For example, if the UnitOrWorker node returns the value 1, meaning that the wanted unit is a lightsoldier, the information about the number of factories available and about whether a lighttank, heavytank and worker is needed becomes irrelevant and would instead be the course of an unnecessarily large table.

It was therefore chosen to split this large node up into three nodes with smaller tables. However though this design separates information not relying on each other from each other, this choice also has some disadvantages. If we again imagine that the UnitOrWorker node returns the value 1, this will lead to the tables BuildTankOfFactory and BuildWorker both returning a 0 meaning build nothing. This is obvious when looking at the diagram for the

Bayesian network (Figure 7-1), however the computer would need to explore all the rows in the tables from the two nodes to figure this out.

Another design issue was the location of the `Money` node. In many variations of the design the `Money` node was placed at the top of the graph and affected therefore the requests for specific units more than it does now when placed at the bottom of the graph.

The primary reason for the chosen design came from trying to think about how one would normally decide what unit to build. If one would normally look at the amount of money available and from this decide what to build, then the first designs would probably have been a better choice. However, often this is not what happens. Normally your needs would come from other elements in the game; your own strategy, the enemies strategy, the number of a particular kind of unit in your army and in the enemies army and so on. Thinking about it this way, then money should not decide what to build. Instead, if one has less money than the price of the needed unit or building, one would simply wait a while until more money had been received. If the `Money` node was placed at the top of the graph, the requested unit in some situations would become a unit that was affordable instead of a unit that was not affordable but in fact needed.

7.4 The graph framework

In the previous sections the random variables needed for the network were determined. With these in place the next step is to implement the network from these variables. A simple solution to this would be to implement the network as a single file and when dealing with only a small number of variables this would indeed be easy. The problem with this solution is that it is very static and does not easily allow for modifications of the network and, when the number of variables increases, this problem becomes even more apparent.

Therefore a more dynamic solution has been developed for this project. In this solution every node is represented as an object of a node class and each node has references to its parents which are used when traversing the network. In this way new nodes can easily be added to the network by creating node objects and adding references to this object in its children. However 6 different types of node classes are needed (see Figure 7-2) since the way a node will determine its return value(state) is different from node to node. In the following, the different nodes are described.

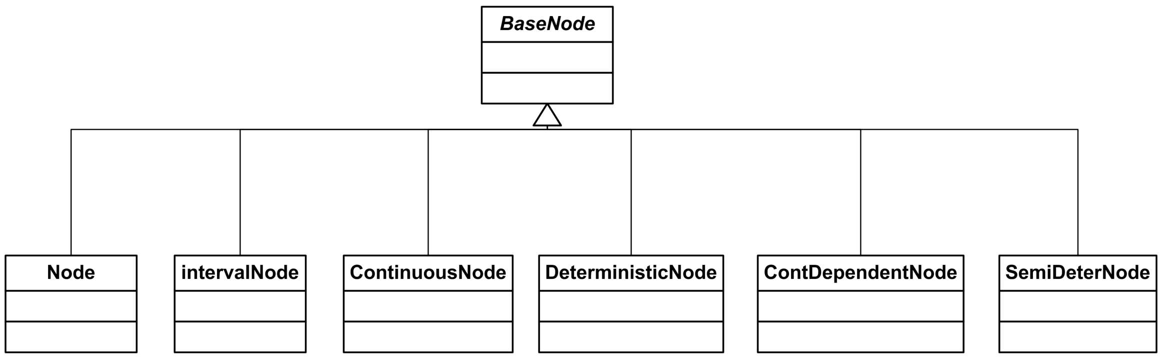


Figure 7-2 Graph framework classes

Node

A node of this type uses the values returned from its parents as index for looking up its own return value in its table.

ContinuouslyNode

This type of node stores information about an entity from the problem domain that can take on any value from 0 to infinity. These nodes will have no table but will simply return the number that will be set before the AI algorithm (query algorithm – see 7.6) is called.

ContDependentNode

Nodes of this type only have two `ContinuouslyNodes` as parents. When asked for a value a `ContDependentNode` will compare the values of its two parents and return a number indicating whether the value of parent 1 is larger or smaller than the value of parent 2.

IntervalNode

An example of this node is a `Money` node. Depending on the amount of money the player has, this node will return a particular value. To determine what value to return, this node will check in which interval of values the current amount of money is. This could be below the price of a worker, between the price of the worker and the price of a heavysoldier and so on. These intervals will be stored in a table belonging to the node.

DeterministicNode

This node can take on a value from a predefined set of possible values. For example the `Worker` node can take a value from 1 to 10. This node will have no table.

SemiDeterministicNode

This node can also take on a value from a predifined set of possible values. However, this node will have an upper limit, meaning that if the attribute takes a value larger than this value, the return value from the node will be set to the maximum value. An example of this is the `NumberOfBarracks` node, which can take a value from 0 to 4 though a player might build 5 barracks or more.

BaseNode

Finally a super class node is needed from which all other node types will inherit. This prevents the nodes from having to know about all other types of nodes when storing pointers to their parents. Instead they only have to know the super class node.

This framework makes it possible to easily build up the network and also modify it by simply creating or deleting node objects and by rearranging the references between these node objects. Section 7.6 describes the algorithm that makes it possible to get an answer from the network.

7.5 Learning and testing implementation

The graph framework makes it easy to build a network and to modify it. However, to train the network and to observe the results of this training other classes are needed. These classes are described in this section.

TableProducer

Before the AI can return a result of which unit or building to construct, the tables for each node in the network need to be initialized. That is, the probability distribution given each possible combination of parent states needs to be written to a table. This could have been done manually but since the tables quickly become very large this would be a cumbersome process. Therefore, the `TableProducer` class has been implemented. This class can construct random tables for the AI graph. Each table produced will follow the specified restrictions for each table, but will besides of that contain purely random numbers. For instance, in the `SoldierOrBarrack` table, it would not make much sense to have a distribution saying that the probability of building a soldier was 0.8 though the player had no barracks from where this soldier could be build. The probability of building a soldier therefore needs to be 0 when no barracks are present.

TableModifier

This class is used for training the AI. It has several methods which in different ways can modify the contents of the tables in the AI graph. The specific modification methods are described in the next chapter.

Monte Carlo

This class is an implementation of the Monte Carlo learning algorithm. It has methods for storing information about what is built in a given state and methods for modifying a table based in this information.

Serializer

This class is in charge of writing and reading tables from the hard disk. When the AI has been trained, it is important that what has been learned does not disappear when the application is closed. Therefore it is necessary to make the tables from the AI graph persistent so that they later can be read into the application again. This also enables for quickly exchanging a table with another table.

ResultWriter

This class is used for saving the results of two AIs playing against each other. If the AIs for instance are playing to 10, then a new file containing all the intermediate results will be made whenever one of the AIs reaches 10 points. Also pure text can be written to a file via this class to explain the results from a test.

TestRunner

This class is responsible for controlling the various learning methods for example by keeping track of the number of iterations performed in a learning method so far, and from this decides what should be modified and how this should be done. However, the actual modifications is done by either the TableModifier class or the MonteCarlo class

The next figure (Figure 7-3) shows the class diagram for the AI related classes.

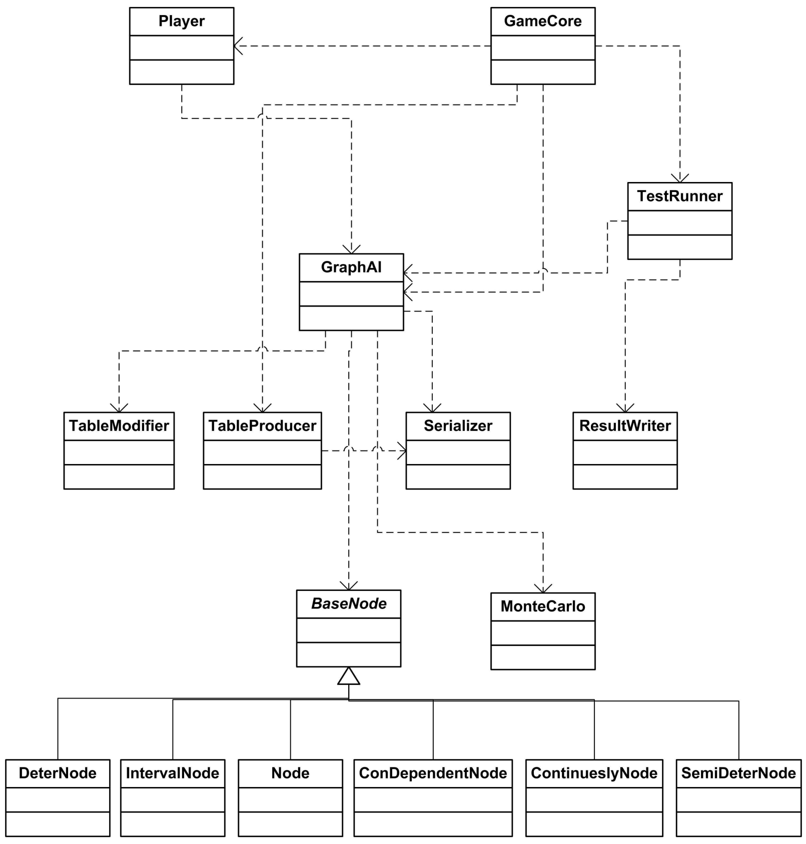


Figure 7-3 AI class diagram

7.6 The Query Algorithm

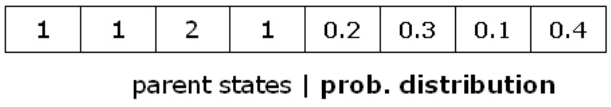
A directed arch from one node to another node shows that the parent node affects the child node. This means that for all nodes which have parent nodes, we cannot get the state from the node before we know the state of the parent nodes. In the network described in the last chapter (see Figure 7-1), the node which eventually decides what to build is the node WhatDoWeBuild. Since this is a leaf node, we cannot get an answer from it before the state of all its parent nodes have been found.

A solution to this would be to start at the root of the graph, given by nodes without parents, and let them send their state to all of their children nodes and again letting these children send their state to their children until we are at the WhatDoWeBuild node which can give us the result we are looking for. However, this requires that we know all root nodes and ask all of these nodes to start sending their values to their children.

The approach chosen for this project works the opposite way. Instead of starting at the root of the graph, this approach starts at the leaf node. Before returning a state, this node will call all of its parents and ask for their state. They will again ask all their parents until a node is reached which has no parents and this node will then return its state to the caller node. In this way the graph will be traversed by recursive calls and will only require for us to know one leaf node instead of a large number of root nodes.

This approach is well suited to this project because the network only has one leaf node. Had there been more leaf nodes the algorithm might as well have started at the root of the graph, as for example the message parsing algorithm does [2].

The actual query algorithm in this project consists of a GetProbability method in all node objects and a table for each node showing the probabilities of the possible states the node can take given the states of its parents. When the GetProbability function is called on a node, the node will call the same method on all its parents and store the returned values in a container. When the state has been received from all parents, the node will use these values as entries for its probability table to find its probability distribution given the current states of its parents.
Figure 7-4 shows an example of a row from the table belonging to the WhichUnitIsNeeded node.



1	1	2	1	0.2	0.3	0.1	0.4

parent states | **prob. distribution**

Figure 7-4 Example of probability distribution

Since the node has four parents, the first four values show the states of these parents and the last four values show the probability for each of the possible states the WhichUnitIsNeeded node can take. When the correct row has been found, a random choice based on the probability distribution in the row is made to decide which state to take and a notification of the chosen state is returned to the child node.

As was mentioned before, the graph framework allows for dynamic networks. This of course means that the query algorithm also needs to be dynamic. It would not be a good solution if a node would return very specific information about which state it was currently in. For example if the WhichUnitIsNeeded node returned the string LightsoldierNeeded as an indication of which state it was currently in, the child node would need to know exactly what this information meant. Therefore the structure of the network that allows for easy modifications would not be of any benefit since the algorithm would need to be modified as well to also work under the new network structure.

Instead a more general state notification is used, namely the index value for the chosen state. If the WhichUnitIsNeeded node (Figure 7-4) took the state with the probability 0.3, the WhichUnitIsNeeded node would return the number ‘1’ since this would be the index if the possible states were placed in their own vector or array. This is illustrated by Figure 7-5.

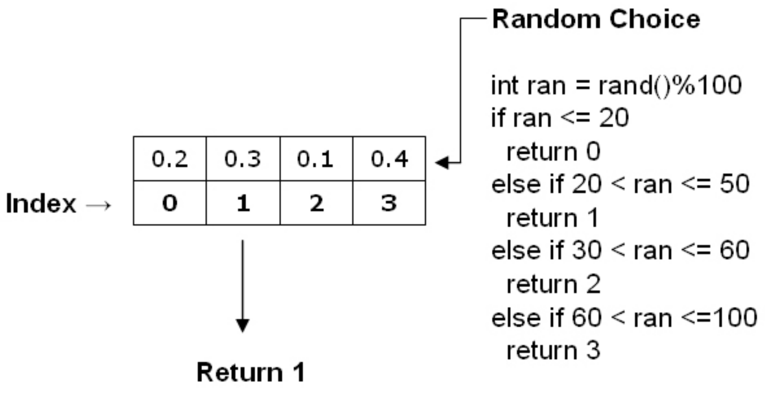


Figure 7-5 Random Choice Example

Now the child node does not need to know what exactly it has received from its parent but can simply use the received information as lookup values in its own table.

The only problem with the graph framework and the query algorithm is that when changes are made to the network it is necessary to also change the tables belonging to the nodes. However, the methods in the class TableProducer which can be used to create tables with random contents are fairly easily modified to take the changes of relationships among nodes into account. The following is an example of how an iteration of the network might take place (see Figure 7-6 for the example and Figure 7-1 for the original Bayesian Network).

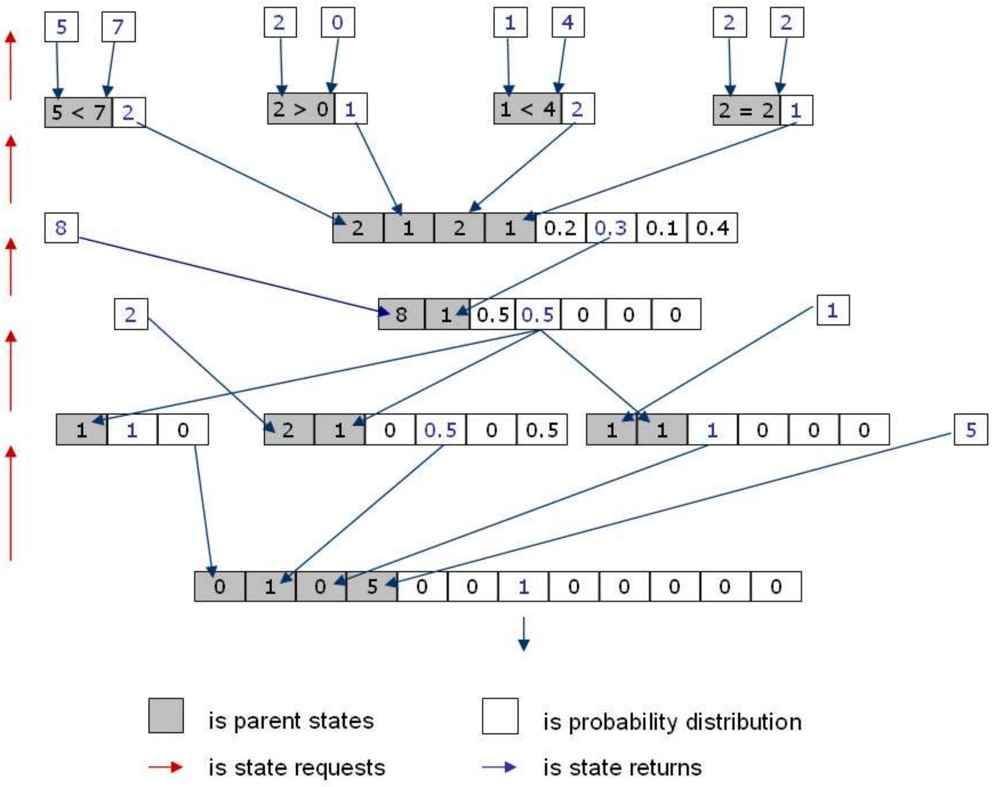


Figure 7-6 AI algorithm example

First the GetProbability method is called on the BuildThis node. This node calls all of its parents for their states and uses these values as lookups in its table. The table at the bottom of the figure shows the row selected from these parent values.

As can be seen, not all of the nodes select their return values from probability distributions. Instead they might return a number showing how many units of a given unit type the AI has or showing whether the value from the first parent is larger than the value returned from the second parent. In these situations it is the actual number that is returned. However, when the return value is chosen based on a probability distribution, it is the index of the chosen state from the container of possible states that is returned. For example, in the bottom row, the first four values are parent values and the last 8 values are the possible attributes that node can take. Therefore, the value returned from this node would be the number 2 since the chosen state has the index 2 in the container of possible states (starting with index 0).

7.7 Conclusion on AI-The Bayesian Network

In this chapter the Bayesian network has been designed and a small framework for easily modifying the network has been described.

Furthomer, various classes for testing the AI has been described as has the algorithm used for getting an answer from the network.

8 Learning

The previous chapters dealt with designing a RTS game and an AI for this game that can decide which units or buildings to construct during the game. This chapter will discuss how the AI can become better at making these decisions. This was one of the main overall goals of this project, namely to develop an AI for a RTS game which could adapt to new strategies.

8.1 What is a strategy?

One of the first sections in this thesis gave a general introduction to the elements of RTS games and also described some overall strategies for such a game. It is these types of strategies that the AI has to learn. Also it has to learn how the strategy used by the opponent can best be countered. For example the AI has to learn how to make a fast attack, how to use primarily one specific unit and so on.

The way an AI can be told to follow a given strategy is by altering the values in the tables belonging to the nodes in the network. These values show the probability of the different options the AI has in a given situation. Therefore, by tuning these values the AI can be made to prefer for instance lightsoldiers or military units over workers and can in this way be said to be following a given strategy.

An interesting aspect of RTS games is that though there are many different strategies, there is no strategy that always leads to victory. All strategies have some counter strategy and therefore a player needs to continuously adjust his strategy to the strategy followed by the opponent. Of course in some situations the followed strategy might be better than the one followed by the opponent and therefore no adjustments are needed.

In this thesis the AI has a given strategy when the game starts. This strategy is given by the table values for the game situation where neither of the players has any units or buildings. If for example these values show a high probability for building heavytanks then building heavytanks will be the starting strategy for the AI. However, as soon as the opponent starts producing units, the AI needs to also take these into account. If the opponent is producing heavysoldiers then heavytanks is a poor unit to build and this needs to be reflected in the table values for that particular game situation.

The total strategy of an AI therefore is the table values for all possible game situations or said in another way, the tables describe different strategies depending in the game situation.

8.2 Ways of learning a strategy – the state space

A way to learn a good strategy against an opponent could be a supervised learning approach. That is, the AI could be taught by an instructor what the best action was in a given game situation. However, this makes it impossible for the AI to learn new strategies on its own and the AI would therefore not be able to adapt to new opponent playing styles. Another approach could be unsupervised learning but as it was discussed previously, the problem with this approach is that new classes of patterns need to be developed as new strategies are developed. Also patterns might start to overlap when modifications are made to a strategy. This can make it difficult to determine which class of strategy the current opponent strategy belongs to, which again would make it difficult to choose the best counter strategy.

Therefore a reinforcement learning approach will be used instead. In this approach the AI explores the space of possible states (table values) and is rewarded when making a good choice.

As described before, there is a good counter strategy to all strategies. Therefore, when the AI is playing against a given opponent, it must search through the space of possible states (setting of tables) for the optimal state. This is illustrated in the following figure where x denotes the opponent, x denotes the current AI state and the x denotes to optimal counter strategy to the opponents strategy (Figure 8-1).

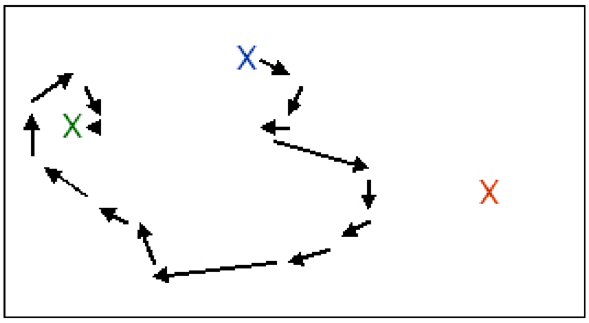


Figure 8-1 The state space

The question is how the AI is to move around the state space in search for the optimal counter strategy. This could happen by small random jumps which always were the same length or the length could be determined from the relative strength between the two players. Another approach could be to adjust one parameter (table) at a time while keeping the others fixed.

In the following four different learning methods will be described. These methods all search through the state space in different ways and in the next chapter they will be compared to find out which of them performs best.

8.2.1 Random IndexAndSign

The Random IndexAndSign method randomly selects which entries in a table to modify and also randomly decides whether to increase or decrease the value of an entry. However, the value used for modifying the table stays fixed. This means, that the method does not receive any reinforcement but instead explores the state space at total random.

The problem with this method is that it in theory could jump around the state space forever without finding a good solution. On the other hand it does not run the chance of getting stuck in a local optimal state since it will modify the tables no matter what the outcome of the last game was.

8.2.2 Random IndexValueAndSign

This approach is similar to the previous method, but modifies the tables by a value depending on the relative strength between the two players. For example, the players might play 100 games and if the AI being trained wins 70 out of the 100 games it is not modified or only modified a little bit. If however the AI wins only 30 games it is modified with a larger modification value.

This means, that this algorithm makes use of the reinforcement received from the previous game played. This reinforcement will show whether the AI won or lost the previous game and also how much better or worse the AI performed compared to its opponent (via the score). Based on this information the algorithm will decide what the modification value should be.

The problem with this method is that it might get stuck in a local point of the state space. This would happen if a state was found that performed ok but which was not the optimal one. This would result in very small modification values which would make it difficult for the algorithm to get away from the local optimal state.

8.2.3 Simulated Annealing

The idea behind simulated annealing comes from the process of finding the optimal state of a material by heating it until the atoms become unstuck from their initial positions and move around freely. Once this state is reached the material is slowly cooled down by decreasing the temperature. This lets the atoms bind in configurations with lower internal energy than the initial configuration, resulting in a material with fewer defects than the original material [29].

In this thesis, the principle of simulated annealing is used as another learning method which adds an additional level to the two previous methods to make it more likely that the optimal solution is found. The algorithm starts by creating a number of completely random tables and determines which of them plays best against a given opponent. This table is used as a template for a new generation of tables which are all small modifications of the template table. The tables from the new generation play against the opponent again and the best is selected as template for the next generation of tables. At each generation the value used for modifying the tables becomes smaller and smaller. This continues until the modification is so small that it does not really make any difference or until no table from a new generation performed better than the template table. This process is illustrated in Figure 8-2 where each circle represents a table (state) and the size of the coloured circles shows the size of the modifications made to the tables based on the template table. Furthermore, the algorithm is presented via pseudo code in Figure 8-3. Though the algorithm here has been described when training one table only, it works similar when training two or more tables at a time.

Though the simulated annealing algorithm was not one of the described algorithms in the Reinforcement Learning chapter, it is still using some of the concepts described in that chapter. As explained, a reinforcement learning algorithm can either choose to exploit what it knows to be the best choice in a given situation or explore to see if it can find solutions or actions leading to even higher rewards. The simulated annealing method first explores the state space by creating a number of random tables. Once this is found the algorithm exploits this table by using it as a template for the next generation of tables. For each of the modification iterations the algorithm explores less and less until the best state has been found. Therefore, the algorithm starts by only exploring and ends by exploiting more and more.

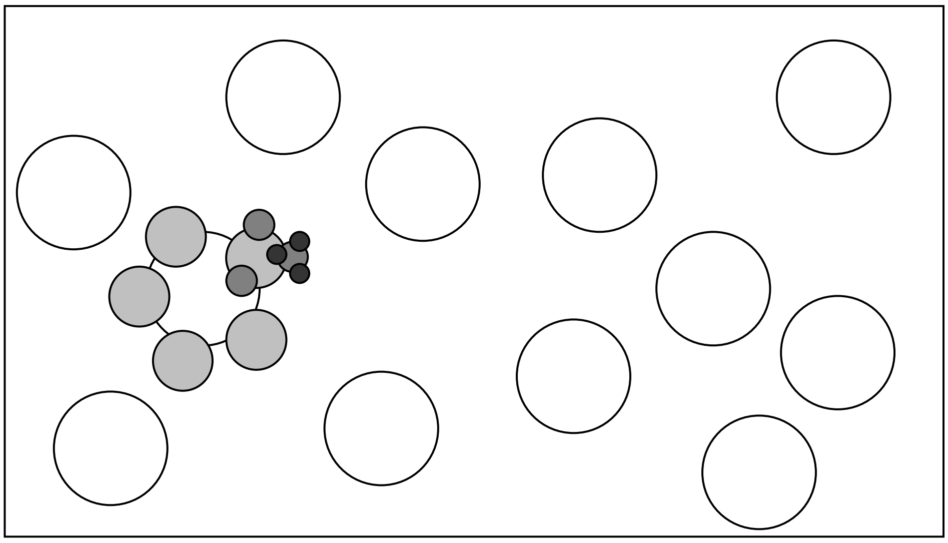


Figure 8-2 Simulated annealing

```
create 100 needs AIs

for(i 1->numberOfRandomAIs)
{
  for(j 1->gamesToPlay)
  {
    fixedAI battle RandomAI
    if(winner==stableAI)
      fixedAIWins += 1;
    else
      randomAI_iWins +=1;
  }
  if(randomAI_iWins > maxwins)
  {
    maxwins = randomAI_iWins;
    storedAI = RandomAI_i
  }
  fixedAI = 0
}

for(i 1 -> MaxPrecision)
{
  for(j 1 -> NumberOfExplorations)
  {
    new AI tmp = storedAI;
    modify(tmp, (1/MaxPrecision)
    for(h 1 -> gamesToPlay)
    {
      fixedAI battle tmp;
      if(winner== fixedAI)
        fixedAIWins += 1;
      else
        tmpWins +=1;
      if(tmpWins >MaxVics)
      {
        MaxVics = tmpWins;
        bestAI = tmp;
      }
    }
  }
  storedAI = bestAI;
}
```

Figure 8-3 Pseudo code for the simulated annealing algorithm

8.2.4 Monte Carlo

Whereas the previous methods only modified the tables based on the outcome of a game, the Monte Carlo method implemented in this thesis also uses intermediate steps in a game to decide what and how to modify the tables.

The algorithm works as follows:

Whenever a choice is made in a table, this choice and the state of the table at that particular time is stored in a container. In this way it is possible to see for example which types of units were build most when being in a particular state. Depending on the outcome of the game (whether the AI won or lost) the tables can be modified according to this information. For example, if the AI lost the game and in a particular state preferred to build lighttanks, then the probability for building lighttanks should be decreased for this state. If, on the other hand, the AI had won the game it could be tried to further increase the possibility of building a lighttank in that state to for instance decrease the time it took to win the game, to make it possible to win more games in a row etc.

The pseudo code shown in Figure 8-4 illustrates a very simplified example. In the actual code this process needs to be done for all the tables from the network which is going through training.

```
vector statevector;

unit; (the unit type that was create at
this iteration)

value;

while(game runs)
 if(currentstate isIn statevector)
 statevector[currentstate][unit] += 1
 else
 statevector.push_back(current state)
 statevector[currentvector][unit]+=1

for(i < statevector.size)
 state = statevector[i]
 unit = findMaxUnitType(AI[state])
 AI[state][unit] += value
```

Figure 8-4 Monte Carlo algorithm

8.3 Conclusion on learning

The result of this chapter has been four algorithms that are used to train the Bayesian Network by modifying the values in the tables belonging to the nodes of the network. These algorithms are the Random IndexAndSign algorithm, the Random IndexValueAndSign algorithm, the Simulated Annealing algorithm and the Monte Carlo algorithm.

In the next chapter these four algorithms will be tested to see which of them is best at training an AI to become better than a given opponent.

9 Experiments

In this chapter various experiments with the AI are performed to test how effective the AI can become and how effective the different learning algorithms are.

The AI is tested to see whether it can become better than a fixed opponent. This illustrates the situation where a new strategy is being used by the opponent. Also experiments are to show whether the AI can become better than various opponents with different strategies or whether it can only become better than a single opponent. That is, can it become good in general?

Four tables from the Bayesian network will undergo training. These tables are the WhichUnitIsNeeded table, the UnitOrWorker table, the SoldierOrBarrack table and the TankOrFactory table. These tables were described in section 7.2 and are the only tables from the network in which actual decisions are being made.

The learning algorithms used for training the four tables are the RandomIndexAndSign algorithm, the RandomIndexValueAndSign algorithm, the Monte Carlo algorithm and last the Simulated Annealing algorithm. These algorithms were described in chapter 8. In this chapter the algorithms will be tested to see which of them performs best. This is determined primarily on the win percentage when playing against an opponent AI but also by how fast the AI can converge to become better than the opponent when being trained by the algorithms.

Besides of showing how well the different learning algorithms perform, the experiments will also show how the number of tables being trained at a time affects the time it takes for the AI to become better than an opponent. That is, the more tables trained at a time, the larger the state space becomes and therefore the convergence rate, or the time it takes to find the optimal strategy, might be affected.

Each experiment will have a short introduction describing the purpose of the experiment and also describing the setting of the experiment, for instance how many games are played, how is the modification value determined etc.

The following experiments will be performed:

Experiment 1	Reference test - same playing conditions for both AIs
Experiment 2	Can an AI become better than an opponent AI?
Experiment 3	Training the WhichUnitIsNeeded table
Experiment 4	Training the UnitOrWorker table
Experiment 5	Training the SoldierOrBarrack table
Experiment 6	Training the TankOrFactoruy table
Experiment 7	Training all tables at the same time
Experiment 8	Which algorithm found the best AI?
Experiment 9	Generalization

Preparing the learning algorithms

For each experiment the algorithms used for training the AI will be initialized with some values specifying, how many iterations the algorithm is to run, how many random tables are to be created etc. To prevent having to explain for each experiment what the values used for initializing the algorithm mean, they will shortly be described here:

- *GamesToBePlayed*. This variable specifies how many games two AIs have to play against each other before determining which of them is best. That is, how many games should be played before for example one of the AI is modified by a learning algorithm. The variable could also be said to specify the length of one round of games.
- *NumberOfModifications*. This variable specifies how many times an AI is to be modified during an experiment. That is, how many different states is the AI going to be in during the experiment?
- *NumberOfRandomTables*. The Simulated Annealing method starts by producing a number of random tables. From these tables it finds the best and uses this as a template for producing a new generation of tables (see section 8.2.3). The number of tables generated at the beginning of the algorithm is specified by the NumberOfRandomTables variable.
- *NumberOfIterations*. This variable specifies how many generations of modified tables are to be produced during the Simulated Annealing algorithm. At each of these iterations, the value used for producing new variations of the template table is decreased.
- *NumberOfTableVariations*. This variable specifies how large the produced generations of modified tables are going to be in the Simulated Annealing algorithm.

9.1 Experiment 1

Purpose
The purpose of this experiment is to test whether two AIs perform equally well when playing under the exact same conditions. This is needed to make sure that changes in AI performance in later experiments are due to differences in the AIs and not to some other aspect of the program.

Condition
In this experiment no learning algorithm is used. Instead two identical AIs will play against each other and the *GamesToBePlayed* variable is set to 100. Since the AIs will have the exact same tables they ought to have the same chance of winning the game.

The experiment is performed three times and in each experiment the WhichUnitIsNeeded table in the two AIs is exchanged with another WhichUnitIsNeeded table. This is done to make sure the result shows the general properties of the AIs.

The result of the experiment is shown in Figure 9-1 to Figure 9-3. These figures show the intermediate number of games won by AI1 and AI2 while they played the 100 games.

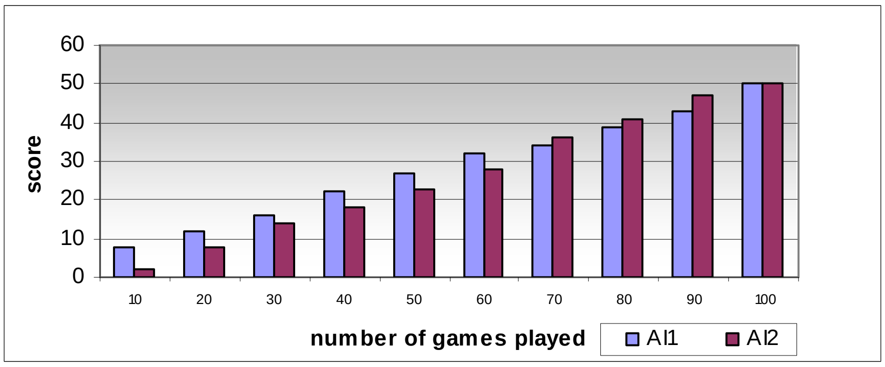


Figure 9-1 Experiment 1/1

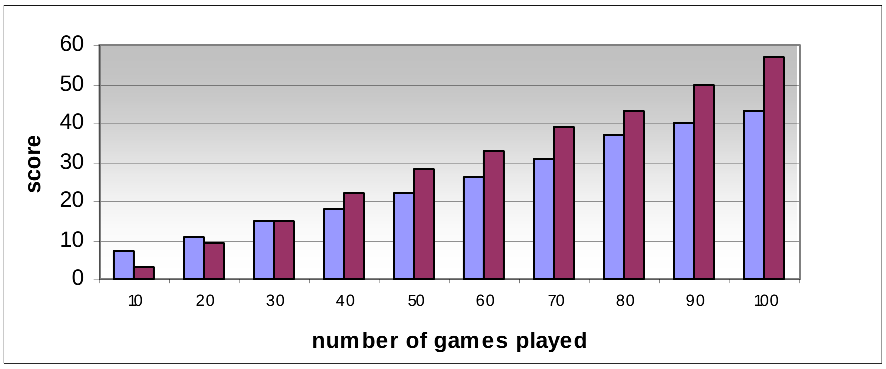


Figure 9-2 Experiment 1/2

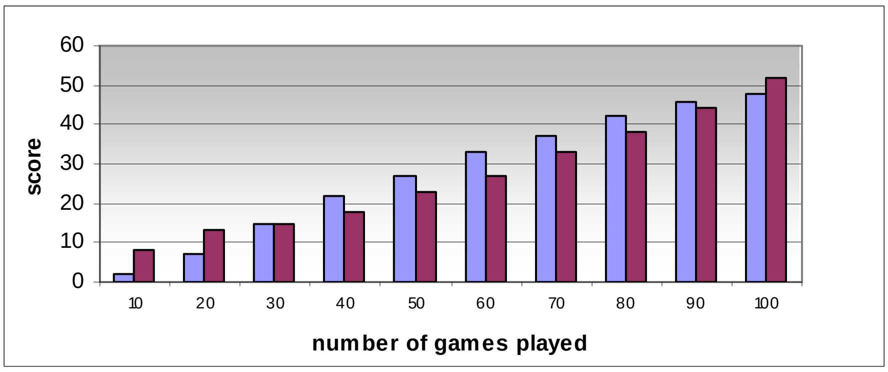


Figure 9-3 Experiment 1/3

Conclusion on experiment 1

The result of experiment 1 shows that the AIs do approximately win an equal number of games when playing under the same conditions, which was also expected (see Figure 9-1 og Figure 9-3). This means, that differences in two competing AIs only can be caused by their tables.

However, as can been seen from Figure 9-2, the result might not always be exactly equal between two identical AIs and therefore a score of 55-45 between two AIs does not necessarily mean that one is better than the other. Instead, it might be the result of coincidence. Just as for example tossing a coin 100 times not always results in 50 heads and 50 tails.

9.2 Experiment 2

Purpose

This experiment will test whether an AI *can* become better than another AI, and will do so by training AI2.

Condition

In this experiment the learning algorithm used is the simulated annealing algorithm (see section 8.2.3). This algorithm will modify the WhichUnitIsNeeded table only since this will decrease the possible state space. The algorithm is initialized with the following values:

GamesToBePlayed: 80

NumberOfRandomTables: 20

Number of iterations: 3

NumberOfTableVariations: 20

The first diagram for this experiment (Figure 9-4) shows how well the 20 initially randomly created tables performed against the fixed opponent AI. The following three diagrams

(Figure 9-5 to Figure 9-7) show the performance of the three generations of modified tables produced based on the best table from the previous generation.

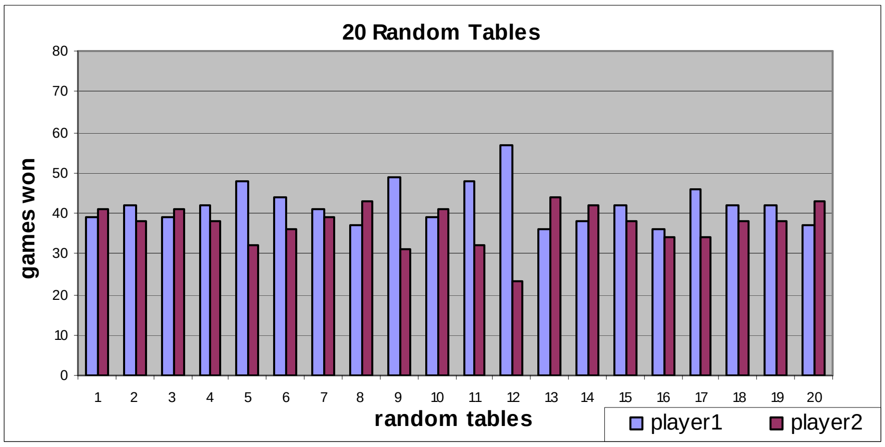


Figure 9-4

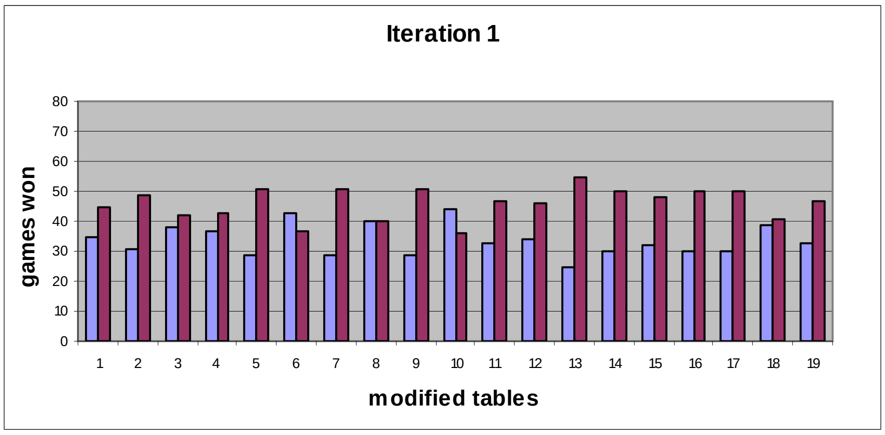


Figure 9-5

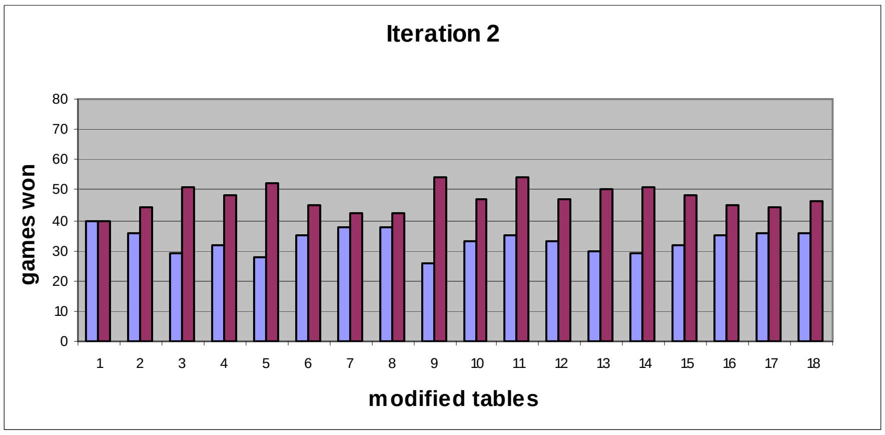


Figure 9-6

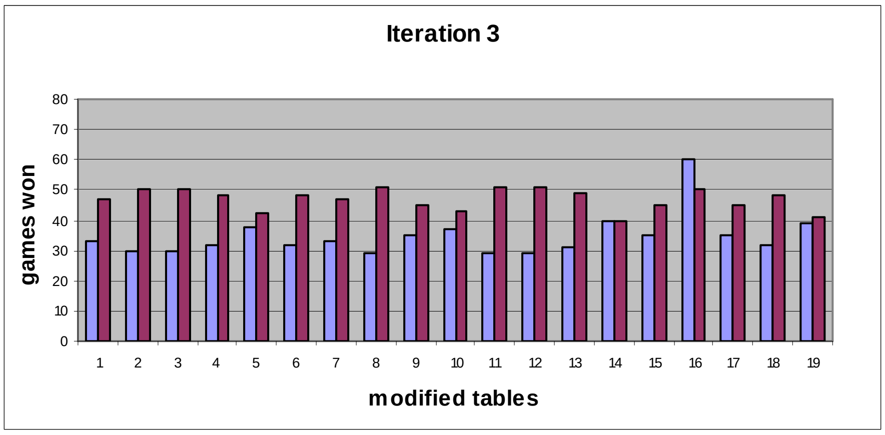


Figure 9-7

Conclusion on experiment 2

From the diagrams it can be seen, that once the best of the initial random tables has been selected AI2 does in fact become better than AI1. In fact, some of the modifications made AI2 win almost 70% of the games in a round.

9.3 Experiment 3

Purpose

This experiment will show how effective the different learning algorithms are when modifying only one table, the WhichUnitIsNeeded table. For the purpose of this experiment AI2 will be trained.

Condition

For all algorithms AI2 will play against an AI (AI1), which has been manually created to be as general as possible. For example, when no units have been constructed by either of the players the probability of building each of the units is the same in the WhichUnitIsNeeded table (0.25, 0.25, 0.25, 0.25).

Initialization

The algorithms RandomIndexAndSign, RandomIndexValueAndSign and the Monte Carlo are initialized with the following values:

GamesToBePlayed: 100

NumberOfModifications: 50

And the simulated annealing is initialized with the values:

GamesToBePlayed: 100

NumberOfRandomTables: 50

Number of iterations: 3

NumberOfTableVariations: 20

The figures

The results of the experiment can be seen in the following figures. Figure 9-8 shows the result of the RandomIndexAndSign algorithm, Figure 9-9 the result of the RandomIndexValueAndSign algorithm and Figure 9-10 the result of the Monte Carlo algorithm. Finally, Figure 9-11 shows the results of the 50 random tables created in the Simulated Annealing algorithm and Figure 9-12 to Figure 9-14

show the results of the 3 iterations of modifications performed in this algorithm.

The 3 first figures shows how many games AI1 and AI2 win each time AI2 is modified. The fourth figure shows how many games AI1 and AI2 win for each of the 50 randomly created AIs (WhichUnitIsNeeded tables) and the 3 last figures shows how many games AI1 and AI2 win for each of the new AIs made by modifying the template AI (the best AI from the previous generation).

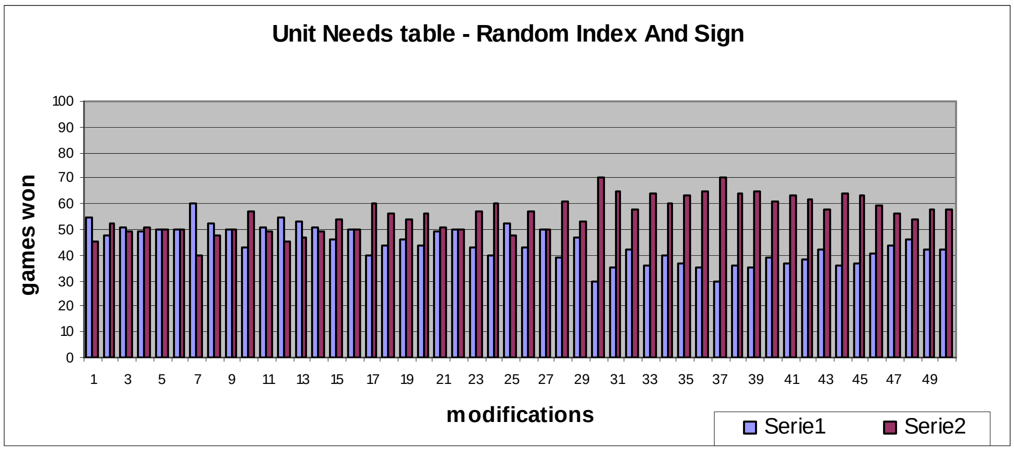


Figure 9-8

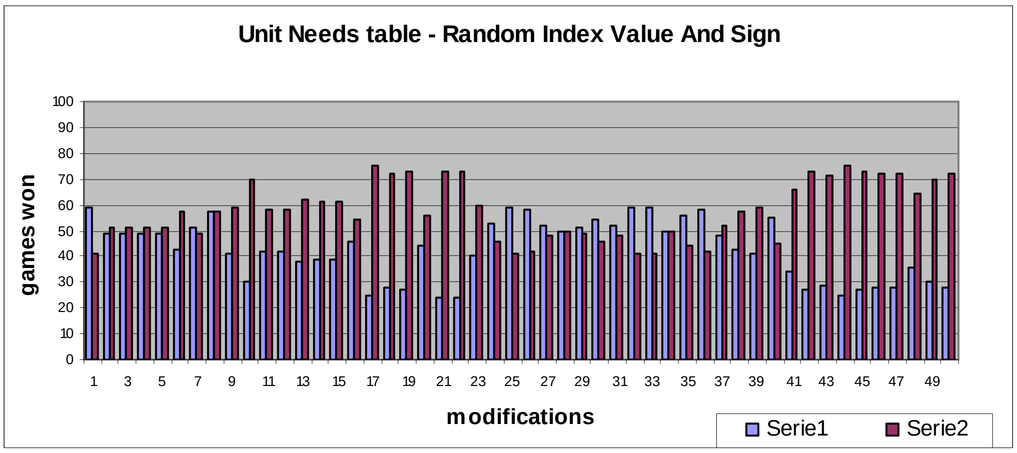


Figure 9-9

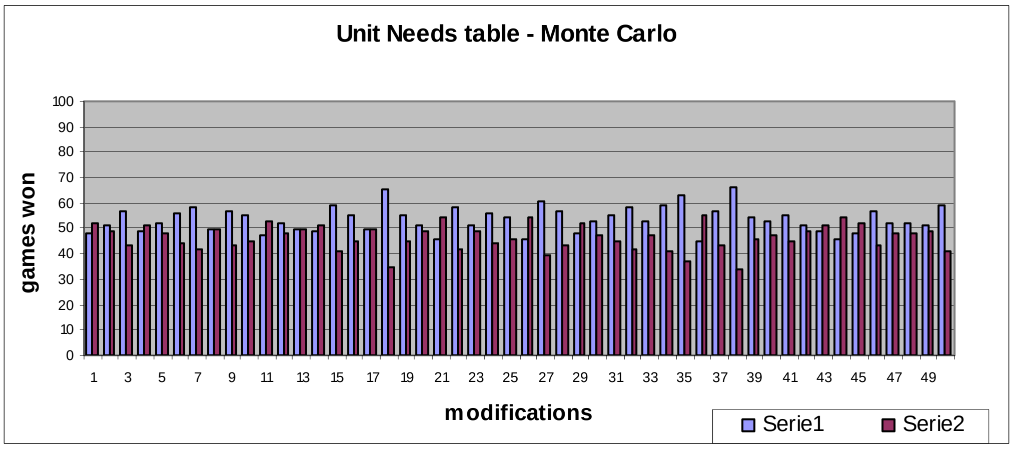


Figure 9-10

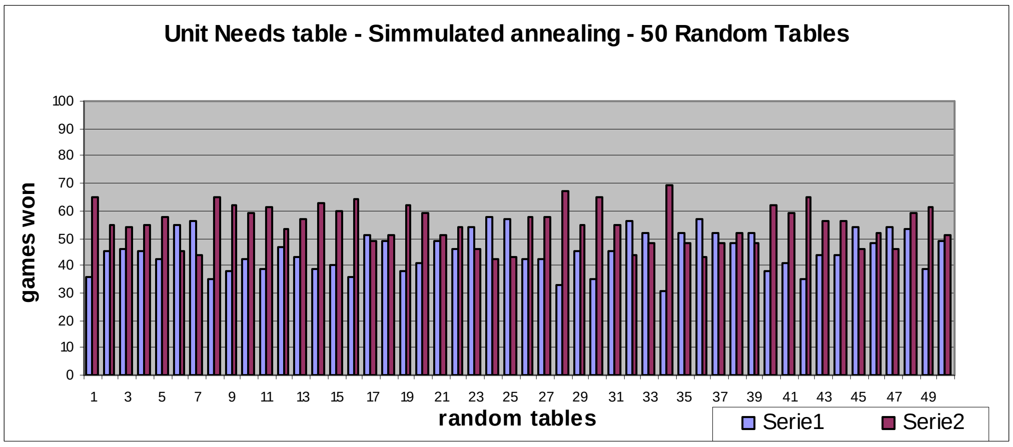


Figure 9-11

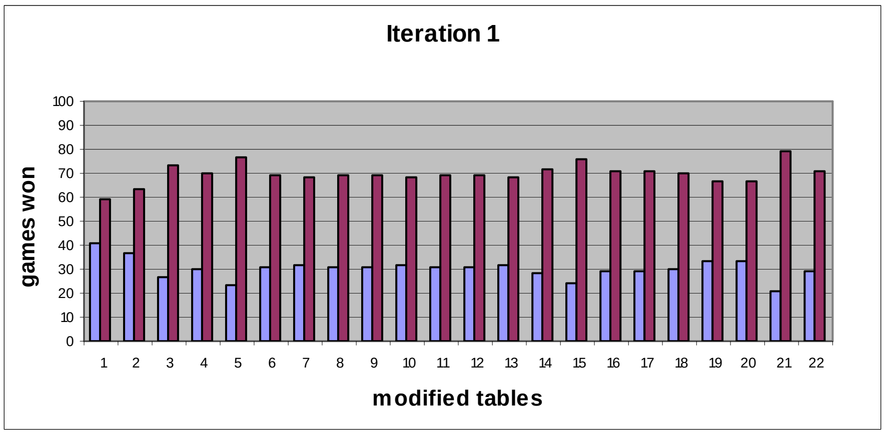


Figure 9-12

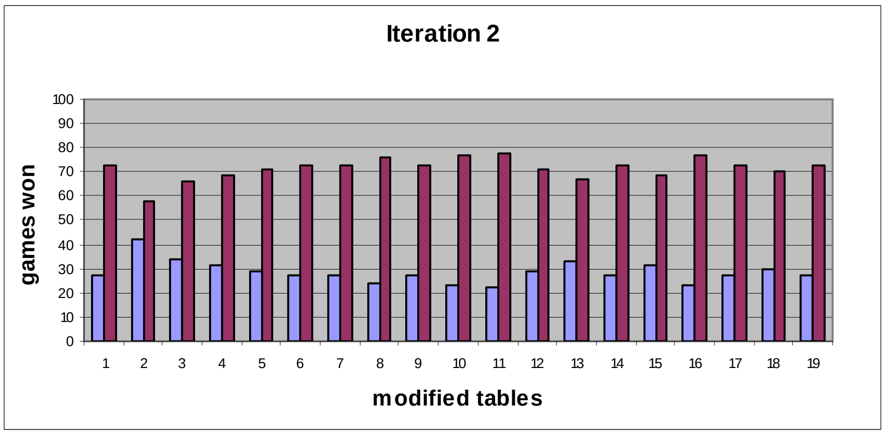


Figure 9-13

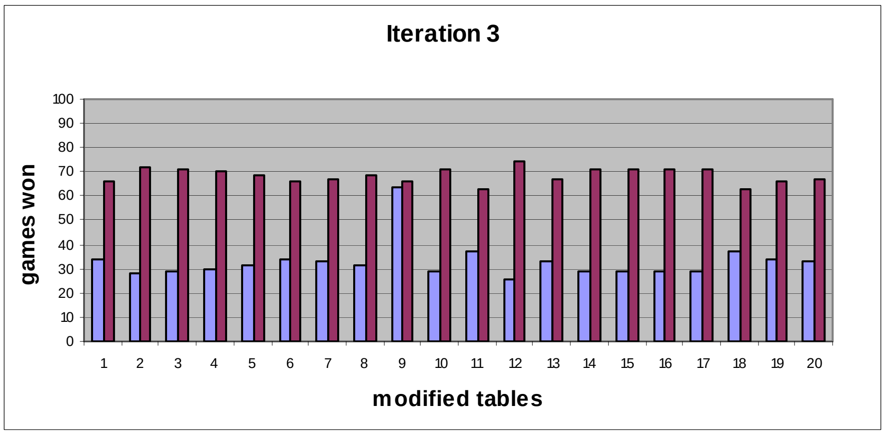


Figure 9-14

Conclusion on experiment 3

First of all this experiment clearly shows, that by modifying the WhichUnitIsNeeded table many different AIs can be created. Some which perform well against an opponent AI and some which do not perform well against this AI.

The results of the two random algorithms RandomIndexAndSign and RandomIndexValueAndSign show how the algorithms are moving randomly around the state space, just as expected (see Figure 9-8 and Figure 9-9). These figures also seem to indicate that there might exist areas in the state space that "fight" especially well against a given opponent AI since the really good performances come in sequences and not just at random.

The learning algorithm performing best was the Simulated Annealing method (see Figure 9-11 to Figure 9-14), which found a table setting making AI2 win 80-20 over its opponent AI1. However, this algorithm is also very expensive and the table setting leading to the 80-20 victory was not found before the 71st round. On the other hand 9 table settings were found after 17 rounds that made the learning AI win approximately 60-40 over the opponent. This is of course not guaranteed to happen every time, but perhaps the costs of the algorithm could be decreased if the algorithm would continue to the modification iterations as soon as a state was found that would make the AI win 60+ games out of 100, instead of searching thoroughly through the state space before moving on to the modifications.

The learning algorithm that shows the least exiting results is the Monte Carlo method (see Figure 9-10). This method did only find 11 table settings that made it better than the opponent, but more often the table modifications made it worse than the opponent. The reason for this could be that this algorithm simply has a slower convergence rate and therefore, the experiment has been done again for the Monte Carlo method only (experiment 3b), but with initialization of the algorithm that makes it search through the state space for a longer time.

9.3.1 Experiment 3b

Purpose

The purpose of this experiment is to show whether the Monte Carlo method will perform better if it is run for a longer period of time.

Condition

Only the WhichUnitIsNeeded table will be modified and this is only done with the Monte Carlo algorithm.

Initialization

The Monte Carlo algorithm is initialized with the following values:

GamesToBePlayed: 100

NumberOfModifications: 100

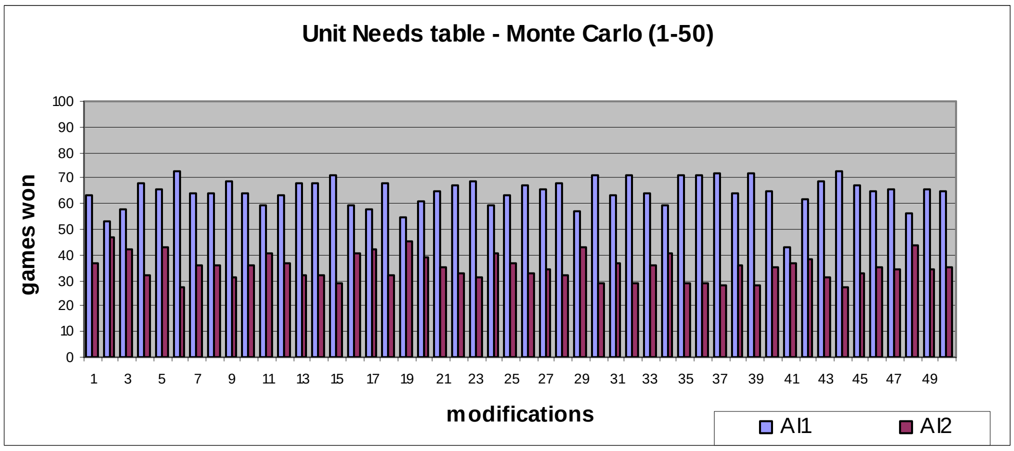


Figure 9-15

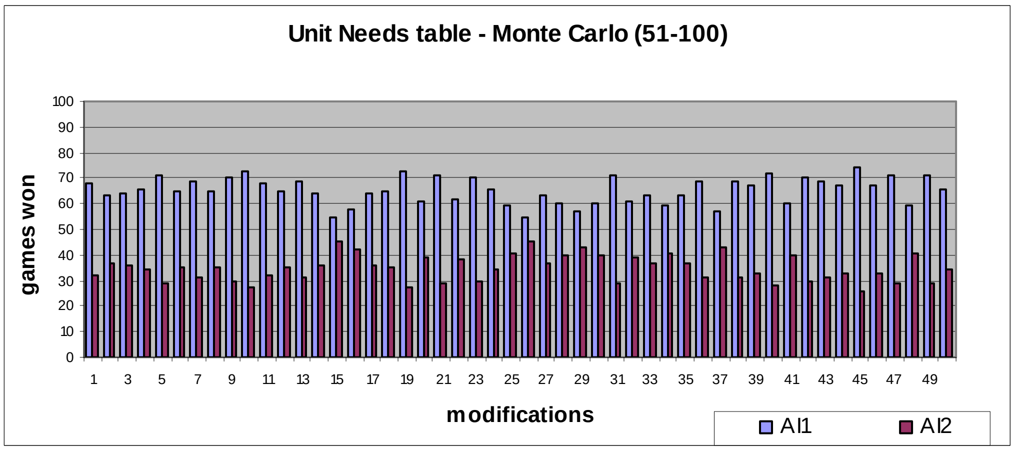


Figure 9-16

Conclusion on experiment 3b

Figure 9-15 and Figure 9-16 show the result of experiment 3b and it can be seen that AI2 (the one being trained) actually performs worse in this experiment though it is run for a longer time. One difference between the two experiments is that the random table produced when initializing the Monte Carlo algorithm is a lot better in experiment 3 than in experiment 3b. This could indicate that the Monte Carlo algorithm does not explore the state space very much but instead stays very close to the initial table through all modifications.

9.4 Experiment 4

Purpose

The purpose of this experiment is to test the different learning algorithms on the table UnitOrWorker. Whereas the WhichUnitIsNeeded table specifies which units are needed, this table decides whether the requested unit should be build or whether a worker unit should be build instead. That is, this table can decide whether the strategy should be a fast attack or whether a strong economy should be established before military units are constructed.

Conditions

As in experiment 3 the AI undergoing training (AI2) will play against an AI that has been manually created before hand.

Initialization

The algrorithms RandomIndexAndSign, RandomIndexValueAndSign and the Monte Carlo are initialized with the following values:

GamesToBePlayed: 100

NumberOfModifications: 50

And the Simulated Annealing is initialized with the values:

GamesToBePlayed: 100

NumberOfRandomTables: 50

Number of iterations: 3

NumberOfTableVariations: 20

The figures

In the following the results from experiment 4 are shown. Figure 9-17, Figure 9-18 and Figure 9-19 show the results of training the UnitOrWorker table with the RandomIndexAndSign algorithm, the RandomIndexValueAndSign algorithm and the Monte Carlo algorithm. Figure 9-20 to Figure 9-23 show the results of training the table with the Simulated Annealing algorithm. For a short description of what the figures show, please see Experiment 3.

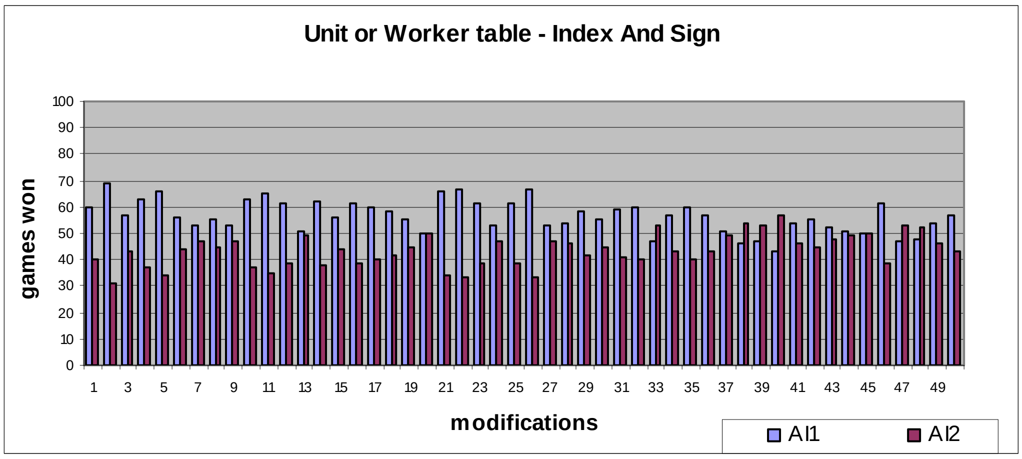


Figure 9-17

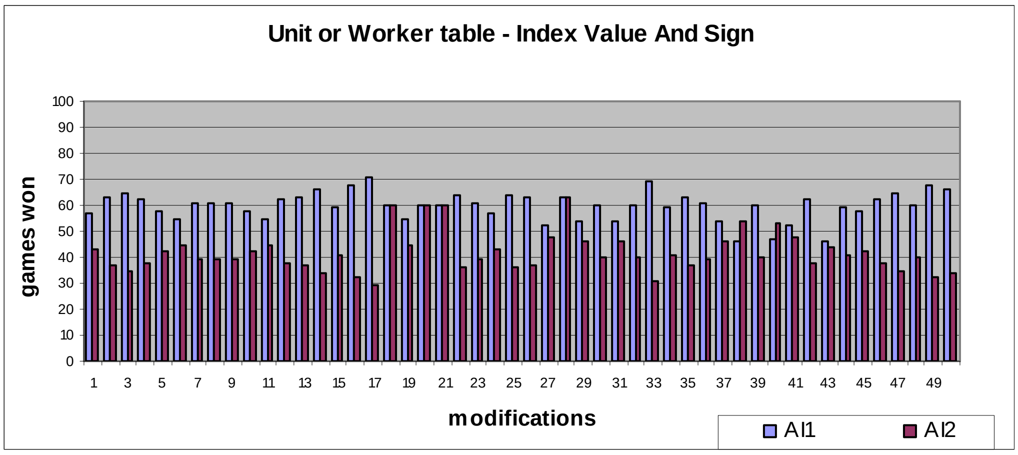


Figure 9-18

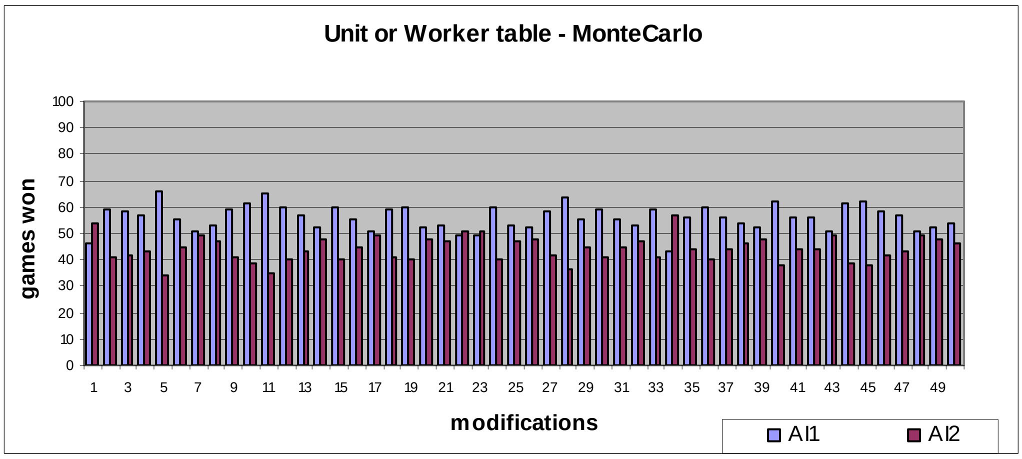


Figure 9-19

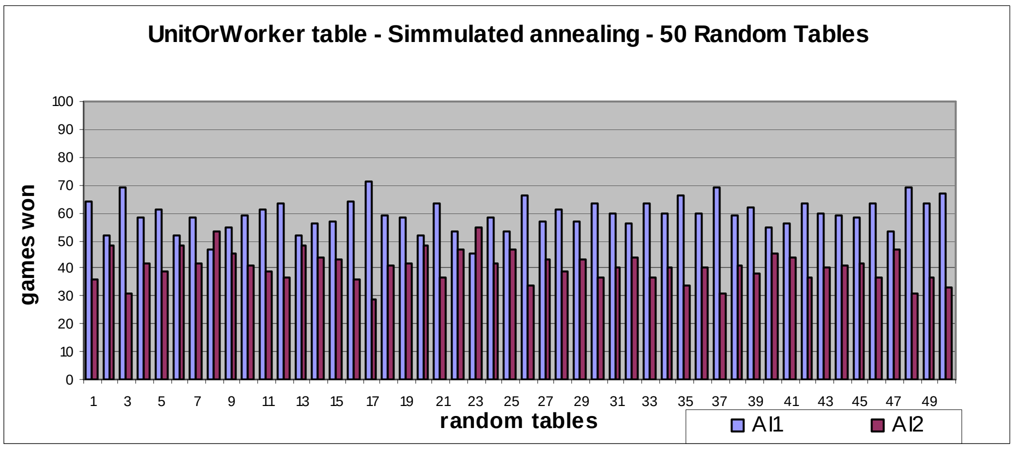


Figure 9-20

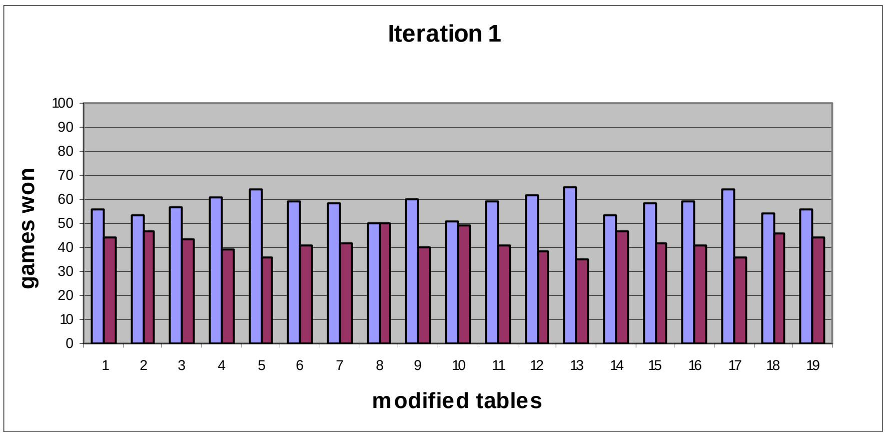


Figure 9-21

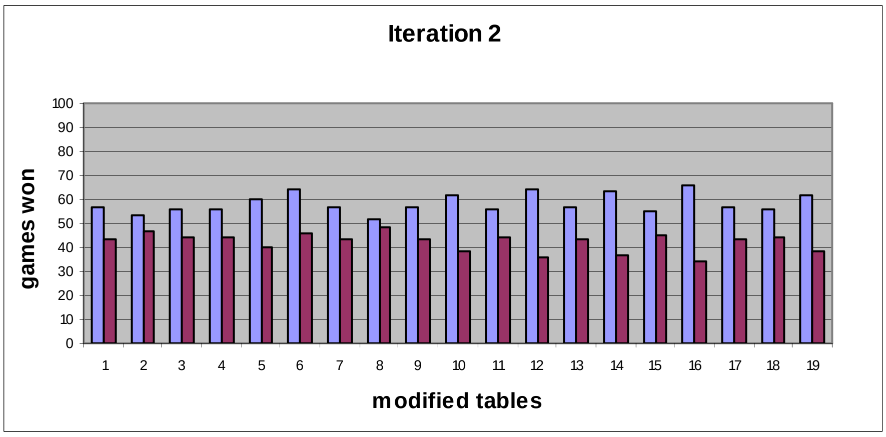


Figure 9-22

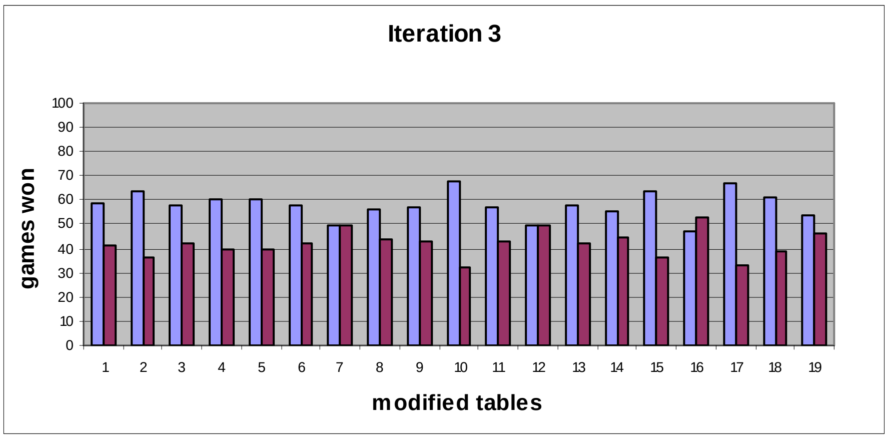


Figure 9-23

Conclusion on experiment 4

Compared to the results of experiment 3 where the WhichUnitIsNeeded table was trained the results of experiment 4 are somewhat disappointing. Neither of the algorithms seems to be capable of finding a table setting for the UnitOrWorker table which makes AI2 clearly better than the opponent. The algorithm that performed best was the RandomIndexAndSign (see Figure 9-17) which after 37 games seemed to have found an area of the state space for the UnitOrWorker table that made the AI perform well. However, even here the results are not as impressive as the results of experiment 3.

The results could indicate that the UnitOrWorker table is a very important table where certain table settings have great negative impact on the outcome of a game. One example that was noticed while observing the two AIs playing during the experiment was that AI2 (the one being trained) at some points was in a state where the probability of building a worker when only having four workers (the number of workers a player starts the game with) was a lot higher than the probability of building military units. This meant that though AI2 got military units before AI1, AI1 quickly became capable of producing more units than AI2 and also to build the units faster than AI2 and therefore often won the game.

Such a table setting also meant, that it did not matter how the rest of the table was modified since the AI almost never got to have more than four workers. Therefore these modifications were not affecting the outcome of the game.

To overcome this problem the algorithms could have been run for a longer period of time or the modification value could have been increased to quickly let the algorithm move away from states that almost certain result in a loss.

However, the Simulated Annealing method, which starts by producing 50 random tables, did not find a table setting either, that would make the AI become clearly better than the opponent. Since it is very unlikely that all these tables should have the problem of almost exclusively building military units, this problem cannot be the only cause for the results.

Another reason could be that the manually created UnitOrWorker table is almost optimal which therefore makes it difficult for AI2 to beat AI1. The following experiment – experiment 4b - is to determine whether this could be the case.

9.4.1 Experiment 4b

Purpose

To determine whether the manually created UnitOrWorker table is close to being optimal which would make it difficult to find better UnitOrWorker tables

Conditions

The experiment is run twice and in both run the algorithm used for modifying the UnitOrWorker table will be the Random IndexAndSign. In the first run, the UnitOrWorker table for AI1 is exchanged with a random UnitOrWorker table and in the second run the UnitOrWorker table for AI1 is set to the best table found when the RandomIndexAndSignalgorithm was run in the original experiment 4. The results of this experiment 4b can be seen in Figure 9-24 and Figure 9-25.

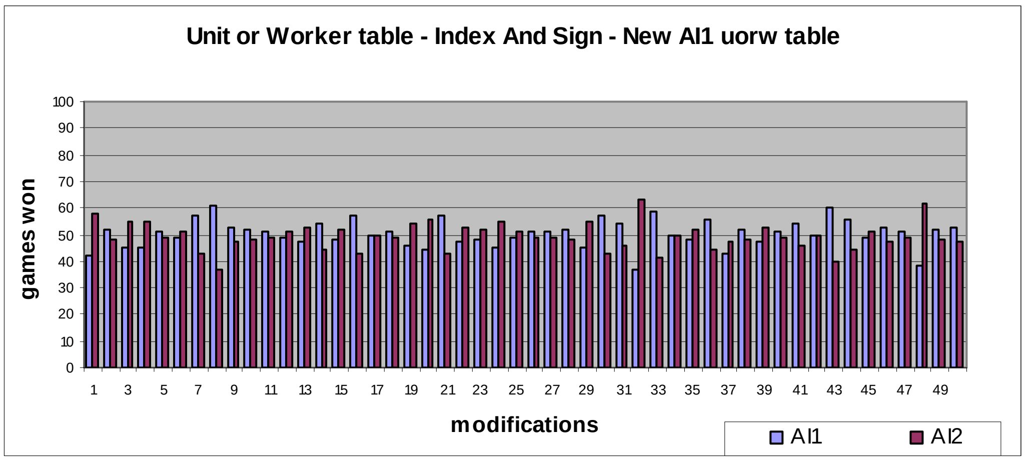


Figure 9-24

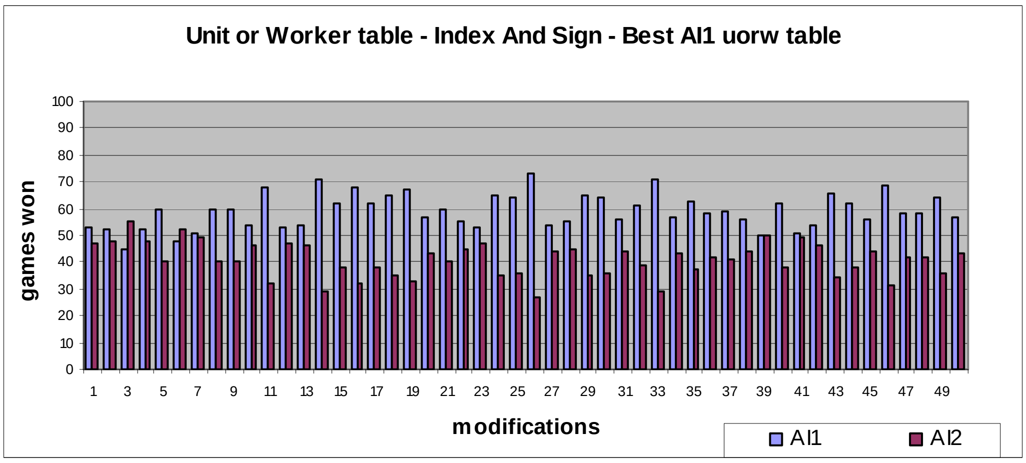


Figure 9-25

Conclusion on experiment 4b

From Figure 9-24 it can be seen that when exchanging the UnitOrWorker table for AI1 with a random UnitOrWorker table, AI2 performs a lot better during the modifications of its UnitOrWorker table. And therefore the manually created UnitOrWorker table might in fact be very difficult to beat.

Further indications of this are found in Figure 9-25. This figure shows the result of running the RandomIndexAndSign algorithm again on AI2 but this time exchanging the UnitOrWorker table in AI1 with the best UnitOrWorker table found by the RandomIndexAndSign algorithm in experiment 4 (the table found after 40 modifications, Figure 9-19).

As can be seen in Figure 9-25 AI1 performs clearly better than AI2 and therefore, the best UnitOrWorker table found by the RandomIndexAndSign algorithm in experiment 4 might actually be stronger than illustrated by Figure 9-19.

9.5 Experiment 5

Purpose

The purpose of this experiment is to test the different learning algorithms on the table SoldierOrBarrack (see section 7.2). When a unit request is being made by the WhichUnitIsNeeded table and the BuildWorkerOrUnit table has decided to build a military unit instead of a worker, the SoldierOrBarrack tables decides whether the requested unit should actually be produced or whether a building capable of producing this unit should be build instead. Depending on the setting of this table the AI can either prefer to build only one building and then start to produce units, or it can choose to build a number of buildings, which will decrease the time it takes to produce a unit but which on the other hand will give the opponent the opportunity to get a large army while the buildings are being produced.

Conditions

Only the SoldierOrBarrack will undergo training. This is done by all four learning algorithms.

Initialization

The algorithms are initialized as in the previously experiments:

The algorithms RandomIndexAndSign, RandomIndexValueAndSign and the Monte Carlo are initialized with the following values:

GamesToBePlayed: 100

NumberOfModifications: 50

The Simulated Annealing is initialized with the values:

GamesToBePlayed: 100

NumberOfRandomTables: 50

Number of iterations: 3

NumberOfTableVariations: 20

The figures

The results of the experiment can be seen in the following figures. Figure 9-26 shows the result of the RandomIndexAndSign algorithm, Figure 9-27 the result of the RandomIndexValueAndSign algorithm and Figure 9-28 the result of the Monte Carlo algorithm. Finally, Figure 9-29 shows the results of the 50 random tables created in the Simulated Annealing algorithm and Figure 9-30 to Figure 9-32 show the results of the 3 iterations of modifications performed in this algorithm. The content of the figures is as previously described in section 9.3.

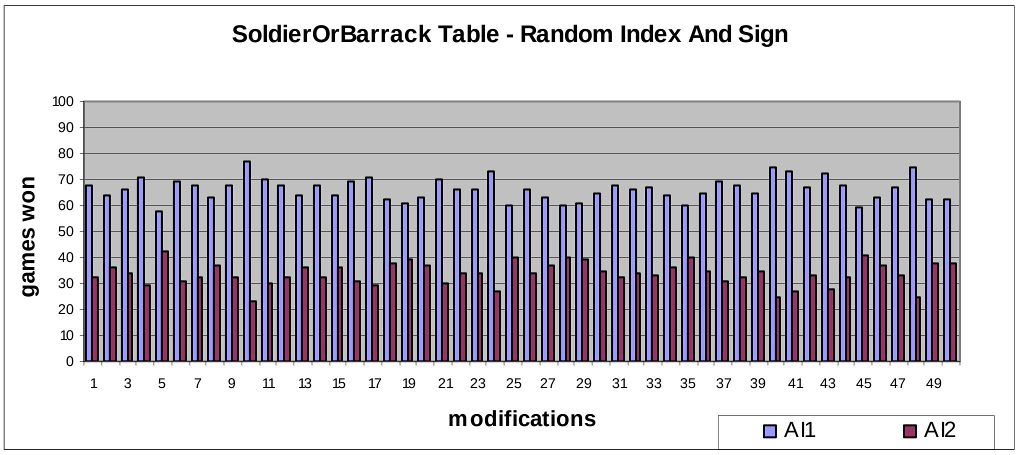


Figure 9-26

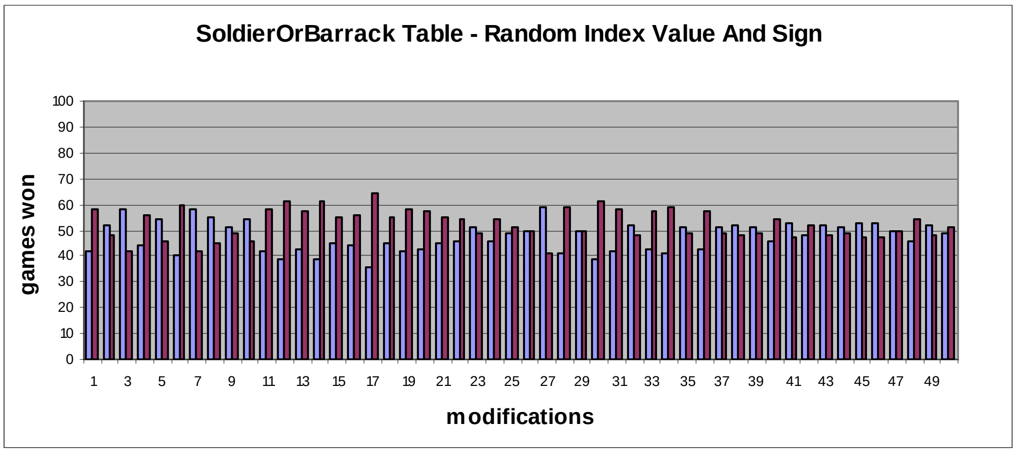


Figure 9-27

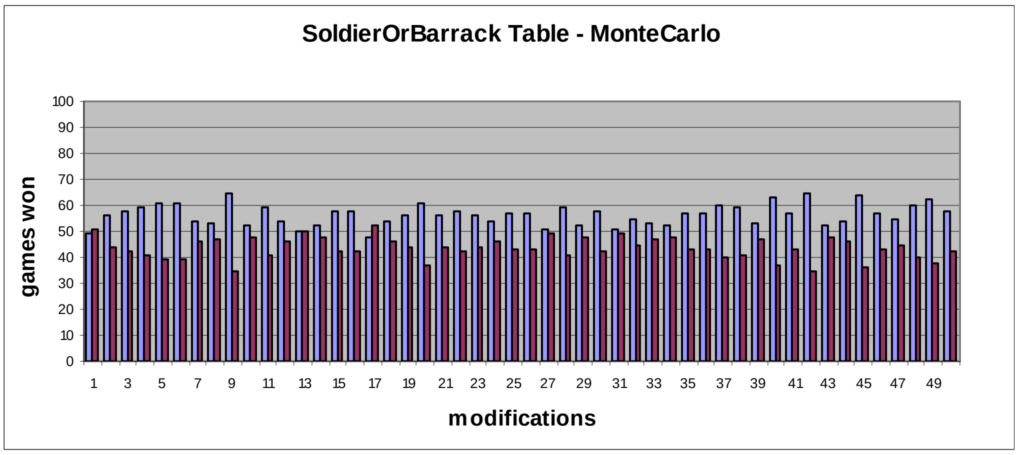


Figure 9-28

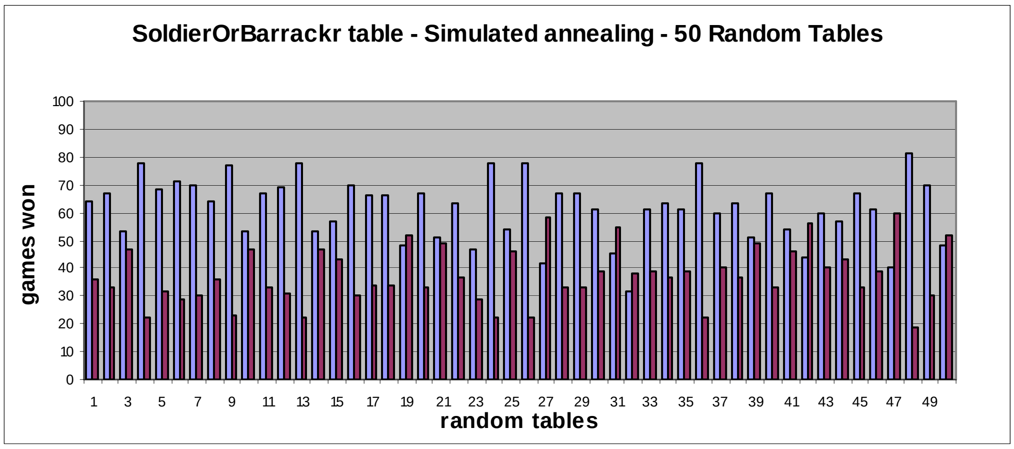


Figure 9-29

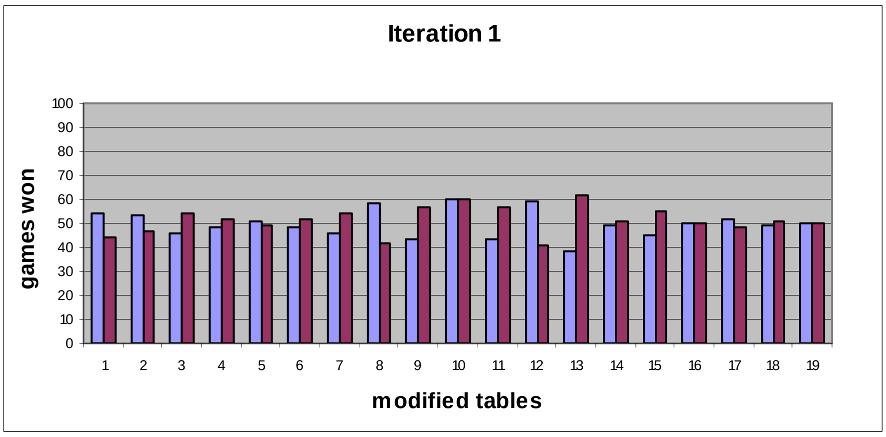


Figure 9-30

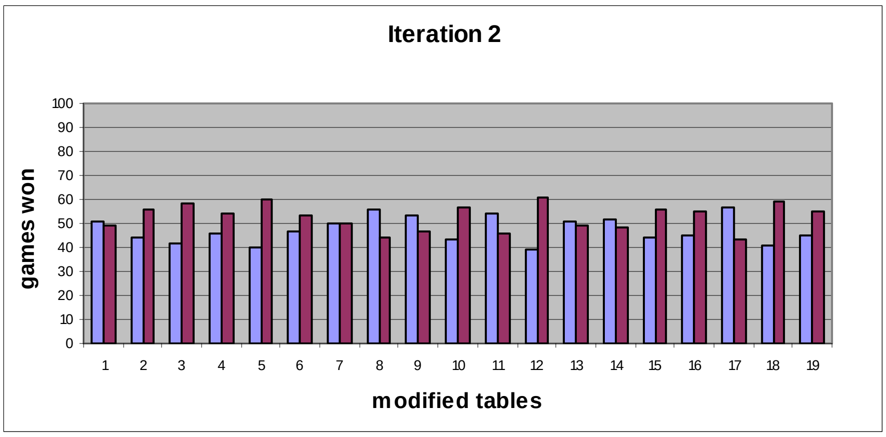


Figure 9-31

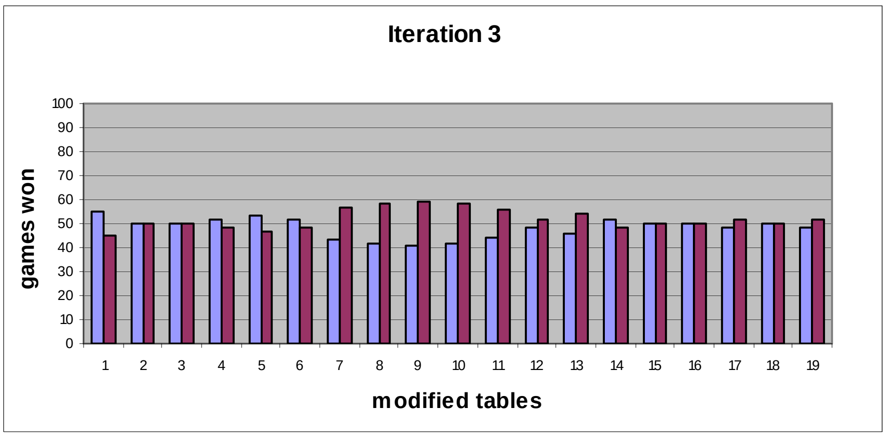


Figure 9-32

Conclusion on experiment 5

Figure 9-26 clearly shows that searching through the state space at complete random does not necessarily result in a good AI. But it does seem strange that whereas the RandomIndexAndSign algorithm performed ok in experiment 3 and 4 it performs really poorly in this experiment. One reason for this could be that the algorithm was not working correct. However, the resulting tables from the algorithm have been examined and they do have a proper probability distribution which furthermore is different than the probability distribution in the random table made when initializing the algorithm. This means that the algorithm *is* modifying the table without violating any constraint put on the table.

Another reason can be found by looking at the first figure for the Simulated Annealing algorithm (Figure 9-29). This figure shows how well different AIs based on randomly created SoldierOrBarrack tables perform. From this figure it can be seen, that the main part of the AIs performs really poorly, which gives an indication that only a small area of the state space contains SoldierOrBarrack tables that are good against the manually made AI1. This would also explain why the RandomIndexValueAndSign algorithm performed a lot better than the RandomIndexAndSign algorithm, since this algorithm starts by creating a random SoldierOrBarrack table that actually makes the AI perform really well.

If this is the reason, then the RandomIndexAndSign algorithm could probably be improved by either letting it run for a longer period of time, or by adjusting the value used for modifying the SoldierOrBarrack table.

The RandomIndexValueAndSign algorithm (Figure 9-27) seems to perform ok. But as mentioned this could be because its starting SoldierOrBarrack table already makes it perform well. To test whether this algorithm actually is good at modifying the SoldierOrBarrack table the experiment could be repeated and the algorithm initialized with a SoldierOrBarrack table making it perform poorly to begin with. However, due to time limits cannot be done in this thesis.

The Monte Carlo method generally results in poor AIs, but the result from this algorithm is still better than the RandomIndexAndSign. Actually, the Monte Carlo method seems to show a tendency, namely that only small variations are found between the different AIs (modified tables) and that the AI always seem to be just a little better or a little worse than the opponent.

9.6 Experiment 6

Purpose

The purpose of this experiment is to test the different learning algorithms on the table TankOrFactory (see section 7.2). Just like the SoldierOrBarrack table decides whether to build soldiers or barracks in a given situation this table decides whether tanks or a factory (or more) should be build.

Conditions

Only the TankOrFactory table will undergo training. This is done by all four learning algorithms.

Initialization

The algorithms are initialized as in the previously experiments:

The algorithms RandomIndexAndSign, RandomIndexValueAndSign and the Monte Carlo are initialized with the following values:

GamesToBePlayed: 100

NumberOfModifications: 50

The Simulated Annealing is initialized with the values:

GamesToBePlayed: 100

NumberOfRandomTables: 50

Number of iterations: 3

NumberOfTableVariations: 20

The figures

The figures will be presented in the same order as it was done in the previous experiments, as will the information they show – now just reflecting the training of the TankOrFactory table. Therefore no further explanation of the figures is given here.

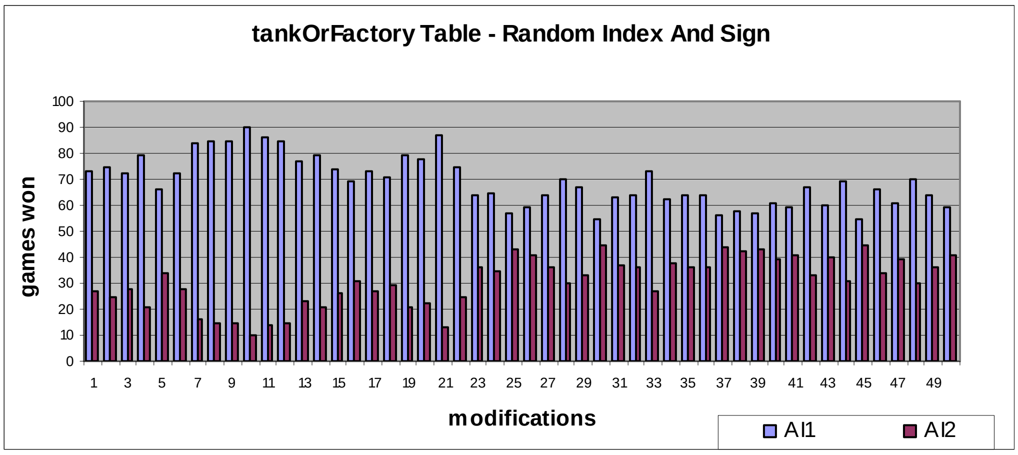


Figure 9-33

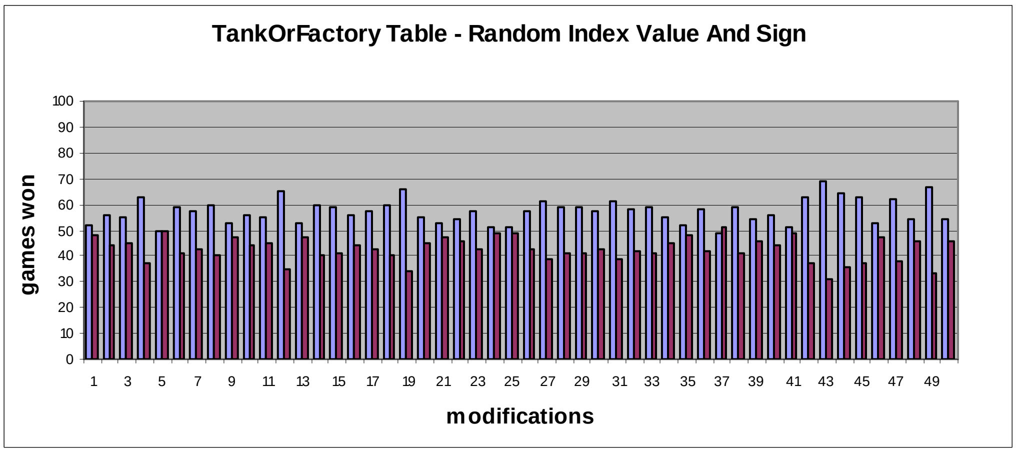


Figure 9-34

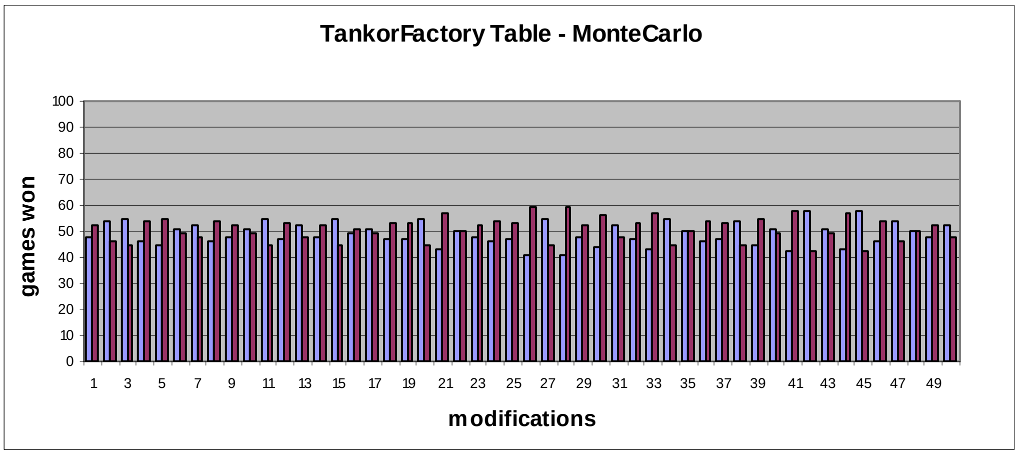


Figure 9-35

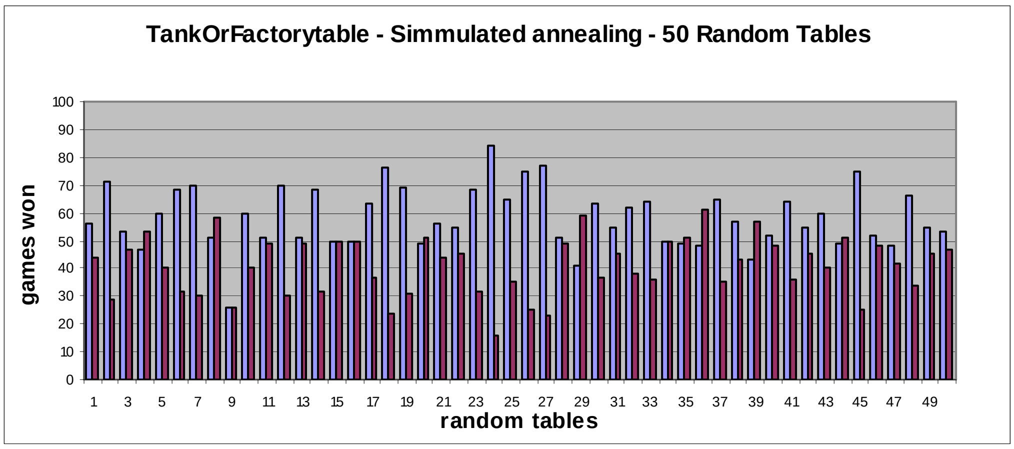


Figure 9-36

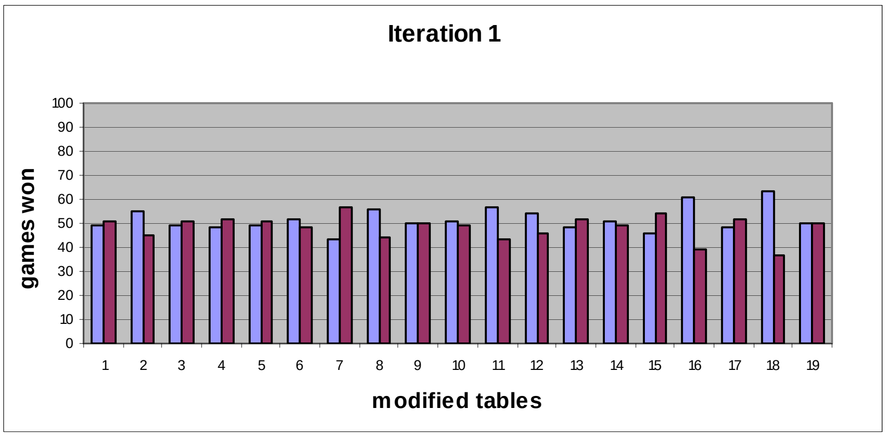


Figure 9-37

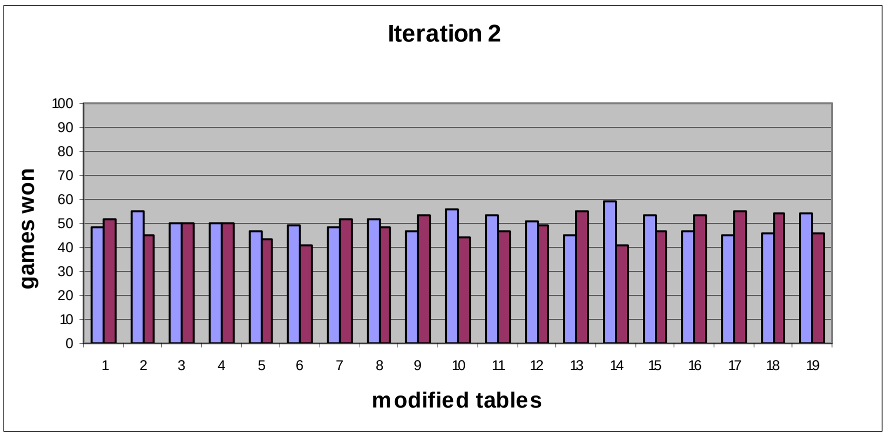


Figure 9-38

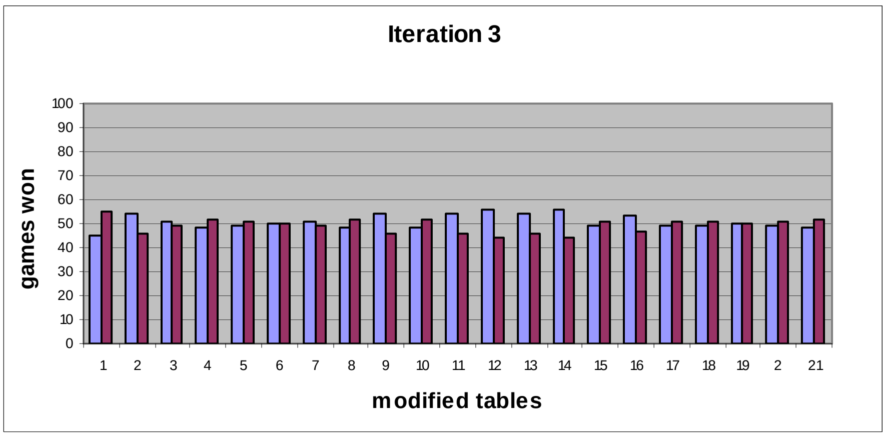


Figure 9-39

Conclusion on experiment 6

The results from this experiment are almost identical with the results from experiment 5. The most interesting difference is that the resulting AIs from the Monte Carlo algorithm win 27 out of the 50 rounds and get a draw in two of them (see Figure 9-35) and this actually makes the Monte Carlo method perform better than the other three algorithms.

The 50 randomly created TankOrFactory tables produced by the Simulated Annealing method (see Figure 9-36) show that there are very big differences in how well the different tables make the AI perform. This was also seen in experiment 5 and could explain why the RandomIndexAndSign algorithm performs worse than the RandomIndexValueAndSign algorithm, since the randomly created tables the two algorithm start with perform very different (Figure 9-33 and Figure 9-34).

9.7 Experiment 7

Purpose and condition

The purpose of this experiment is to see how well the four algorithms perform when they are modifying all four tables at the same time.

One thing could indicate that this in the long run could result on stronger AIs. For example, building military units when having only five or six workers means that money cannot be wasted on producing many barracks or factories but should instead be used on getting the actual units (so only one barrack or factory should be build). This strategy would involve the UnitOrWorker table and either the SoldierOrBarrack or TankOrFactory depending on whether soldiers or factories were preferred. The UnitOrFactory table should make sure that only few workers were built and the SoldierOrBarrack / TankOrFactory should make sure that units and not building were actually produced.

In the previous experiments only one table has been trained at a time. This could mean that though a given table setting actually would be optimal for a given strategy, the AI in fact performed very poorly because the other tables were not set for this strategy as well.

Ideally all combination of tables ought to have been trained together to find the combination of tables that gave the best results. However, due to time limitations these experiments cannot be performed in this thesis. Instead this experiment will, as already mentioned, let the algorithms train all four tables at once.

Training only one table at a time as it was done in previous experiments showed that the state space for some of the tables were very large, or at least that the algorithms needed a long time to find good table settings. When training four tables at a time this state space of course gets a lot bigger and therefore the algorithms most likely need to run for even longer time. For this experiment the algorithms will therefore run twice as long as they have done in previous experiments.

Initialization

The algorithms are therefore initialized with the following values:

RandomIndexAndSign, RandomIndexValueAndSign and Monte Carlo:

GamesToBePlayed: 100

NumberOfModifications: 100

The Simulated Annealing algorithm:

GamesToBePlayed: 100

NumberOfRandomTables: 100

Number of iterations: 3

NumberOfTableVariations: 20

The figures

Figure 9-40 shows the results of the 50 first modifications of all tables by the RandomIndexAndSign algorithm. Figure 9-41 shows the results of the last 50 modifications.

Figure 9-42 shows the results of the 50 first modifications of all tables by the RandomIndexValueAndSign algorithm. Figure 9-43 shows the results of the last 50 modifications.

Figure 9-44 shows the results of the 50 first modifications of all tables by the Monte Carlo algorithm. Figure 9-45 shows the results of the results last 50 modifications.

Figure 9-46 shows the results of the 50 first randomly created tables by the Simulated Annealing algorithm. Figure 9-47 shows the results of the last 50 randomly created tables.

Figure 9-48 to Figure 9-50 shows the results of the modifications of all tables in the three iterations of the algorithm.

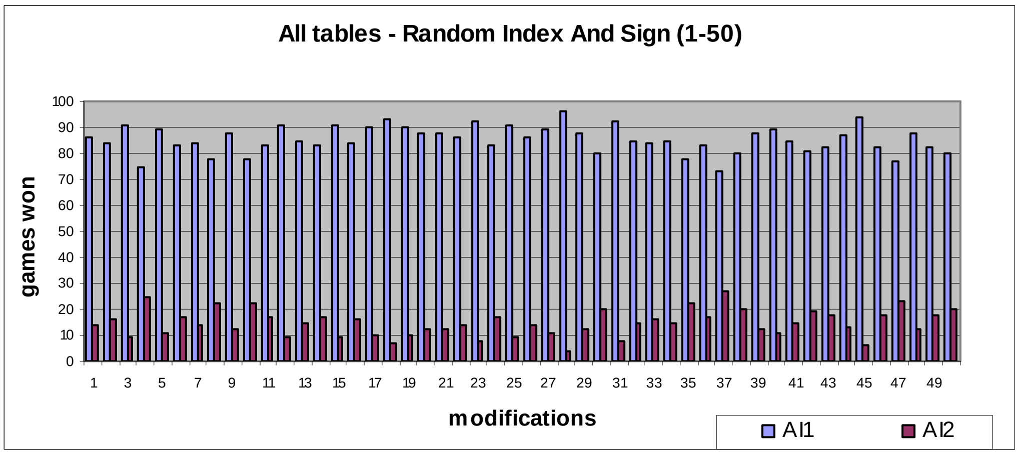


Figure 9-40

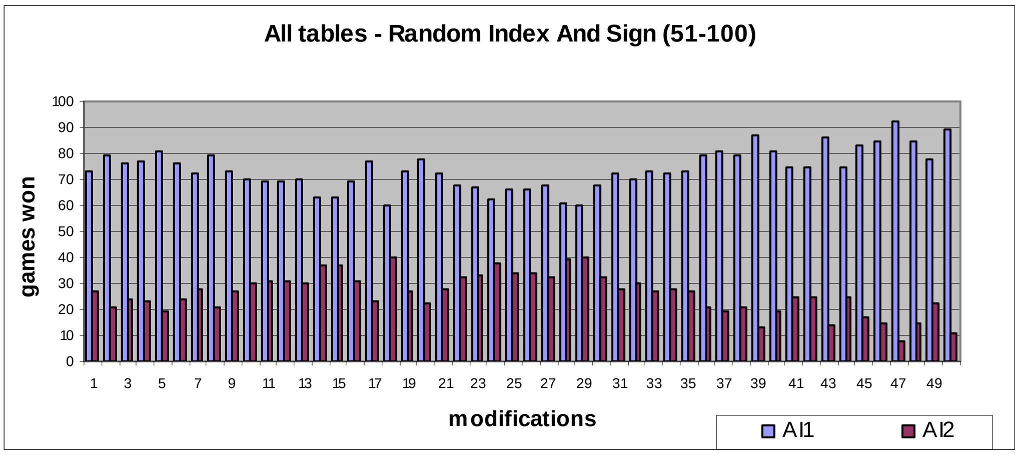


Figure 9-41

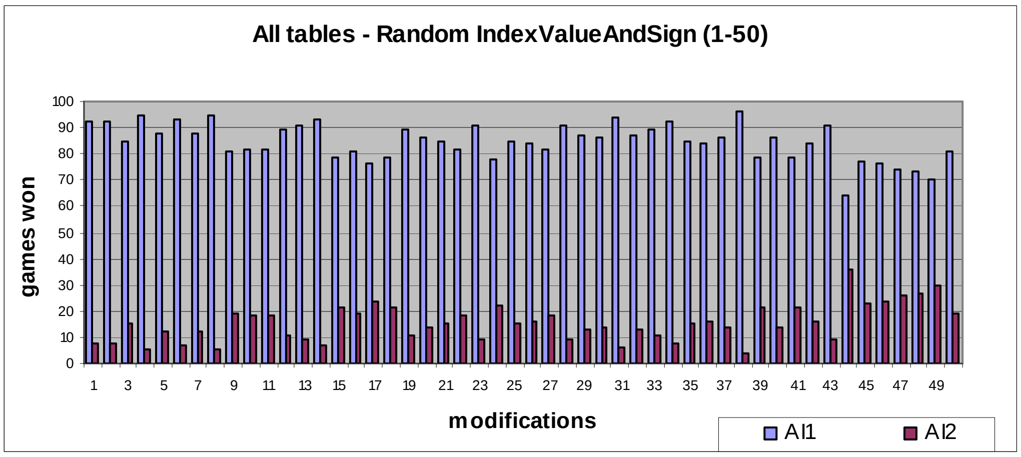


Figure 9-42

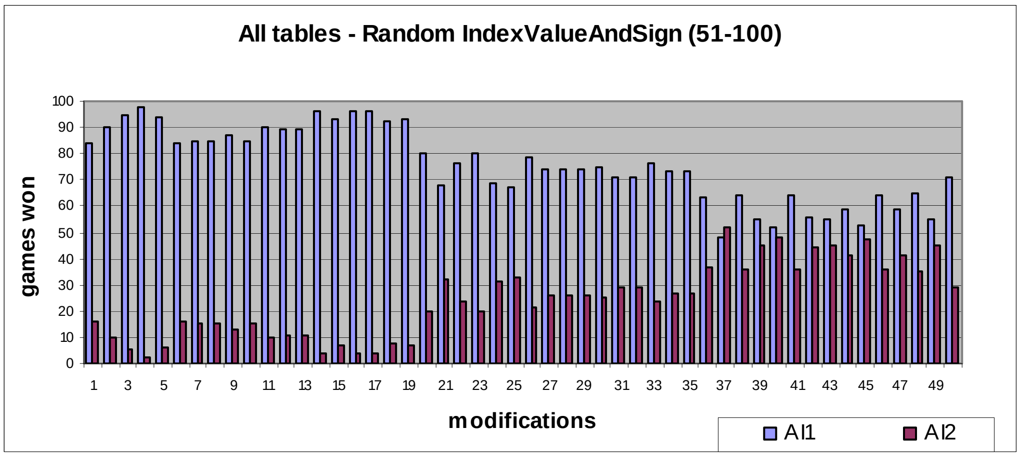


Figure 9-43

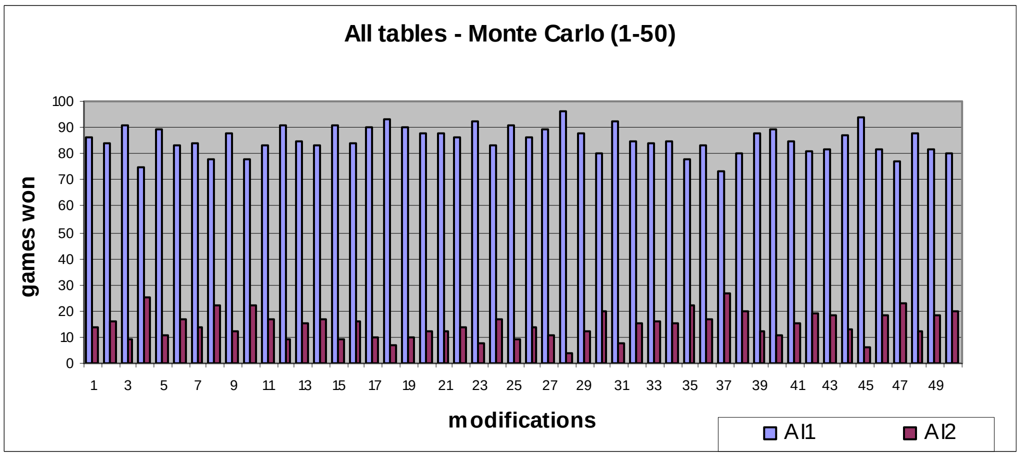


Figure 9-44

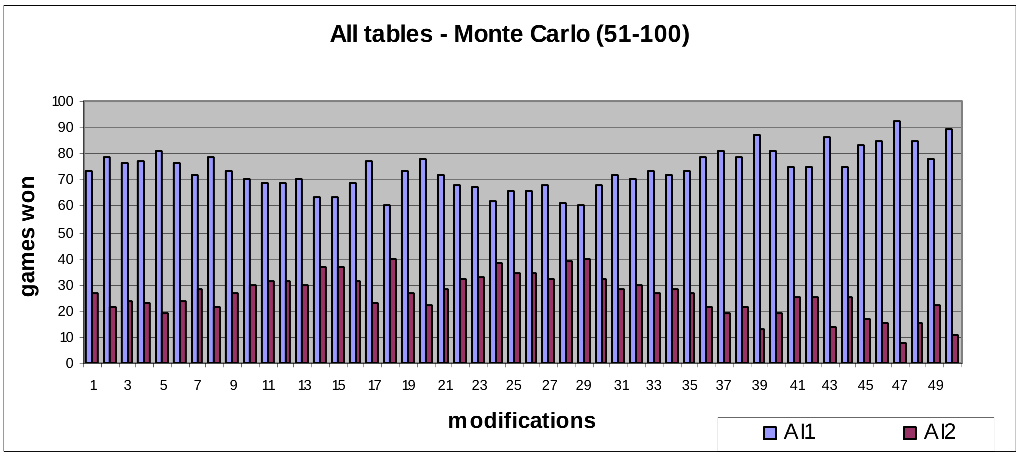


Figure 9-45

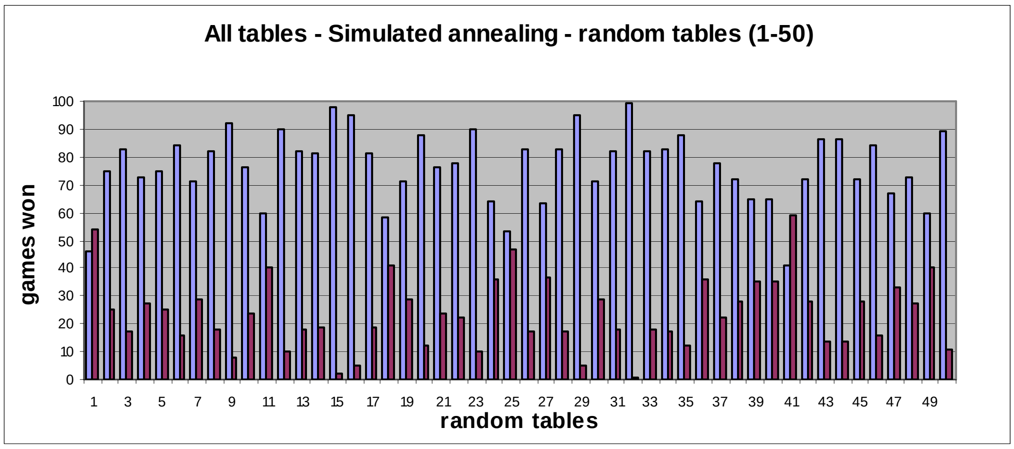


Figure 9-46

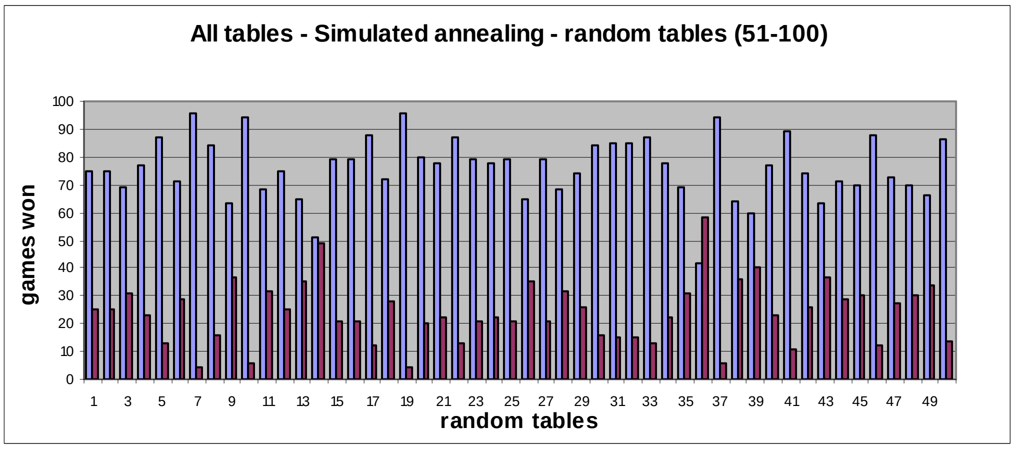


Figure 9-47

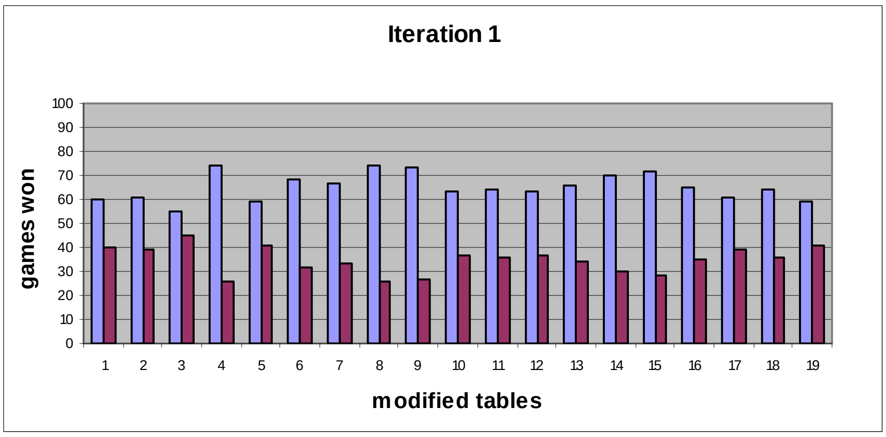


Figure 9-48

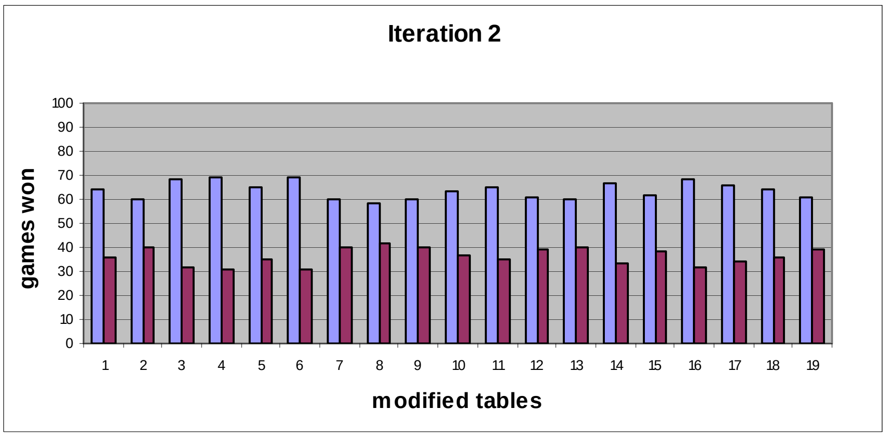


Figure 9-49

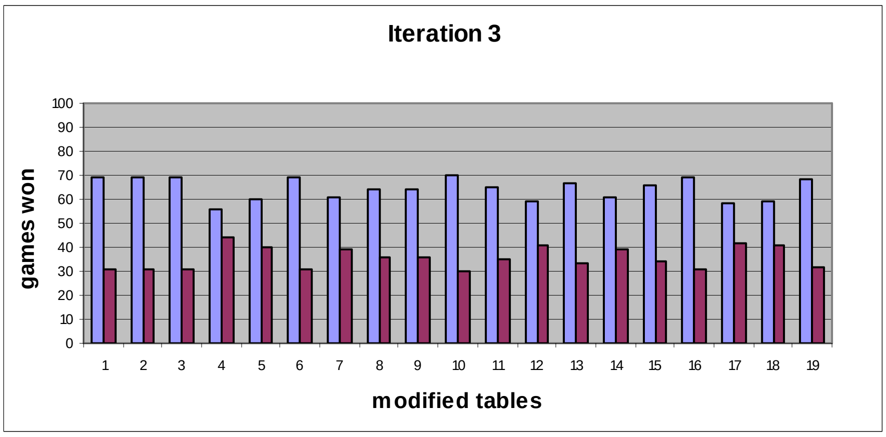


Figure 9-50

Conclusion on experiment 7

This experiment clearly shows that when the state space becomes larger all algorithms have difficulties finding table settings which make the AI perform well. This is clearly shown in the Simulated Annealing method (Figure 9-46 and Figure 9-47) where only very few of the 100 randomly created tables makes the AI perform well. The algorithm that seems to have found the best area of the state space is the Random IndexValueAndSign algorithm (Figure 9-42 and Figure 9-43), which after 71 modifications finds several states which all make the AI win between 30 and 50 of the 100 games played between the two AIs.

Therefore, though the idea of training all tables at the same time was that the tables in this way would end up sharing the same strategy and thereby would be able to make the AI become really strong, the state space simply seems to become to big for the algorithms to be able to find these tables within the 100 modifications which were the number of modifications chosen for this experiment.

9.8 Experiment 8

Purpose and condition

This experiment will summarize the results of the previous experiments. This is done by letting an AI, made up of the best tables found by each algorithm in experiment 3, 4, 5 and 6, play against the manually created AI for 200 games. That is, the best WhichUnitIsNeeded table, the UnitOrWorker table, the best SoldierOrBarrack table and the best TankOrFactory table found by each of the four algorithms will be combined into 4 AIs which are then set to play against the manually created AI.

This will show which learning algorithm was best at what they were actually developed to do, namely to make the AI capable of adapting it self to new strategies.

The experiment will also show whether training each table at a time is a better solution than training all the tables at the same time. Therefore four additional AIs are made from the best tables found by the four algorithms when training all four tables at once (Experiment 7).

Figures

Figure 9-51 shows the results of the AI made of the four best tables found by the RandomIndexAndSign algorithm in experiment 3, 4, 5 and 6 (when training one table at a time) when playing against the manually created AI.

Figure 9-52 shows the results of the AI made of the four best tables found by the RandomIndexValueAndSign algorithm in experiment 3, 4, 5 and 6 (when training one table at a time) when playing against the manually created AI.

Figure 9-53 shows the results of the AI made of the four best tables found by the Monte Carlo algorithm in experiment 3, 4, 5 and 6 (when training one table at a time) when playing against the manually created AI.

Figure 9-54 shows the results of the AI made of the four best tables found by the Simulated Annealing algorithm in experiment 3, 4, 5 and 6 (when training one table at a time) when playing against the manually created AI.

Figure 9-55 shows the results of the AI made of the four best tables found by the RandomIndexAndSign algorithm in experiment 7 (when training the four tables at the same time) when playing against the manually created AI.

Figure 9-56 shows the results of the AI made of the four best tables found by the RandomIndexValueAndSign algorithm in experiment 7 (when training the four tables at the same time) when playing against the manually created AI.

Figure 9-57 shows the results of the AI made of the four best tables found by the Monte Carlo algorithm in experiment 7 (when training the four tables at the same time) when playing against the manually create AI

Figure 9-58 shows the results of the AI made of the four best tables found by the Simulated Annealing algorithm in experiment 7 (when training the four tables at the same time) when playing against the manually created AI.

The content of the figures shows the intermediate score of the 200 games played by the two AIs.

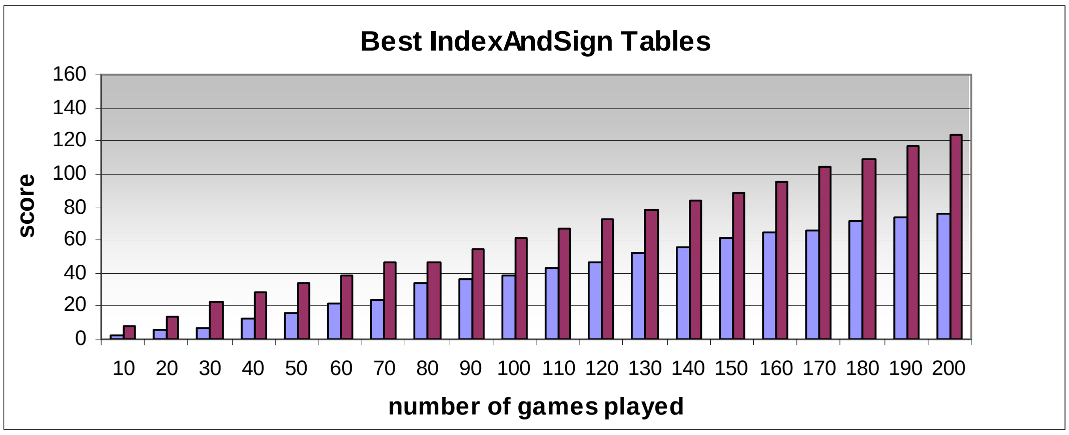


Figure 9-51

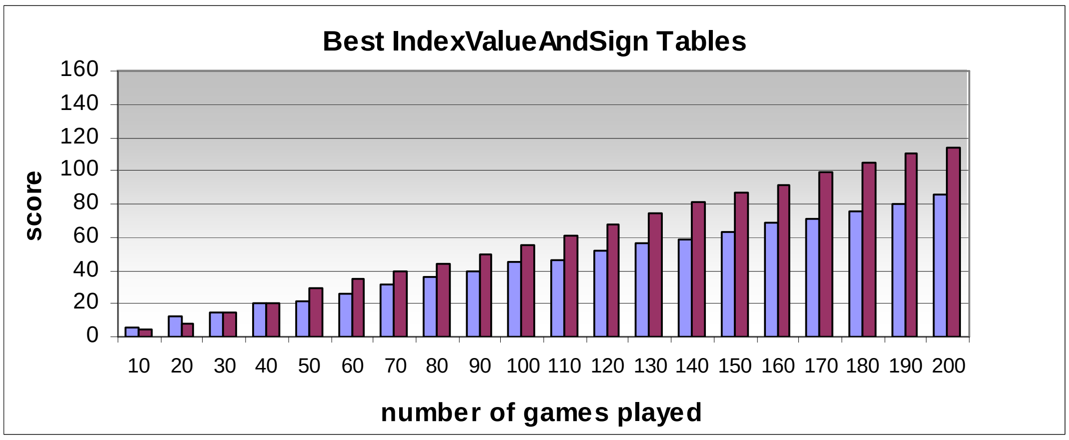


Figure 9-52

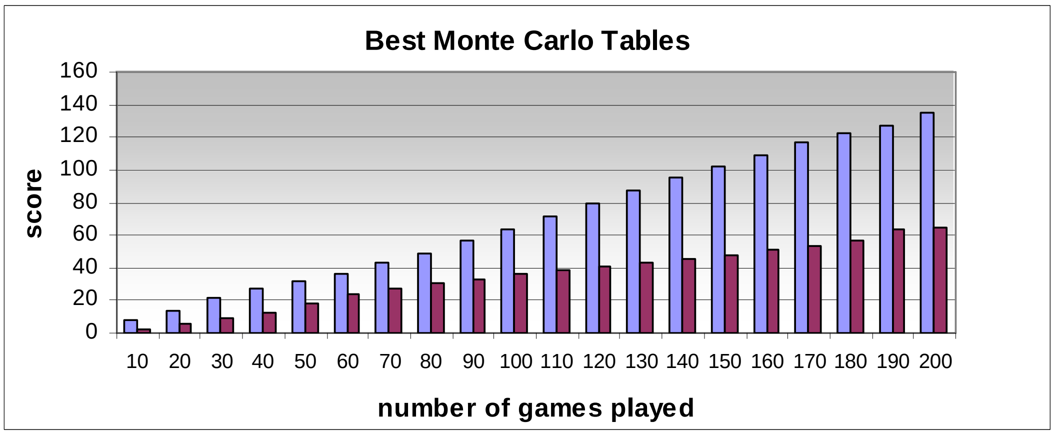


Figure 9-53

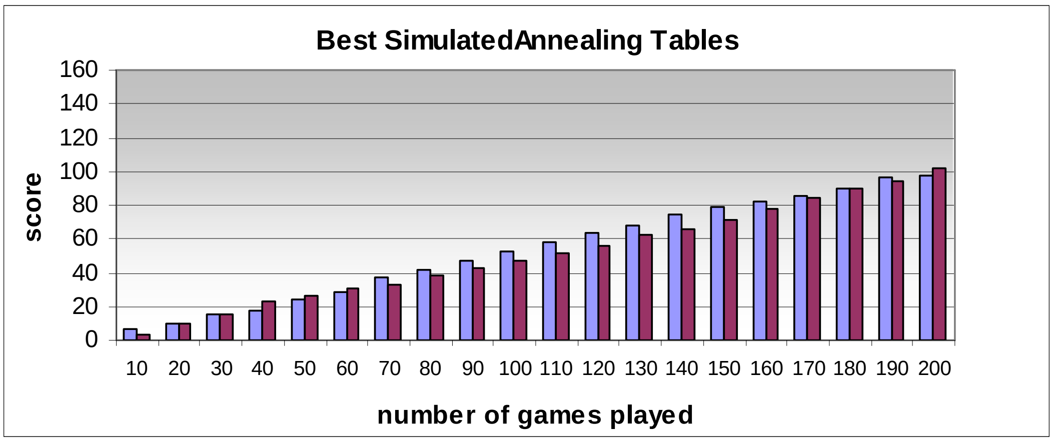


Figure 9-54

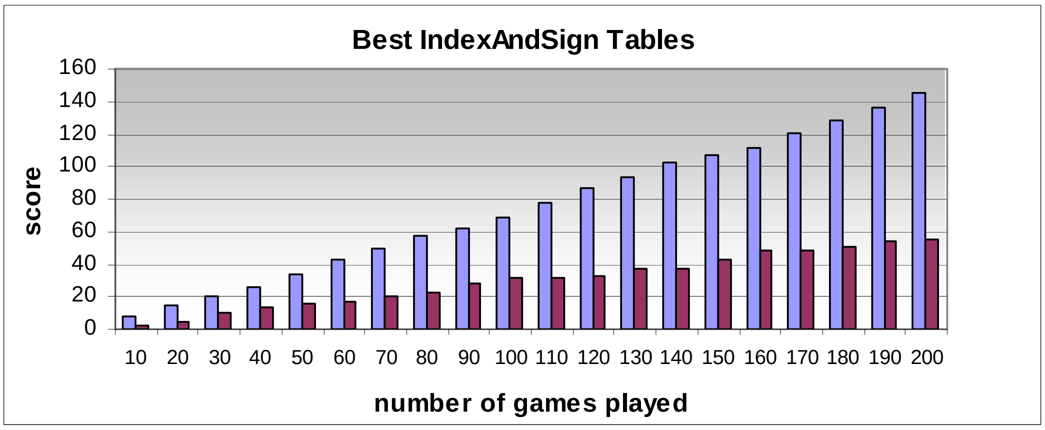


Figure 9-55

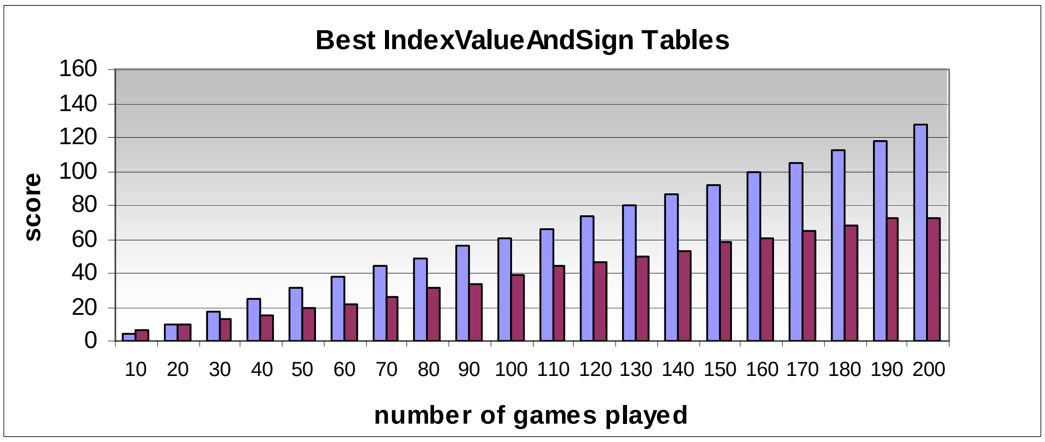


Figure 9-56

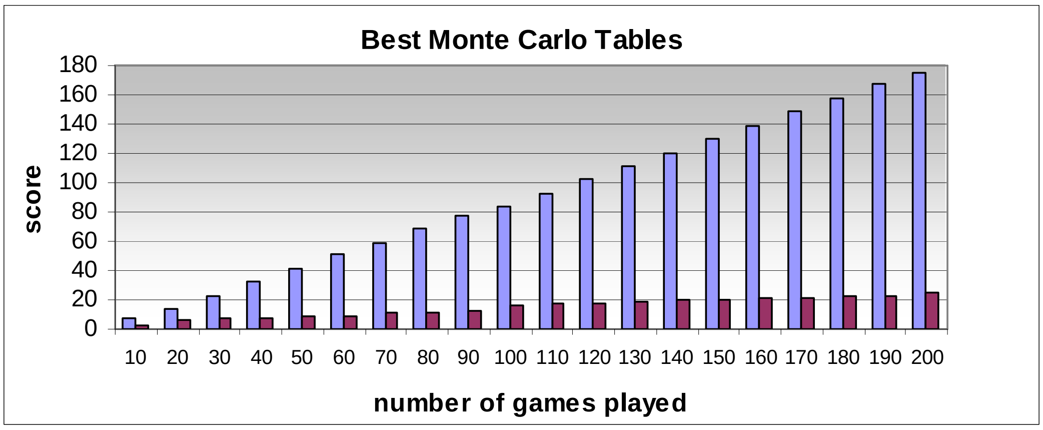


Figure 9-57

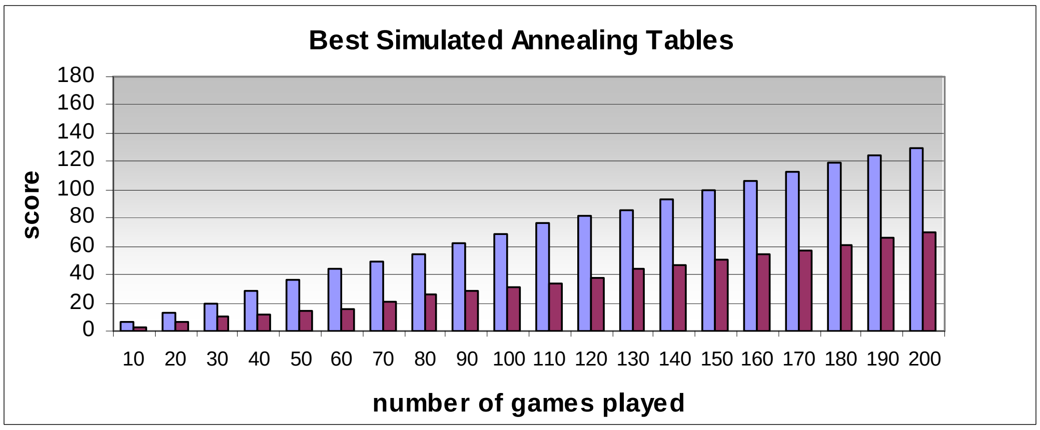


Figure 9-58

Conclusion on experiment 8

From the figures it can be seen that the algorithm that produced the best tables were the Random IndexAndSign and the Random IndexValueAndSign algorithms when they were training one table at a time (Figure 9-51 and Figure 9-52). The AI build up of these tables clearly are better than the manually created AI. This can seem a little strange due to the results of experiment 5 and 6 where especially the Random IndexAndSign had some difficulties finding good tables. One explanation could be that the four tables found by each of the two algorithms actually resulted in a very strong strategy when seen combined.

The algorithm which overall performed best in experiment 3, 4, 5 and 6 was the Simulated Annealing algorithm. Therefore it is disappointing that the tables found by this algorithm in these experiments (Figure 9-54 and Figure 9-58) did not result in a strong AI for this experiment. This AI instead only managed to play as good as the manually created AI as can be seen in Figure 9-54. Again, the reason for this could be that these tables simply did not fit well together.

In theory, a way to solve the problem of tables not fitting together would be to train all tables at the same time. This would result in tables that were all tuned to the same strategy and which therefore would fit well together. However, as it can be seen from the last four figures (Figure 9-55 to Figure 9-58), and could also be suspected from the results of experiment 7, this is not the case. Actually the tables which were trained at the same time are not nearly as strong as those which were trained individually. The reason for this can probably be found in the large state space that follows when adding more parameters to tuned: The larger the state space, the longer time will it take to find the optimal state.

Whether the tables could in fact have been improved by letting the algorithms search through the state space for a longer time will not be investigated further in this thesis.

9.9 Experiment 9

Purpose

One of the AIs that performed best in experiment 8 was the one made up of tables found by the `Random IndexAndSign` algorithm (Figure 9-51) and the AI that performed worse was the one made up of tables found by the `Monte Carlo` algorithm (Figure 9-53) when training all four tables at once in Experiment 7.

This last experiment will show how well these two AI perform when playing against 500 randomly made AIs. That is, how well are the two AIs in general?

The first AI showed to be clearly better than the manually created AI, but it has not been shown whether this also means that the AI is good against a lot of other AIs.

Condition

The two AIs described before will play one game against each of 500 randomly created AIs.

Figures

Figure 9-59 shows the results of the two described AIs playing against 500 randomly created AIs.

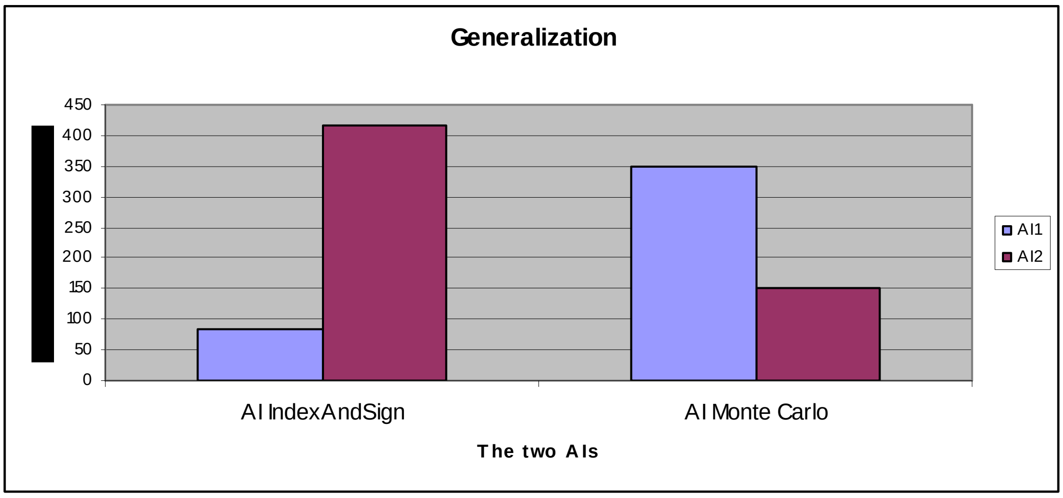


Figure 9-59

Conclusion on experiment 9

The AI based on the tables found by the Random IndexAndSign algorithm wins just over 400 of the games (see column 2 in Figure 9-59), whereas the AI based on the tables found by the Monte Carlo algorithm (when training all tables at the same time) only wins 150 of the 500 games (see column 4 in Figure 9-59).

The AI in column 2 will therefore very often be a lot better than a randomly created AI. However, this does not necessarily mean that this AI would be capable of winning over for example 500 different manually created AIs. It could simply be because the randomly created AIs perform poor. For example it has been seen in the previous experiments, that when the Simulated Annealing method explores the state space by creating random tables, far from all of these tables make the AI perform well. Therefore the first two columns in Figure 9-59 could as well be showing the tendency that randomly created tables normally play a poor game.

However, the AI based on the tables found by the Monte Carlo tables only wins 150 out of the 500 games and this could indicate that the randomly created AIs indeed can provide some challenge for an opponent. If this is the case, then the two first columns on the figure might actually show that the Random IndexAndSign algorithm managed to find some unbalanced aspect of the game. For example that always producing on specific type of unit or always following the exact same strategy would make the AI win more than it would loose. Such balance problems are found from time to time in RTS games after they have been released and updates to the games are therefore created that make sure that the game stays balanced.

Whether this is the case for this game will not be examined further in this thesis. This would require that the AI found by the Random IndexAndSign algorithm would have to play against lots of other AIs (which were not just produced at random) and see whether the strategy followed by this AI would make it win over a certain percentage of the opponent AIs.

9.10 Conclusion on the experiments

As described in this chapter various experiments were carried out in order to show different properties of the four learning algorithms. These experiments showed that there was a big difference in how well the four algorithms performed and also that the number of tables being trained at a time had a big influence on the algorithms performance.

Especially in experiment 3 where only the WhichUnitIsNeeded table was trained did the algorithms perform well and found AIs that were clearly better than the manually created AI. Only the Monte Carlo algorithm did in this experiment not manage to find an AI that was clearly better than the opponent AI.

The Monte Carlo algorithm

In general the Monte Carlo algorithm did not perform really well though it on the other hand never performed as bad as some of the algorithm did in for example experiment 5 and 6. This is somewhat disappointing since this is the algorithm that uses most information from the game to decide how the tables are to be modified. It is perhaps not surprising that the

Monte Carlo algorithm in general did not perform as well as the Simulated Annealing method since this method makes a very thorough exploration of the state space (and which therefore also is very expensive). However, it was hoped that the Monte Carlo algorithm would have performed better than the Random IndexAndSign and the Random IndexValueAndSign algorithms since these algorithms basically search the state space at random (one more randomly than the other). But as the experiments have shown, the Monte Carlo algorithm never really managed to find tables (AIs) that were clearly better than the opponent, which the two random algorithms did, though they also in some experiments performed really poorly.

The reason for this could be that the algorithm simply was not designed in a way that makes it good at finding the best possible tables. As it is now, the algorithm stores all the actions – or choices - made by the AI during a game and also stores the state which the AI was in when taking this action. The algorithm then determines which action was taken most often and if the AI looses the game, the algorithm decreases the probability for taking that action in that state again. Another approach could be to increase the probability of the action with the lowest probability in each state when the AI looses. This would more clearly let the AI explore new states instead of always averaging the probabilities as it happens now (see Figure 9-60). The algorithm could also have been modified in other ways, but whether these modifications actually would make the algorithm better will be left as a question not further investigated in this thesis.

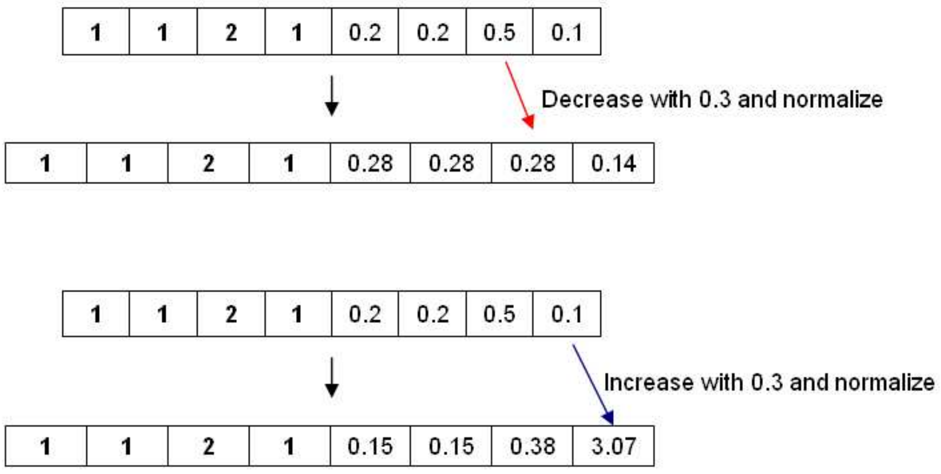


Figure 9-60 A new Monte Carlo Approach

Number of tables trained at a time

Another issue that was dealt with in the experiments was the number of tables trained at a time. Experiment 8 shows that training the tables individually results in far better AIs than when all the tables are trained at the same time.

However, training four tables at a time makes the state space at least 4 times larger. It can even be argued that it becomes 64 times bigger (4^4). But in spite of this growth of the state space, the algorithms training all tables at a time were only run for twice as long as the algorithms training one table at a time. The reason for this is the time it takes to run an experiment. One game takes approximately 4 seconds to play (when the game is set to run 100 times faster than normal). When for example training one table with the Monte Carlo algorithm 50 rounds of 100 games are played (the AI is modified 50 times). This results in 5000 games which take approximately 5-6 hours to complete. Had the algorithms been run for four times as long this would have been at least 20 hours for the Monte Carlo and about 40-50 hours for the Simulated Annealing. This was simply not possible to accomplish in this thesis.

Another approach that could have been taken would be to train some of the tables together and let some be trained individually. For example, the UnitOrWorker table and the SoldierOrBarrack could have been trained together since it primarily is in these that the strategy for whether for example a fast attack or strong economy should be preferred is decided. On the other hand the WhichUnitIsNeeded is to make sure that the AI always builds the type of units that is most needed when taking into account the units in the AIs and the opponent's army. Whether this approach would result in stronger AIs is left for further investigation.

10 Discussion

The goal of this thesis was to develop an AI that would be capable of learning a counter strategy to a new strategy used by an opponent, even after a game had been released. For example, this could happen right after a game was over, where the AI, if it lost the game, could make small modifications to its tables. When playing the next game the AI might have become good enough to beat the human player. If not, the AI could be modified again etc.

However, a number of problems were identified related to the resource management AI developed in this thesis:

First of all, if the learning algorithms are to be able to train the AI, the opponent has to use the exact same strategy over and over again. If this is not the case, the AI will loose everything it has learned when the opponent changes strategy, since the table modifications learnt wont be optimal against the new strategy used by the opponent. Even though some players tend to stick to one strategy, it is very unlikely that a player will stick to the exact same strategy for many thousands of games. Also, if another player borrows the game, his strategy might differ from the owners' strategy and everything learnt will therefore be lost.

A solution could be to let the game remember how the opponent played in a game (when was a building constructed, where was it placed, what types of units were constructed etc.) and then to use this information to teach the AI the new strategy (counter strategy). Let's imagine a situation where a human player tries a new strategy against the AI and wins the game since the AI does not know this new strategy. If during the game, the AI remembered how the opponent played, it could, once the game was over, be put into a learning mode where a number of simulated games was run between an AI following the new strategy just used by the human player and one who tried to find a counter strategy to this strategy. When this strategy had been found the game would exit the learning mode and would be ready to be played again. This learning process could perhaps be given 5, 10 or 20 minutes depending on how patient the human player was and in this way a player could decide which strategies he wanted the AI to learn.

However, even though a solution could be found to the problem of a player having to play the exact same way for many games, the time it takes for the learning algorithms to find a counter strategy to a given strategy is too long to actually be useful in an online setting. As it was seen during the experiments described in the previous chapter (9), the algorithms did not find a good counter strategy in 10 or 20 minutes. Instead they needed between 5 and 40 hours to find a good counter strategy, and it cannot be expected that a human player will let his computer run for 40 hours to let it learn a new strategy.

Although the resource management AI developed in this thesis cannot be said be able to learn online, it would still be very effective if the learning process were to take place before the game was released. In such a setting, various AIs could be set to play against each other and in this way strong strategies could be found without the developers having to hardcode these strategies. Also, new strategies, which the developers might not have thought of, might be found before the game was released. This would result in more strategies that were ready to challenge the players when the game was released, and when being trained offline a 40 hour training session would not be unthinkable if it in the end would increase to quality of the game.

Improving the resource management AI

As mentioned above, the developed AI is not capable of learning fast enough in an online learning situation but would be suitable for offline learning. The question therefore is how the AI should be changed to make it capable of also learning online.

The learning algorithms of course are one part of the AI that could be improved. This could be done by using more controlled and well-considered methods for searching through the state space instead of doing the random search as many of the algorithms did in this project. In the Monte Carlo algorithm (see section 8.2.4) we already tried to do this but the algorithm still did not perform very well. Specific improvements of this algorithm were discussed in the previous chapter, but a more general improvement could be to let the algorithm focus more on what the opponent was doing instead of trying to find out what the AI was doing wrong. E.g. instead of first trying to increase the probability of building a lightsoldier in a given state, and then, when this does not lead to success, trying to increase the probability of building a heavytank in this state etc. the algorithms could base the probability modifications on the unit types build by the opponents. If the opponent often builds a lighttank then build heavytank and so on. By searching through the state space in this way, hopefully the convergance rate of the algorithms could be decreased but it would require for the resource mangement AI to know which types of units were strong against which other types of units.

Another part of the AI that could be improved is the Bayesian Network that was used as the model for the AI. Whether or not Bayesian Networks are suitable for building a resource management AI is a bit hard to answer. As such, Bayesian Networks have been a good way to model the choices a player has in a RTS game. Choices that, when added together, make up the strategy of a player. This has made it possible to develop new strategies by modifying the values in the tables instead of having to hardcode new strategies. On the other hand, since the learning algorithm needed such a long time to find good AIs, it could seem that maybe the entire approach of making a node - or a table - for each of the resource management related choices in a RTS game simply is not a good idea. The game developed for this thesis only resulted in 4 tables that needed to be trained. However, in larger RTS games this number will surely be a lot higher resulting in even more time necessary to train the AI. Therefore, if Bayesian Networks are to be used as the resource management AI either learning algorithms which are a lot faster than the ones developed in this thesis need to be developed or the network needs to be redesigned in a way that will descrease the number of table entries needed.

Is the developed AI really a resource management AI?

In section 3.1 it was decided to focus on developing a resource management AI that should be capable of adapting itself to new strategies. Furthermore it was decided not to implement any higher-level AIs in this thesis and so the entire strategy aspect was placed in the resource management. It is also easy to make a connection between the concept of a strategy and the resource management AI: e.g. if we have 9 workers then stop producing workers and start producing military units instead. This is an example of decisions that the resource management AI developed in this thesis has been taking all the time, of course based on some probability distributions. However, by leaving such decisions to the resource management AI we might have turned it into something it was not supposed to be, namely a component which lays the overall strategy for the AI. Even though the resource management AI appears this way, because the higher-level AI components are missing, this is actually not the case.

It is true that the tables express a given strategy or make sure that a given strategy is being followed. If the game however was to be further developed a component in charge of deciding which strategy to follow would have to be implemented. This component would continuously make an analysis of the strategy followed by the opponent and based on this analysis change the tables in the network to make sure that the optimal tables (strategy) were being used. That is, this component would decide which strategy to follow and the resource management AI would make sure that this strategy was actually being followed (see **Fejl! Henvisningskilde ikke fundet.**).

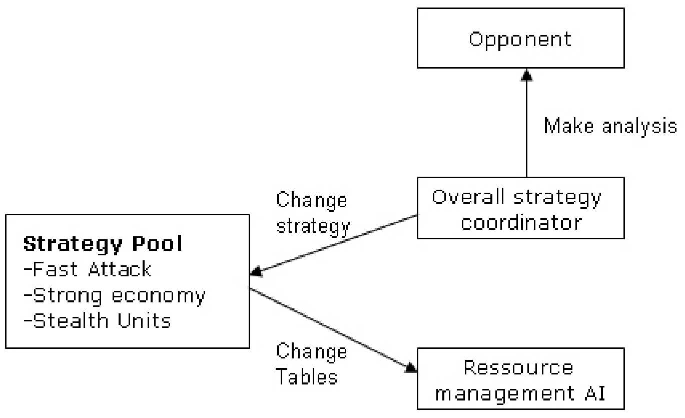


Figure 10-1 New AI design

But even though the resource management AI is not actually laying the overall strategy, it is perhaps given too much responsibility after all by having a lot of details about the current status of the game to be able to decide what to build. The problem with this is not so obvious in this game where only four different units and two different buildings can be constructed. But if the resource management AI were to be integrated in a larger RTS game, there would be a lot more units and building to choose between which would lead to a very complicated decision process. So how could the resource management AI be changed to avoid this problem?

At the very foundation, the resource management AI is supposed to decide what should be build. This part is clear. However, the information on which it bases these decisions has perhaps been to detailed - or has been information which actually belonged to a higher-level AI. In the following a design of the AI including some higher-level AI components are described which could have been used instead.

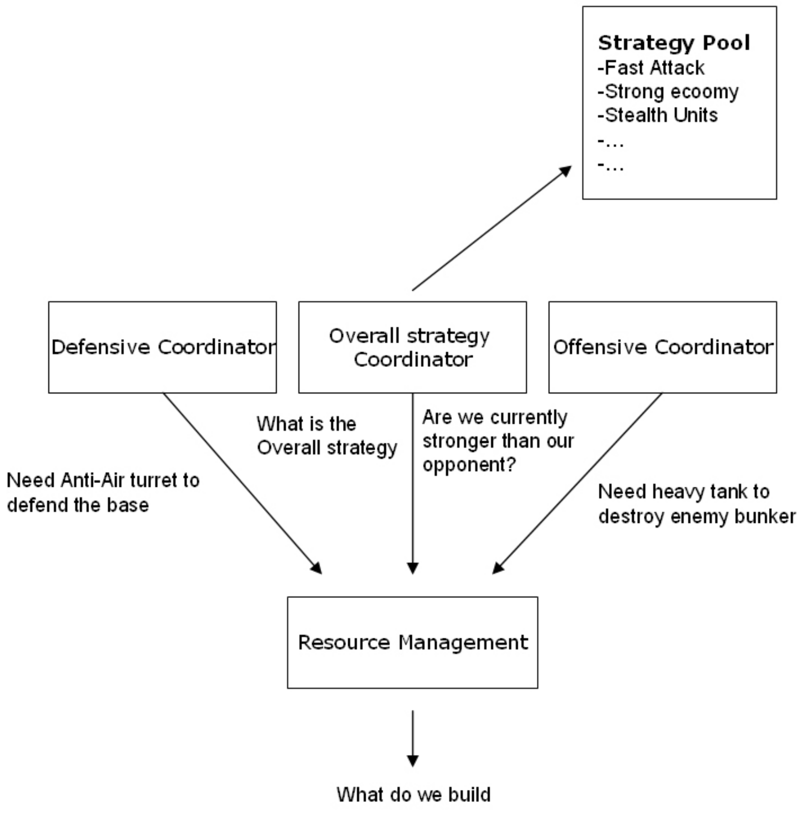

Strategy Pool
-Fast Attack
-Strong ecoomy
-Stealth Units
-...
-...
Defensive Coordinator
Overall strategy Coordinator
Offensive Coordinator
What is the Overall strategy
Are we currently stronger than our opponent?
Need Anti-Air turret to defend the base
Need heavy tank to destroy enemy bunker
Resource Management
What do we build

Figure 10-2 Alternative AI design including higher-level AI components

In this example some high-level AI components inform the resource management AI what they need and also what the current strategy is and whether or not we currently are on top of the situation or whether the opponent is giving us some trouble. From these informations the resource management could build a Bayesian Network from which it could decide which unit or building were currently most urgent to build. And this would also relieve the resource management AI from having to know about every detail of the current game status to be able to decide what to build. The resource management AI would still be part of expressing the overall strategy and would still need to be trained to learn new strategies (which could then be stored in the Strategy pool) but now also the higher-level AIs would need to undergo training before the strategy could be made effective.

A question then is how these high-level AIs should be developed. As mentioned it becomes a complicated process to train a Bayesian Networks what to build in a given situation when involving more than just a few tables. But maybe the place to use a Bayesian Network in a game actually would be as for example the overall strategy coordinator where more abstract decisions has to be made based on limited information about the current game situation. For example, by scouting the map early in the game it might be discovered that the opponenet has a given building which allows for constructing stealth units. This information alone – and not the actual number of enemy stealth units – could make the overall strategy coordinator prefer strategies involving opponent stealth units from strategies which do not. And following a strategy that involves opponent stealth units could strongly influence what units or building in the end was constructed by the resource management AI.

11 Conclusion

The goal of this thesis was to develop an AI for a real-time strategy game that would be capable of adapting to new strategies used by an opponent. The reason for this was to solve the problem that the single player mode of RTS game quickly becomes less attractive than the multiplayer part because the AIs do not adapt to the new strategies used by a human.

Through the analysis it was decided to focus on the problem concerning which units and buildings to produce during the game instead of focusing on how the individual units could be used in new ways. More specificly it was chosen to develop a resource management AI primarily because resource management is a very central aspect of RTS games but also because resource management only peripherical is discussed in RTS game theory [11, 17, 19].

To be able to evaluate the developed resource management AI either the AI had to be integrated with an already existing RTS game or a new small RTS game had to be developed from scratch. The latter option was chosen since the game in this way could be specificaly developed to evaluate the resource management AI.

Finally it was decided to use Bayesian Networks as the model for the resource management AI and the concepts of Reinforment Learning for training this network. Other theories were also considered, but Bayesian Networks and Reinforcement learning were found to be the two theories best suited for this task.

Based on the decisions made in the analysis a small RTS game and a resource management AI were implemented. Furthermore a small framework was created to make it easy to construct and modify the network used as a model for the resource management AI. On top of this, a function was developed for getting information from the network. This function used recursive calls to traverse the network and was made dynamic enough to work in spite of modifications of the network.

With the game and the AI in place four learning algorithm were developed to train the network and also functionality for storing and retreaving the tables from the network were implemented. These four algorithms were evaluated by letting two AIs play against each other in nine different experiments. One of these AIs was based on manually created tables and the other was trained to become better than the first AI. The results of the experiments showed that there was a big difference in how well the algorithms performed and that also the number of tables being trained at a time had a great influence on the algorithms performance.

In general, the result of the experiments was that the leaerning algorithms developed were not capable of training the Bayesian Network fast enough to let the resource management AI be used in an online learning setting. On the other hand, the resource management AI could be trained in an offline learing setting which could relieve the game developers from having to hardcode the strategies. The conclusion on Bayesian Networks was that Bayesian Networks can be used in games but should be done so only in situations involving few parameters – for example as the overall strategy coordinator in a RTS game. The resource management AI for the small RTS game developed in this thesis already had so many parameters that training the network became a slow process. Therefore very specific functionalities as for example training the individual units in an RTS game would probably be solved better by using other theories like genetic algorithms.

Some suggestions were given to how the algorithms could be improved and further development of the resource management was discussesd.

Anders Walther, 1st June 2006

12 References

[1] David M. Bourg, Glenn Seemann - AI for Game Developers, O'Reilly Media, Inc.; 1 edition (July 23, 2004)

[2] Richard E. Neapolitan – Learning Bayesian Networks, Prentice Hall; 1st edition (April 1, 2003)

[3] Richard E. Neapolitan – Learning Bayesian Networks, Prentice Hall; 1st edition (April 1, 2003) page 21

[4]Chanyun Wang – Bayesian Belief Network Simulation

[5] http://ai-depot.com/FiniteStateMachines/FSM-Background.html

[6] David M. Bourg, Glenn Seeman - AI for Game Developers chap 10, O'Reilly, 2004

[7] http://www.esreality.com/?a=post&id=663198

[8] David M. Bourg, Glenn Seeman - AI for Game Developers chap 13, O'Reilly, 2004

[9] David M. Bourg, Glenn Seeman - AI for Game Developers chap 12, O'Reilly, 2004

[10] http://www.links.net/dox/warez/games/edu/rts/

[11] Brian Schwab, AI Game Engine Programming, Charles River Media; 1 edition (September 2004)

[12] http://www.fileuniverse.com/Total_Annihilation_Mirror/totala/index.html

[13] http://www.dawnofwargame.com/

[14] http://www.blizzard.com/starcraft/

[15] http://www.blizzard.com/war3/

[16] http://www.ea.com/official/cc/firstdecade/us/index.jsp

[17] Bob Scott, AI Game Programming Wisdom, Charles River Media

[18] www.empireearth2.com

[19] Todd Barron, Strategy Game Programming with DirectX 9.0, Wordware publishing inc, 2003

[20] Matt Buckland, AI techniques for game programing, Stacy L. Hiquet, 2002

[21] http://www.gatsby.ucl.ac.uk/~zoubin/approx.html

[22] http://www.cse.unsw.edu.au/~cs9417ml/RL1/tdlearning.html

[23] http://www.cs.ualberta.ca/%7Esutton/book/ebook/node16.html

[24] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns, Addison-Wesley Professional; 1st edition (January 15, 1995)

[25] Mance E. Harmon, Stephanie S. Harmon, Reinforcement Learning: A Tutorial http://www.nbu.bg/cogs/events/2000/Readings/Petrov/rltutorial.pdf

[26] Alex J. Champandard, AI Game Development, New Riders Games; 1st edition (October 31, 2003)

[27] Stephan ten Hagen and Ben Kröse, A Short Introduction to Reinforcement Learning, http://citeseer.ist.psu.edu/tenhagen97short.html

[28] Tim Eden, Anthony Knittel, Raphael van Uffelen, Reinforcement Learning

http://www.cse.unsw.edu.au/~cs9417ml/RL1/algorithms.html

[29] http://en.wikipedia.org/wiki/Simulated_annealing

[30] http://en.wikipedia.org/wiki/Random_variable

[31] Stuart Russel, Peter Norvig, Artificial Intelligence - A Modern Approach, Prentice Hall; 2nd edition (December 20, 2002)

[32] http://www.gamegirlz.com/pc-game-reviews/starcraft.html

[33] http://nn.cs.utexas.edu/NERO/about.html

[34] Smith, Harvey. "Orthogonal Unit Differentiation". Paper presented at the Game Developers' Conference, San José, March 4-8, 2003. http://www.gdconf.com/archives/2003/Smith_Harvey.ppt

[35] http://www.tips.dk/

13 Appendix A

Content of the CD

On the CD all the code for the RTS game, the ressource management AI and the learning algorithms developed in this thesis can be found in the GameAI folder. All the code has been written in C++ using Microsoft Visual Studio 2005. To be able to compile the project Visual Studio 2005 or a newer version will be needed.

If Visual Studio 2005 or a newer version is available, the project can be opened via the GameAI3.sln file in the GameAI folder. To decide which learning algorithm to use and which tables to modify the code needs to be modified. See appendix B for a description of how to do this.

The experiments and the exe files for each experiment

For each experiment performed in this thesis, an exe file has been created. These have been placed in the Experiments folder. If a particular experiment needs to be performed again all that needs to be done is to copy that exe file from that experiment into the root of the GameAI folder and then run the file. Also, if either player1 or player2 is to use special tables during the game, these can be copied into this folder before the game starts. The name of these tables are: needs, unitorworker, soldierorbarrack and tankeorfactory for player1 and needs2, unitorworker2, soldierorbarrack2 and tankeorfactory2 for player2. If no changes are made, the tables will be identical for the two players and will be tables that have been manually created.

Running an experiment:

If an experiment is run the result of the experiment will be placed in a textfile in the result folder. Also, the best tables found during the experiment as well as the final tables will be stored in the tables folder.

Playing against the computer

Two exe files have been created which will make it possible to play against the computer. These have been placed in the ‘playe against the computer’ folder. In one of them the game speed will be 1 and in the other the game speed will be 5. To play the game copy the exe file into the root of the GameAI folder.

Creating random tables

To create 1000 random tables run the game and click on the buttom for creating the tables of interest. The tables will be placed in the tables folder.

14 Appendex B

There has not been implemented a GUI which would make it possible to choose how the game and the AI are to be initialized. Therefore, these initializations must be made by hand in the code. This is exlained next:

- **Set the speed of the game:**
  - GameAIDlg class:
    - Line 45: Change the `gametime` variable

- **Decide whether player 1 one should be an AI or a human player:**
  - GameAIDlg class
    - Line 481: Call the `battle` function if player 1 is to be a human player or the `battletest` if it is be controlled be an AI
  - GameCore class:
    - Line 253: Out comment the call of the `RunAI1` method

- **Decide which experiment / test to run:**
  - GameCore Class:
    - Line13: Change the `testtorun` variable

- **Decide which table to modify:**
  - TestRunner Class:
    - There is one method for each of the four learning algorithms in the TestRunner Class. In each of these methods it must be made sure that only the table(s) of interest are modified by out commenting modifications of the other tables. This has to be done four places in the `RandomIndexValueAndSign` method, the `RandomIndexAndSign` method and in the `Monte Carlo` method. In the `Simulated Annealing` method this has to be done 9 places.
    - Furthermore, the TestRunner object creates random tables in its constructor. Again make sure that this only happens for the table(s) of interest

- **Special case – The `Monte Carlo` algorithm**
  - Node class:
    - If you are training the tables with the `Monte Carlo` algoritm, you must decide which of the nodes from the network will save its state and decisions through out the game. This is done from line 68 to line 83 in the Node class.