TECHNISCHE UNIVERSITEIT EINDHOVEN

Department of Mathematics and Computing Science

MASTER'S THESIS

Multi-player game search in Cartagena

by
X.A.H. Steinmann

Supervisors: Dr. ir. A.I. Cristea and Dr. ir. J.W.H.M. Uiterwijk

Eindhoven, April 2003

Preface

This document was written as a Master's Thesis for the Computing Science department of the Eindhoven University of Technology. The work was performed at the Institute for Knowledge and Agent Technology (IKAT) of the Computer Science Department of the Universiteit Maastricht.
This document is meant for readers who are interested in heuristic search in games, especially search in multi-player games. The information in this document can be used to give an insight in multi-player game search and to improve multi-player game search algorithms.
I want to use this opportunity to thank my supervisor in Maastricht, Dr. ir. Jos Uiterwijk, whom I had regular contact with and who knows a lot about game search. I also want to thank my supervisor in Eindhoven, Dr. ir. Alexandra Cristea, even though our contact was very irregular. Furthermore, I want to thank the other two persons in my commission, Prof. dr. Paul de Bra and Dr. ir. Tom Verhoeff. And last but not least I want to thank my two physiotherapists, Drs. Jean-Pierre Bergmans and Lot Oudshoorn, without whose help this document never would have been completed.

Index

1. Introduction

A lot of research has been done in the area of two-player games, which resulted in efficient two-player search algorithms. But in the field of multi-player games (games with more than two players) no extensive research has taken place. Multi-player game search is more difficult because players can for instance form coalitions, and there are more players to take into account when a player determines what move to make. Therefore the multi-player search algorithms are quite inefficient.

This document describes three new multi-player search algorithms, which are an attempt to improve the efficiency and performance of the current multi-player search algorithms. One of the algorithms is derived from an existing multi-player search algorithm, one algorithm uses Monte Carlo search and the last algorithm uses Neural Networks. All the algorithms are tested by playing the board game Cartagena.

In the second chapter the rules of Cartagena are described. A short introduction on search algorithms and neural networks follows in chapters three and four. The fifth chapter describes the multi-player search techniques that were researched, including the algorithm that uses neural networks. The implementation of the game and the computer players are described in the sixth chapter, and the performance and efficiency results of the algorithms are described in the seventh chapter. In the last chapter the conclusions and possible future research on the subject are stated.

2. Cartagena

In this chapter the board game Cartagena will be described. The chapter starts with the description of the game, followed by a short description of tactics involved. This should give a good idea about the difficulty of the game.

2.1 The rules

Cartagena is a multi-player board game for two to five players. The game tries to reproduce a famous 1672 jailbreak of about 30 pirates from the supposedly impregnable fortress of Cartagena. Each player controls a group of 6 pirates and tries to have all 6 of them escape through the tortuous underground passage that connects the fortress to the port, where a sloop is waiting. As soon as a player has brought his 6 pirates aboard, the sloop sails away and the game is over. There are 2 versions:

- The **Jamaica version** in which cards are concealed (more luck): The players hold their cards in hand, hidden from the other players. The rest of the deck is placed face down on the table to form a stock, from which players will pick new cards during play.
- The **Tortuga version** in which all cards are face up (more skill): The players lay out their cards face up on the table for all to see. From the rest of the deck, 12 cards are dealt face up in a row. The card with the arrow is placed under the row and shows the direction of the sequence - that is, the order in which new cards are picked up during play instead of from a stock. When the row is depleted, a new 12-card row is laid out.

The rules of the game:
The game is played in turns clockwise. The player who looks most like a pirate begins the game. Whenever it is your turn, you may take 1 to 3 actions. Two kinds of actions are allowed:

1. Play a card and advance a pirate:
Play a card to the discard pile, select one of your pirates (either one already in the underground passage or one who has yet to enter it) and advance him to the next vacant space marked with the same sign as the sign on the card you just played. Vacant means that no other pirate occupies the space. If there is no vacant space with the sign you selected ahead of the pirate you intend to move, you can advance him all the way to the end of the passage and let him climb aboard the sloop.

2. Move a pirate backwards and pick up one or two new cards:
Select one of your pirates and move him backwards to the first space occupied by one or two pirates, whether yours or belonging to an opponent. While moving backwards, vacant spaces and spaces occupied by three pirates are ignored and passed by. (Note that there can never be more than three pirates on a space). If your pirate moving backward lands on a space occupied by 1 pirate, pick up 1 card. If occupied by 2 pirates pick up 2 cards. But keep in mind that when moving backwards you are not allowed to skip a space

occupied by one pirate in order to reach one occupied by two. Neither is it allowed to move back to the starting position. In the Jamaica version cards are picked up from stock, in the Tortuga version from the row of face-up cards. When the deck is depleted, form a new deck with the cards in the discard pile.

When it is not possible for a player at the beginning of his turn to advance a pirate or move a pirate backward, that player receives one card and his turn is ended.

End of game:
First to land his 6 pirates on the sloop wins the game.

2.2 Some insights in the game

The trick is that you have to move your pirates backwards in order to allow them to move forwards. So it is essential to move your pirates as little backwards as possible to collect as many cards as possible, and to move your pirates forwards as far as possible with every card.

But there lies the tricky part because your opponents try to do these same things, and the one thing you do not want to do is help your opponents. Unfortunately you cannot waylay your opponents. You can only 'not help' them, so making a row of pirates on the same signs (a chain) is good for you but even better for your opponents when they have a card with that sign. It is therefore important to play cards that your opponents do not have and to make it difficult for your opponents to move forwards a lot or to easily collect cards.

The strategy that advanced players use is to collect as many cards as possible in the first couple of turns, and then to suddenly rush to the sloop. When you calculate correctly and you are the first to do this then you have a very good chance of winning. But then again it is important to break up large chains of the same sign to avoid benefiting your opponents too much.

Cartagena is a very difficult game with very little rules and it is enjoyed by many who play it, so it is an excellent game to research multi-player search algorithms for.

3. Search algorithms

This chapter will discuss some aspects of search in a game. First of all we will describe what search is, why it is used and its strengths and weaknesses. Then some common search techniques for two-player and multi-player games will be described. This chapter makes use of [Russell and Norvig, 1995] for search in two-player games and [Sturtevant, 2002] for search in multi-player games. Furthermore the articles [Marsland, 1986], [Sturtevant and Korf, 2000] and [Ross, 2002] proved useful.

3.1 What is search?

Imagine that someone is playing a game of Cartagena. What he wants is to win the game. In other words, he wants to reach a state in which he is the winning player. At each state in the game he has a couple of possible actions, which lead to other states. This information can be used to consider subsequent states of a hypothetical game that eventually will lead to a winning game state (goal state). The process of looking for such a sequence is called a search.

A search algorithm takes a problem and returns the solution in the form of an action sequence. Once a solution is found, the actions it implies can be carried out. Thus search is a "formulate, search, execute" algorithm.

In order to search, the player must be able to identify and define the state he is in, what possible actions he can undertake and what the consequences of those actions (state changes) are. He should also be able to identify a goal state and possibly be able to put a cost on different state sequences, because for instance in Cartagena, as in many other (board) games, winning in 1 turn is preferred over winning in 2 turns.

When a player is generating an action sequence he is continually trying out actions that lead to different states. This process is called *expanding* a state. At each level one action is chosen and expanded. This is the essence of a search, and can be seen as the construction of a search tree that is superimposed over the state space.

Search tree definitions

There are many definitions in search problems and many of those definitions will be frequently mentioned in the next couple of sections and chapters. Therefore a short list of definitions is added:

- *path*: a sequence of states linked by the actions that trigger the state changes.
- *node*: a hypothetical state that is reached when performing an action. A search tree is built up by nodes, and the nodes are connected by actions.
- *branching factor*: the (average) number of possible actions at a node (also called the *search width*).

- *terminal state*: a state in which a game ends, for instance a goal state.
- *terminal node*: a node that can not be expanded
- *root*: the starting node from which the search tree is built up.
- *leaf*: a terminal node or a node that is not (yet) expanded in the tree.
- *child of node 'n'*: a node that lies one level lower than *n* and is connected to it.
- *parent of node 'n'*: a node that lies one level higher than *n* and is connected to it.
- *search depth*: the number of node generations already expanded at a particular moment.
- *d-ply search*: a search with a limited search depth of *d*.
- *pruning*: eliminating parts of a search tree by not expanding them.
- *breadth-first search*: building up a search tree by expanding all the nodes at one level before expanding nodes at deeper levels, starting at the root node.
- *depth-first search*: expanding a search tree by expanding the first child of the last expanded node. Only when the search hits a dead end (a terminal node) or a given maximum depth does the search go back and expand nodes at shallower levels.
- *solved problem*: when the absolute best move can always be determined in a problem.
- *evaluation function* or *utility function*: a function that evaluates a state and gives it a value. When a state is better than another state then the evaluation function should give it a higher value.

3.2 Two-player games

The presence of an opponent makes the search somewhat more difficult because it is uncertain how the opponent will play. Moreover, what makes most games really difficult is the fact that they are usually too hard to solve. A game of chess has an average branching factor of 35 and games usually take about 50 actions per player, so a search tree needs about 35^{100} nodes in the beginning of the game!

Therefore in game-tree search the idea is to find a good approximation of the best action in as short a time as possible. This is done by pruning parts of the tree that are considered to make no difference to the final choice, and by using an evaluation function to estimate how good a state is without searching the entire search tree.

The trade-off between search depth and the quality of the evaluation function

It is very important to have a good evaluation function. When the evaluation function can perfectly estimate the chances of winning then a (deep) search is no longer necessary. On the other hand, when search to the terminal nodes is possible then no (good) evaluation function is necessary.

In practice both situations are not realistic, because a very good evaluation function usually takes a lot of time to calculate and requires a lot of information about good play, and a deep search takes too much time. Therefore a trade-off is necessary between the

two: increase the search depth somewhat and reduce the complexity of the evaluation function, or the other way around. Especially when there is no good strategy / evaluation function known, a simple evaluation function with a deep search is usually the best.

We will now look at a well-known search algorithm called Minimax, and a commonly used improvement for Minimax called Alpha-Beta.

3.2.1 Minimax

In a two-player game there is a player who has to determine his best action (MAX). It plays against one opponent and assumes that the opponent tries to minimize his chances of winning (MIN). Suppose that for the game that is played the entire search tree can be constructed, for instance in tic-tac-toe. MAX builds the entire search tree and attaches utility values at the leaves, for instance 1 for a win, 0 for a draw and –1 for a loss. MAX can then choose a sequence that leads to a winning state.

Unfortunately MIN's actions can (and probably will) influence the result of the game. MAX must therefore find a strategy that that will lead to a winning (or optimal) terminal state, no matter what MIN does. A 2-ply search tree is shown in figure 3.1.

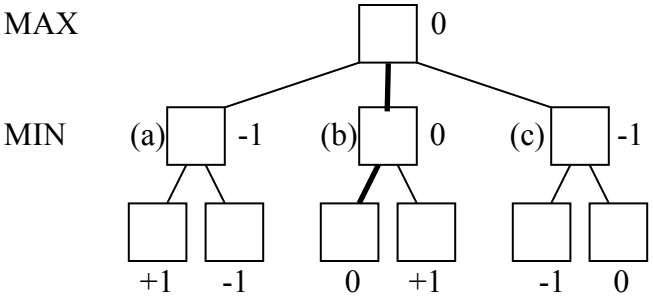


Figure 3.1: a 2-ply Minimax search tree with win, loss and draw utilities in the nodes

At each level MAX wants to maximize the utility because he wants to win, and he assumes that MIN tries to minimize his utility. So MIN chooses between the respective two options –1 at state (a), 0 at state (b) and –1 at state (c). MAX then maximizes amongst the three resulting states; so he chooses (b) with value 0 (a draw) as the best move for him. This algorithm is called the Minimax algorithm. The sequence of actions foreseen (the bold lines in figure 3.1) is called the principal variation.

The Minimax algorithm assumes that the program has time to search until it reaches terminal states, but usually this is not possible or practical. This is also the case in Cartagena where the average branching factor is about 2000 moves and a game takes about 35 moves per player in a two-player game.

Therefore the Minimax algorithm in Cartagena uses an evaluation function to estimate the chances of winning in a node, and the (maximum) search depth is limited from the start. The evaluation function looks at all the positions of the pirates and the cards of the

players to determine how good a state is. With the evaluation in the leaf nodes the best move can be determined as is shown in figure 3.2.

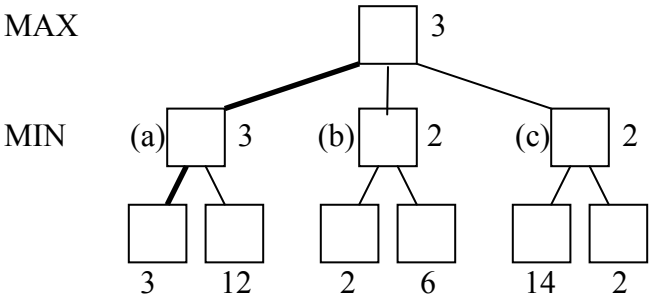


Figure 3.2: Minimax with evaluation values

This search algorithm is still quite slow, because all the possible actions have to be examined. The Alpha-Beta pruning improvement reduces the number of states that have to be examined, so it increases the search speed. The saved time can then be used to increase the search depth.

3.2.2 Alpha-Beta pruning

When it is possible to prune parts of the tree a lot of calculation time can be saved. The Minimax algorithm has the nice feature that it can prune parts of the search tree by using the Alpha-Beta algorithm without affecting the result.

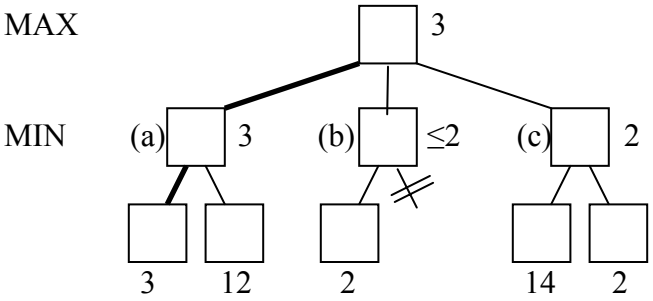


Figure 3.3: Minimax with evaluation functions and Alpha-Beta pruning

Consider the depth-first 2-ply search in the figure 3.3, which shows the same situation as in figure 3.2. MIN chooses for the state with value 3 in (a), but now he remembers that the best value so far for MAX at the root is 3. When the algorithm reaches state (b) MIN finds a utility of 2. The current maximum for MAX is 3 and the minimum in state (b) is 2 or even lower. Because MAX maximizes MAX will never choose (b) as his best move even if (b) is completely expanded, because its utility is lower than 3. Therefore MIN can stop searching in that part of the tree and the rest of the actions at (b) are pruned.

Note that no pruning can be applied at node (c). After examining the first (left) child the minimum in (c) is 14 or lower. Only after examining the second child can the algorithm determine that the minimum is lower than the current maximum in (a).

Alpha-Beta is a depth-first search, so at any time only the nodes along a single path are examined. Let α (alpha) be a lower bound for the value for MAX and β (beta) an upper bound. The α and β values are continually updated throughout the search tree. Whenever a value is outside the α-β window that part of the tree can be pruned. This is where the algorithm inherits its name from.

The number of nodes calculated in a game with branching factor *b* and search depth *d* equals b^d when only the Minimax algorithm is used. With Alpha-Beta pruning this can be reduced to $O(b^{d/2})$ nodes [Russell and Norvig, 1995].

A lot of research has been done in two-player games and also a lot of successes were achieved and reported. However, in multi-player games this has yet to be accomplished. We will examine some multi-player algorithms and see why they are so much more complicated than two-player algorithms.

3.3 Multi-player games

In multi-player games there are more difficulties than in a two-player game. Everybody tries to win, players can team up (temporarily), players can decide to go for second place and many more situations can appear that cannot happen in a two-player game. These are elements that are difficult to work with in a search. There are a few multi-player algorithms that were compared in [Sturtevant, 2002] which will be shortly described here. Of course, there are many other possible algorithms that can be used, but here only the most commonly used and successful algorithms are presented.

3.3.1 Max^n

Max^n is a multi-player algorithm that assumes that all the players are trying to maximize their own score. The algorithm builds up the entire search tree and at each node the algorithm assumes that the players will choose the best move for themselves, ignoring the scores of the other players. See the example in figure 3.4 for a 3-player game; the scores of the players are represented as (score player 1, score player 2, score player 3).

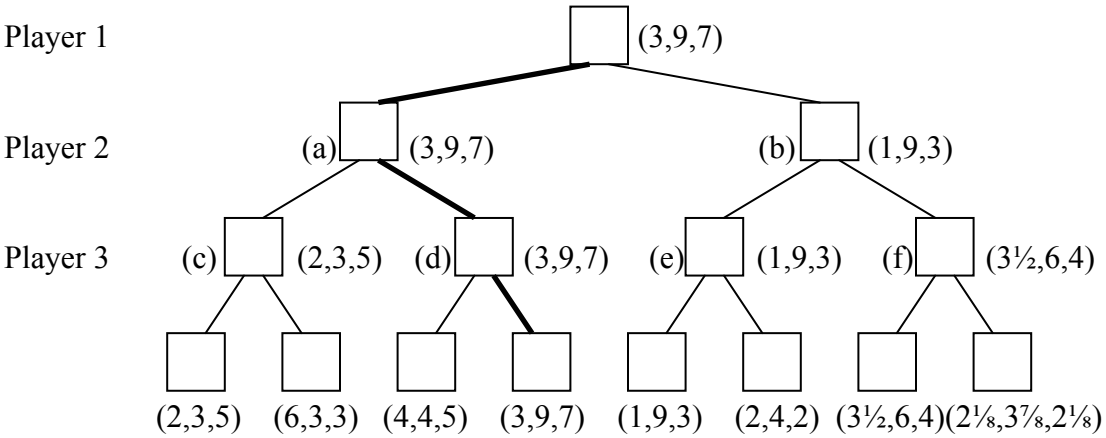


Figure 3.4: Max^n search tree example

Player 1 has to find his best action. He can only take an action leading to node (a) or (b). To choose the best action he first lets player 2 and 3 choose their best actions after which player 1 picks the move with the highest score for him.

In nodes (c), (d), (e) and (f) player 3 picks the move that has the highest value for him (the third value in the evaluations). Player 2 then does the same in (a) and (b) but he looks at the second value and player 1 then has two choices in which he looks at the first value of the evaluations.

A Max^n player tries to maximize his own score but he is almost ignorant with respect to the other players. As can be seen in the example, player 1's estimated maximum score is very bad compared to the other players' scores. Furthermore, pruning is not possible so the entire search tree has to be built up which leads to slow searching. Therefore the Max^n algorithm is not suited for a deep search or possible cooperative play.

Note the evaluation values of the children of (f). These values may seem strange at first, but they are later used to explain some properties of the Paranoid and the Not-So-Paranoid algorithms. Those algorithms will use the tree in figure 3.4 to determine their best move.

3.3.2 Paranoid

The Paranoid algorithm is an algorithm that uses a Minimax-like approach towards a multi-player game. The algorithm assumes during a search that all the other players have teamed up against the moving player and are trying to minimize the score that the moving player is trying to maximize. This enables pruning.

In the Paranoid algorithm there is only one value to be considered: the value of the moving player. The moving player tries to maximize while the other players try to minimize the score. The pruning takes place when a calculated value is situated outside a window, just like in the Alpha-Beta algorithm. Take a look at the example in figure 3.5.

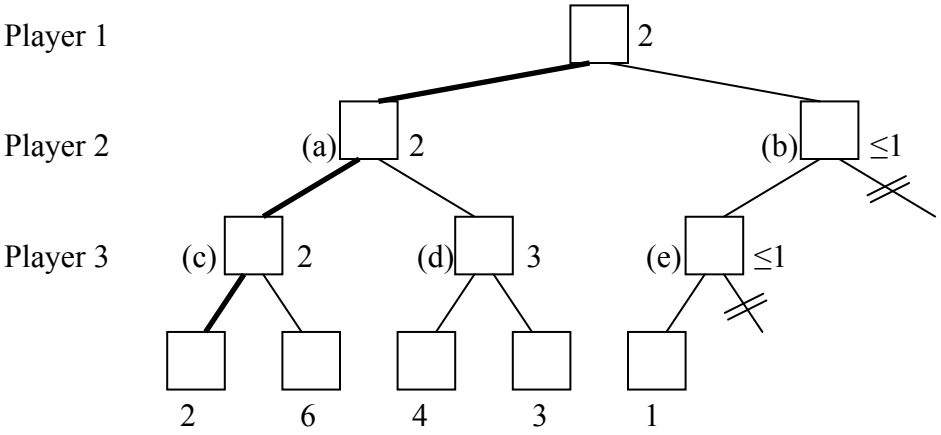


Figure 3.5: Paranoid search tree example

To be able to compare this example to the Max^n example in figure 3.4, the value of a leaf in this figure is the value of the same node in figure 3.4 for player 1.

Player 1 has to find his best action. Players 2 and 3 are on the same side here, so they try to minimize while player 1 tries to maximize. In situations (c) and (d) player 3 chooses the lowest score, as does player 2 in (a) and (b). When player 3 has to decide in (e) he already knows that there is a minimum of 2 for player 1. So after finding a 1 he knows that player 1 will never choose the action that leads to (e), so that part of the tree can be pruned. Similarly, the right sub tree of (b) can be pruned.

This algorithm is exactly the same as Alpha-Beta, only there is a little bit less pruning because player 3 cannot prune. Moreover, the other players try to minimize the score of player 1 while disregarding how this influences their own score. Therefore the terminal state is not the same as in figure 3.4.

With a branching factor b, a search depth d and n players Max^n has to calculate b^d nodes. The Paranoid algorithm could expand only $O(b^{d(n-1)/n})$ nodes instead of b^d in the best case [Sturtevant, 2002].

This type of play may be slightly strange (and paranoid of course), but it reduces the number of nodes that have to be calculated in the search tree. This reduces the calculation time of the search, and this saved time can then be used to increase the search depth. So this assumption that all the other players have formed a coalition can increase the search depth, which is very important in Cartagena as can be seen in the results in chapter 7.

The mentioned multi-player algorithms were not very fast when we started to use them. Therefore some adjustments had to be made.

3.4 Algorithm adjustments

The first problem encountered was that the calculation time of the algorithms was very high, which considerably reduced the search depth. The high calculation time is a direct result of the high branching factor in Cartagena. Every move consists of one to three actions, and every action can be one out of 43 possible actions. This means that there is a maximum of 43^3 (=79507) possible moves at each node! This is an incredible amount, considering that when searching 2 moves deep means a maximum of 79507^2 possible nodes etc. Fortunately the average number of expanded nodes is on average about 600 nodes per move, but in the middle game 2000 expanded nodes per ply is not unusual.

Let us compare 2-player Cartagena to Chess. Chess has a branching factor of about 35 and a search depth of about 14, which means that in chess 35^{14} nodes are expanded. If Cartagena took about the same time as chess to expand a node, it can in the same time search only 8-ply deep with a branching factor of 600. Because the branching factor in the middle game is about 2000 a 4-ply deep search would then be more probable.

The two-player algorithms in Cartagena were very slow so it was therefore decided to use a selective search: the possible actions at each part of a move are reduced to 3. This means that when deciding to take an action, a player only looks at his best three possible actions and moves further from there, which results in a maximum branching factor of 27 (3 actions per move means 3^3 possible moves). This increases the speed considerably but takes a toll on the quality of the search.

Unfortunately a 5-ply search is still about the maximum that can be reached in real-time. The reason for this low search depth still is the branching factor (all the possible actions still have to be calculated at each step, even though only 3 are used) and the small amount of pruning possible in multi-player Cartagena. Furthermore, these two algorithms play very poorly against a human player, which was the main reason for researching better and faster algorithms for multi-player games. The researched algorithms are described in chapter five.

4. Neural Networks

This chapter describes what a neural network is, the way it learns and how it can be used in Cartagena. For more information about neural networks see [Reed and Marks, 1999], [Russell and Norvig, 1995] or the Internet (for instance http://www.ai-junkie.com, the source of the pictures used).

4.1 What is a neural network?

A neural network is a way of trying to simulate the brain electronically. To understand how a neural network works we first must have a look at how the brain works.

Our brains are made up of about 100 billion tiny units called *neurons*. Branching out of the cell body are a number of fibers called *dendrites* and a single long fiver called the *axon*. Dendrites branch into a bushy network around the cell, whereas the axon stretches out for a long distance (up to a meter in extreme cases). Eventually, the axon branches into strands and sub strands that connect the dendrites and cell bodies with other neurons. Figure 4.1 shows the parts of a neuron.

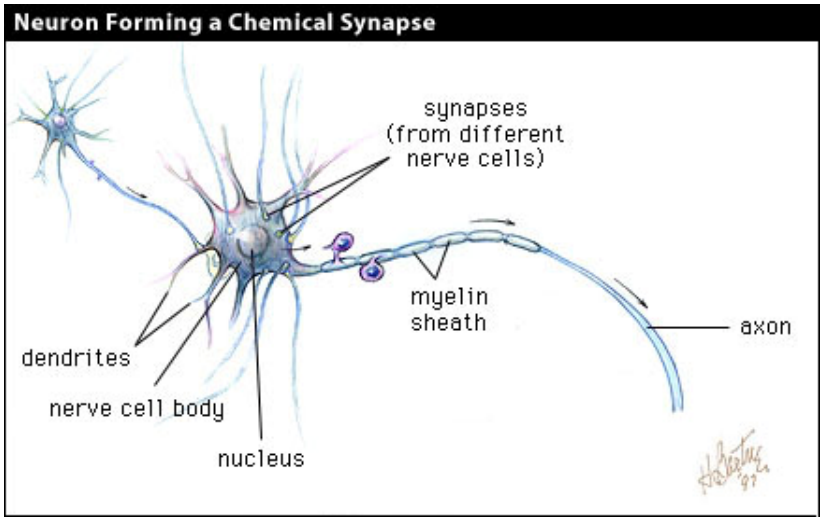


Figure 4.1: A neuron.

Signals are propagated from neuron to neuron by a complicated electrochemical reaction. Chemical transmitter substances are released from the synapses and enter the dendrite, raising or lowering the electrical potential of the cell body. When the potential reaches a threshold, an electrical pulse is sent down the axon (the neuron *fires*). Perhaps the most significant finding is that synaptic connections exhibit long-term changes in the strength of the connections in response to the pattern of simulation.

Neural networks are made up of many artificial neurons. An artificial neuron is simply an electronically modeled biological neuron, also called a *unit*. The units are connected by

links (synapses), and each link has a numeric weight associated with it (its influence strength). Weights are the primary means of long-term storage in a neural network, and learning actually takes place by updating the weights.

Each unit has a set of input links from other units, a set of output links to other units (or to 'the' output), an *activation level* and a means of computing the activation level. See figure 4.2 for a simple sketch of a unit.

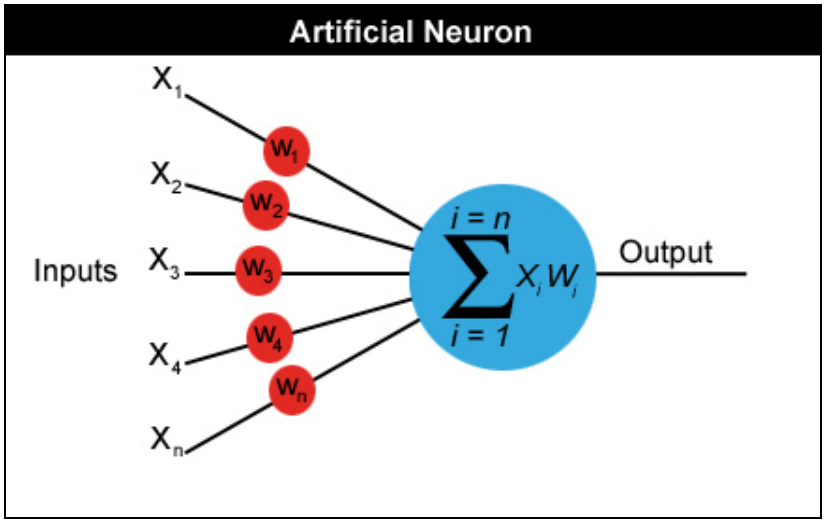


Figure 4.2: A unit (an artificial neuron).

Each unit performs a simple computation: it receives signals from its inputs and computes a new activation level that it sends along each of its outputs (one in the case of figure 4.2). The computation is split into two components: the input function in_i that computes the weighted sum of the input values and the activation function that transforms the weighted sum to the final output value. Usually, all units in a network use the same kind of activation function.

A neuron can have any number of inputs from 1 to n, where n is the total number of inputs. The inputs may be represented as x_1, x_2, $x_3 \ldots$ x_n (as in figure 4.2). The corresponding weights for the inputs are w_1, w_2, $w_3 \ldots$ w_n. The summation of the weights multiplied by the inputs (the activation in_i) can be written as:

$$in_i = \sum_{i=0}^{n} w_i x_i, \text{ with } in_o = -1 \text{ and } w_0 = threshold$$

Two common activation functions are the sigmoid function and the step function. The step function (figure 4.3, left side) outputs a 1 when the threshold is reached (it fires), and it outputs a 0 when the threshold is not reached. This function is deducted from how the neurons in the brain work. The sigmoid function (figure 4.3, right side) outputs a value between 0 and 1, and the output value is relative to the difference between the threshold and the sum of the weighted inputs.

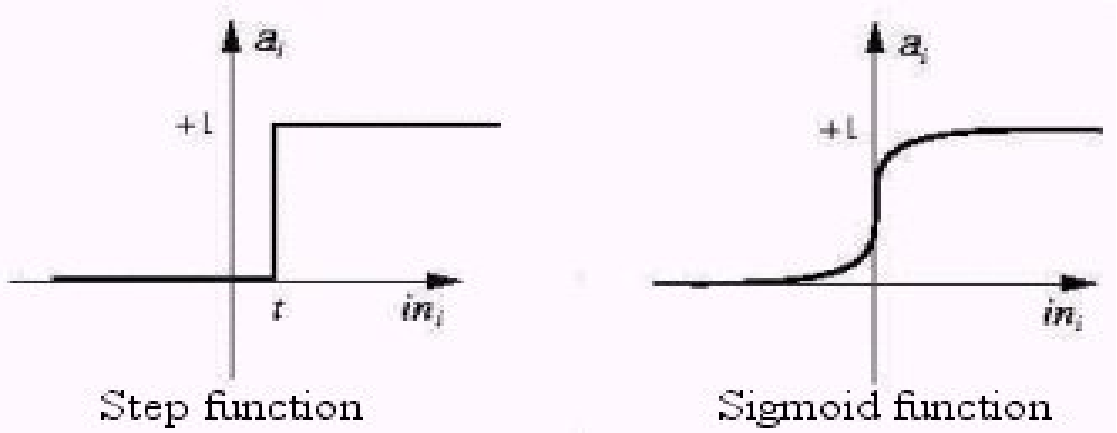


Figure 4.3: Two activation functions: on the left side the step function and on the right side the sigmoid function.

The units are used to build up a neural network. A popular way of building up a network is by organizing the units into a design called a *feed forward network* (see figure 4.4). It gets its name from the way the units in each layer feed their output forward to the next layer until the final output from the neural network is reached.

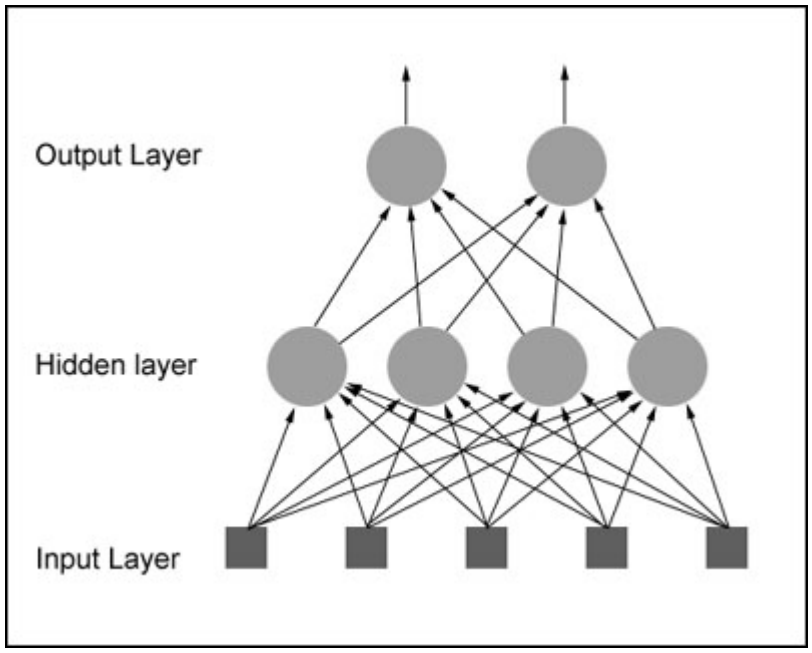


Figure 4.4: A feed forward network.

Each input is sent to every unit in the hidden layer and then each unit in the hidden layer sends its output to every neuron in the next layer. There can be any number of hidden layers within a feed forward network, and there can also be any number of units in each layer.

The neural network can also learn. Learning is done by feeding the network inputs, and checking whether the outputs it generates are correct. When they are correct nothing changes, but when they are wrong the weights in the network are slightly adjusted to reduce the error and ultimately to make the network generate the right outputs. A popular method for learning is *error back-propagation*. This learning algorithm divides the output error over all the units that contributed to the error.

A neural network can learn data but it is mainly used to recognize patterns in data, especially when humans cannot find any patterns. The pattern recognition makes neural networks sometimes a very good method for estimation purposes. We want to use the neural network to estimate the best move by teaching it the moves generated by a (search) algorithm.

4.2 Network size, input and output

A neural network should in principle be able to learn any (non-) linear function if the activation function is non-linear as well, but in practice that is not the case. It can happen that the network is extraordinarily large or that the training set is huge. This can be influenced by changing the input and output complexity of the network.

A neural network is actually a mixture of a database of input-output pairs and a pattern-recognition algorithm. So to learn well it needs a lot of data to learn from and to recognize the patterns in the data. To recognize these patterns it is recommended not to add too many ambiguous or unimportant data and to encode the data as simply as possible. An example: for a neural network to learn to recognize Dutch words it is not useful to also add the meaning of the words to the input data. But when the input (and/or output) is very large it usually means that the network will also be larger. The network should not become too large because that will result in large calculation times and 'over-learning'. 'Over learning' means that the network will be able to learn the input data but will not be able to recognize the patterns in it. Unfortunately there is no easy way to determine whether the network is too small or large.

In the following sections the variables in Cartagena will be mentioned as well as two possible representations of the input and output data for a neural network. The representations will then be compared and the neural network algorithm mentioned in section 5.3 will use the best representation.

4.2.1 Representation: variables and constants

The game state (input) has quite some variables:

- The board (there are 6!×2=1440 different ways to build-up the board). The Cartagena board is built up out of 6 parts that can be positioned in 2 ways.
- The game can be played with 2 to 5 players.

- The positions of the pirates on the board are variables and the number of pirates is six times the number of players.
- The cards that the players have.
- The card pile (for the Tortuga version).

To reduce the complexity, the board and the number of players will be kept constant. This is quite an obvious choice that will reduce the complexity considerably. The board will become the 'standard' board, which is also displayed on a piece of paper in the Cartagena box. The number of players will be 3 because the research interests lie in (simple) multi-player game play.

The variables of a move (output):

- There are 3 actions in one move.
- An action is 'move forwards', 'move backwards' or 'do nothing'.
- For each action there is one out of six pirates performing the action.
- When moving a pirate backwards the player receives one or two cards.
- When moving a pirate forwards the player has to play a card (which the player loses).

The maximum number of possible actions is 43 (6×6 'move forwards', 6 'move backwards' and 'do nothing') so the maximum number of possible moves is 43^3. To reduce the complexity we chose not to use one large network but to split up the problem into three (smaller) neural nets, which will learn the best action, not the best move, at a particular state. When the neural network can learn the first action it can probably also learn the next two actions with two other neural networks.

Furthermore, for an action it is not really interesting that the player loses a card, gets one or two cards or that a pirate changes position. That is because those 'variables' are totally dependent on the action of the chosen pirate. Therefore they are not included in the representation.

Two different ways to translate the game state and the best action were tried out. A small input and output for a small network but with a lot of encoded material, and a large input and output with all the information in very large chunks for a big network but hopefully easy to learn. These representations will be called the 'pirate position representation' and the 'board representation' respectively.

4.2.2 The pirate position representation

Input representation

For this small representation most variables are represented as a single value between 0 and 1. The game state is represented as follows:

- The position ID's of the pirates (ranging from 0 to 37, with 0 the start and 37 the boat)

- The number of cards that the players have of each kind
- The 10 ID's of the signs on the cards of the card stack

There are 6 pirates per player and 3 players, yielding 18 values. To reduce the possibility of having 1 state described in many ways the pirates are first sorted per player by their position, and the players are sorted by their move order (the moving player is put first, the next player second). It makes no difference whether pirate 1 or 5 is at position 10 as long as is mentioned that there is a pirate on position 10. The network only uses values between 0 and 1 so the values are scaled down by multiplying them with 0.02, with the exception when the ID is 37 (the sloop) that is scaled to the value of 1. The values are not scaled 'perfectly' (for instance by dividing the ID by 37) because the sloop is more important than the other spaces, and a neural network is not very perceptive of tiny differences when actually big differences are needed.

The number of cards that each player has is also represent able with 6 times 3 values. There are 18 cards of each type, but a player usually never has more than 10 cards of one type. Therefore the values are scaled down by dividing them by 10 with a maximum of 1.

The (Tortuga) card pile is represented in a totally different way. To represent the card pile it is not advisable to represent the card ID's by integer numbers and then scaling them [Reed and Marks, 1998], so they are represented in a binary fashion. This is probably not the best way but it is smaller that many other possible ways. This results in 3 bits per ID and 10 cardID's so 30 values for the card pile. This makes a total of 66 input values.

Why so many cards in the card stack as input? For one action (a move backwards) you can get a maximum of two cards, so the other eight cards seem redundant. This is not the case. The method that you are learning from uses those ten cards (and possibly even more cards!) to simulate multiple moves and getting cards from stack. So these cards are quite important to input in the neural network. For optimal learning the neural network should have the entire stack as input, or the search method should only look at the first ten cards and guess the next cards. But because at the moment we are looking at a maximum of 4-ply, not all ten cards will probably have been received yet.

Output representation

Now to define a small representation for an action:

- The pirate which makes the action can be represented by the pirate's ID
- The card that is played when moving forwards can be represented by its ID
- The type of action can be represented by an ID

The pirate ID ranges from 0 to 17 for three players. But these ID's are not really important in the game itself. Whether a pirate has ID 1 or 2 when it moves does not matter when the pirates are at the same space on the board. Because we decided to sort the pirates by ID (per player) we can use ID's 0 to 5 according to the position on the board. 3 bits can represent these 6 possible ID's.

There are six possible card ID's. Again 3 bits can represent this ID, just like with the pirate ID. When a pirate does not move forwards we use a value that is not in the range of the ID's, in this case 7 (111 binary).

The 'move forwards' and 'move backwards' are represented by 1 bit, and the 'do nothing' is encoded as a special case of the move forwards with pirateID 7 and cardID 7. In that case the move representation can be done with seven bits.

4.2.3 The board representation

Input representation

The game state is represented as follows:

- The pirate positions are represented by using all the positions on the board, with the number of pirates of every player on it.
- The number of cards that the players have of each kind is represented by their values.
- The 10 signs of the cards on the card stack.

The number of cards is represented in exactly the same way as in the pirate position representation, because it is of constant length and seems the best way.

The card stack is represented a little bit differently than in the pirate position representation. Instead of using bits to identify an ID, 6 values with always only one of them a 1 are used, depending on the ID. As mentioned in [Reed and Marks, 1999] this is more easily learnt by a neural network.

The most important difference from the pirate position representation is the representation of the position of the pirates. Here it is done by representing all the positions with the number of pirates on it for each player. This number is represented by three binary values. When there are no pirates of one color on a position, all three values are 0. When there is one pirate on it, the first value is 1. When there are two, the first two values are 1 and when there are three all three values are 1. The two exceptions are the start position and the boat, which are represented by six values (because there can be six pirates of one color on those positions).

This representation is used for all three players, so only for the positions of the pirates there are $3\times36+2\times6 = 120$ values for one player, thus 360 values for only the pirate positions for three players! Then there are still the 18 values for the cards and the 60 values for the card stack, for a total of 438 values. That is almost 7 times as many inputs as with the pirate position representation.

Output representation

Instead of encoding an action, we simply take all the possible actions and use one output value for every one of them. There are 6 different ways to move backwards (because there are 6 pirates to choose from), there a 36 different ways to move forwards (6 pirates times 6 types of cards) and then there is the 'do nothing'. So there are a total of 43 values in the output, with always only one of them a 1 and the rest 0's.

When the network learns perfectly, only one of the outputs will be a 1 and the rest 0's. Because a neural network usually does not learn perfectly, it is possible that the network will output more than one 1 or no 1's at all. This is not a problem, because the best action is the one with the highest output, and the other high valued outputs are very probably also good actions, but not the best ones. So you have a neural network that can calculate the best action, and other good actions as well.

4.2.4 The representations compared

The pirate position representation

The input and output size are quite small so the neural network will probably also be small. To find out how small a network should be is a matter of trial and error. There are some rules of thumb about the total number of neurons, but we just tried a couple of network sizes to see which size was best.

To see whether a network learns well a training set and a test set are required. The training set is used to train the network and the test set is used to test how well the neural network has learned by comparing the output of the network and the required output in the test set. Both sets were generated by the stand-alone version of Cartagena, mentioned in the implementation chapter, by letting three Not-So-Paranoid players (see Chapter 5) play against each other.

For the pirate position representation a training set of 5000 state-action combinations and a test set of 1000 state-action combinations was generated. The network was trained with the training set until the Mean Square Error (MSE) of the test set was minimal. The results are shown in table 4.1.

Sizes of the hidden layers	MSE
30x30	0.269791
40x40	0.280907
50x50	0.284828
60x60	0.285366
70x70	0.303954
80x80	0.310469
90x90	0.322217

Table 4.1: Training results for the pirate position representation

The MSE value on the right side of table 4.1 is the minimal Mean Square Error between the network output of the test set and the output that it should have generated. The best network (30x30) has an MSE of 0.27, and considering that a random network has an MSE of 0.5 and a perfect network an MSE of 0 we can conclude that this network performs poorly. Furthermore, testing has shown that the network cannot even reproduce its training set, so this representation is unusable.

The board representation

For the board representation we used a training set of 2000 state-action combinations and a test set of 500 state-action combinations. These sets are smaller than the sets of the pirate position representation because the larger network and the larger representation reduce the training speed a lot. The network was trained with the training set until the MSE of the test set was minimal. The results are shown in table 4.2.

Sizes of the hidden layers	MSE
100x100	0.0256
125x125	0.0233
135x135	0.0230
150x150	0.0249

Table 4.2: Training results for the board representation

The Mean Square Error with this representation is a lot smaller than with the pirate position representation. This could indicate that the network learns a lot better than with the previous representation, but that is not the case here. With an MSE of 0.0233 an average of 1 value out of 43 is wrong in every output, but when you construct a neural network that only outputs zeros you also have an MSE of about 0.0233!

This was unfortunately the case. Initially in more than half of the runs all the outputs were zeros (or at least below 0.0001), which is totally unusable. We therefore tested with many different kinds of outputs, for instance:

- We changed the 0 to 0.1 and the 1 to 0.9 in the output (we also tried 0.25 and 0.75)
- We trained with first with smaller and later with large training sets
- We left out the card row in the input
- We changed the 1 in the output to a higher value (5) to increase the influence on the MSE

Only the last change reduced the number of all-zero outputs. To see whether the outputs were still useful had to be investigated. The results are mentioned in section 5.3, where an algorithm is presented that tries to improve the search depth of search algorithms by using neural networks.

5. Algorithmic developments

The algorithms described in chapter 3 were insufficient for a reasonable level of play. Therefore new, better algorithms had to be researched:

1. The first algorithm that will be mentioned is derived from the Paranoid algorithm. The only difference is that the use of the evaluation function has been slightly altered for a less 'paranoid' play. It is therefore called the Not-So-Paranoid algorithm.
2. The second algorithm uses Monte Carlo. This is a well-known algorithm for use in a non-deterministic environment. Because the Tortuga version in Cartagena is deterministic, some non-determinism had to be introduced to apply this algorithm, as will be described later on.
3. The last algorithm uses Neural Networks to increase the search depth by estimating the best move at a particular time.

The evaluation functions for a single player used in all the algorithms are exactly the same. The differences lie in the search. The evaluation function used in Cartagena is simple and straightforward: it encourages the players to move forwards, take cards and win the game, but it is not tuned for good play.

Of course there are many more algorithms in the scientific world that can possibly be used in a multi-player game, but obviously there was not enough time to locate, describe and test them all.

5.1 Not-So-Paranoid algorithm

The power of the Paranoid algorithm lies in the fact that it can prune in a similar way as Alpha-Beta does, so it is quite fast, but because the Paranoid algorithm considers all the opponents as conspiring against the moving player the play is not very intuitive. This algorithm can possibly be improved by letting players look at their relative positions to other players and in that way determine where the real danger lies. For instance, helping a player that is behind you is not as bad as helping a player that is in front of you. The computation of the relative positions will probably increase the search time somewhat, but the hypothesis is that it will also make the players play better, as well as more intuitively.

The important difference when compared to the Paranoid algorithm is that we now need to look at the differences between and the values of the state evaluations of all the players. Furthermore, we need to look at the evaluations of all the players in the node, because all the players want to improve their own utility, unlike in the Paranoid algorithm.

Consider the *utility* value of a state for player *i* to be the value associated to that given state for player *i* (for instance, the evaluation value used by the Max^n algorithm), and the

evaluation value of a state for player *i* to be the value that the Not-So-Paranoid algorithm will use for player *i*. In the Not-So-Paranoid algorithm the influence of the utilities of players is increased when they are in front of the player for which the evaluation is calculated, and when they are behind, the influence of their utilities is decreased. The evaluation for player *i* looks like this:

$$evaluation_i = utility(player_i) - \sum_{j=1, j\neq i}^{nrofplayers} (influence_{i,j} \times utility(player_j))$$

The influence that a player has on the evaluation is:

$$influence_{i,j} = 1 - \frac{utility(player_i) - utility(player_j)}{utility(player_i)} = \frac{utility(player_j)}{utility(player_i)} \in [0, MAX_{utility}]$$

When player *j* is in front of player *i* the utility of player *j* will be higher than the utility of player *i*. The result of the *influence* function will then be higher than 1, so in the *evaluation* function the value of the *utility* of player *j* will be multiplied by a value higher than 1, which increases the influence of his value on the evaluation. When player *j* is behind player *i* the *influence* function will be lower than 1, so the influence of player *j*'s utility on the evaluation will be decreased. The evaluation function can be rewritten as:

$$evaluation_i = utility(player_i) - \sum_{j=1, j\neq i}^{nrofplayers} \frac{utility(player_j)^2}{utility(player_i)}$$

The quality of this evaluation function is very dependent on the behavior of the utility function. When all the utility values are always close to a large value then the influence of the other utilities is always about the same. To increase (or decrease) the influence of the other utilities the sum-part of the evaluation function can be multiplied by a value. Or even better, a function can be inserted that influences the relative utilities in a way that is best for the behavior of the utilities.

In the Not-So-Paranoid algorithm the players look at their utilities relative to the other players. A big difference with the Paranoid algorithm is that there is no longer only one value that has to be calculated. In the Paranoid algorithm every evaluation is done in the same way because there was always one player optimizing and the other players were trying to stop him. But now all the players are trying to optimize their own situations and at each state a player picks the best move for himself. Take a look at the Not-So-Paranoid search tree in figure 5.1. The situation is the same as in figure 3.4 (and 3.5), but now the evaluation values are replaced by the Not-So-Paranoid evaluation values with the values of figure 3.4 as input (as the utility values).

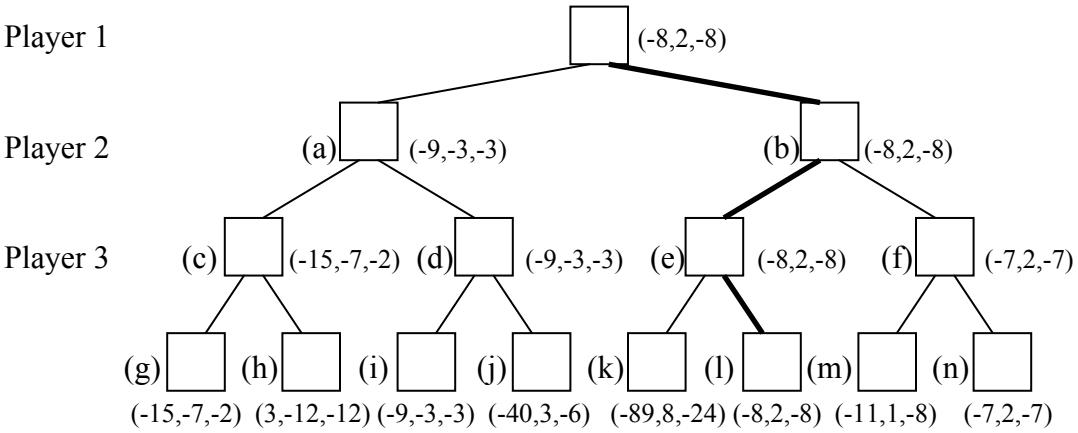


Figure 5.1: Not-So-Paranoid search tree example

As you can see the Not-So-Paranoid player picks a path different from the other algorithms. The Not-So-Paranoid player tries to maximize his relative position to the other players and assumes that the other players are trying the same thing. When looking at the paths that the players choose: Max^n chooses a-d-j with value (3, 9, 7), Paranoid chooses a-c-g with value (2, 3, 5) and Not-So-Paranoid chooses b-e-l with value (2, 4, 2). Max^n finds the highest score for himself (the first player), but relatively it is the worst choice. Paranoid is more conservative, but Not-So-Paranoid picks the best one relative to the other players.

The number of calculations in the Not-So-Paranoid algorithm is not much higher than the number of calculations in the Paranoid algorithm, but unfortunately using this kind of evaluation makes the pruning no longer evident.

The pruning problem in Not-So-Paranoid

In the Paranoid algorithm pruning can take place when the evaluation value is outside the bounds. In the Not-So-Paranoid algorithm these bounds are not so strong. See figure 5.1: when (e) is fully expanded then the lower bounds at (b) are (–9, 2, –∞). The –9 is the lower bound for player 1 because the current maximum for player 1 at the root is (–9, –3, –3), and the 2 for player 2 comes from the current maximum (–8, 2, –8). When the algorithm now reaches (f) and expands the first node there a value of –11 found for player 1 and a value of 1 for player 2. These values are lower than the lower bounds. Unfortunately the Not-So-Paranoid algorithm cannot prune here. The evaluation function multiplies the relative utilities with the utilities themselves, so there can be a situation with a better value for the deciding player, player 3 in this case, which has values above the lower bounds for the other players. This also happens to be the case in figure 5.1, where the second node of (f) has a higher value for player 3 and where the lower bounds are no longer exceeded.

This problem can be avoided when the evaluation function only looks at the relativity of the utility values and not at the relative utilities multiplied by the utility itself. This results in a different evaluation function:

$$evaluation_i = 1 - \sum_{j=1, j\neq i}^{nrofplayers} \frac{utility(player_j)}{utility(player_i)}$$

This evaluation function changes the behavior of the evaluation function slightly. It still reflects the quality of the position of a player, but when a player now finds a maximum and at the same time decreases the evaluations of the other players to below their lower bounds, then there is no longer another situation possible where the maximum for that player is higher and where the values of all the other players are above their lower bound. When one player's evaluation value increases then at least one of the other players' evaluations always decreases in value. See the example in figure 5.2. This figure uses the utility values of figure 3.4 so it represents the same situation as in 3.4, 3.5 and 5.1.

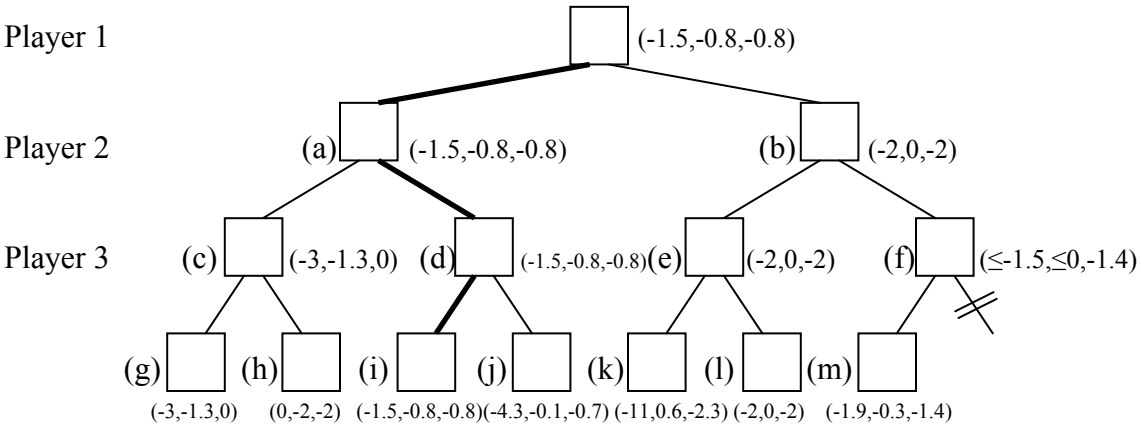


Figure 5.2: Not-So-Paranoid search tree example with pruning

In (f) the lower bounds are (–1.5, 0, –∞). The evaluation values of the first node for players 1 and 2 are below their lower bounds, so there is no action possible that leads to a state in which the evaluation for player 3 is higher and the evaluations for players 1 and 2 are above their lower bounds. Therefore pruning can take place at (f), without affecting the result.

Note that this version of the Not-So-Paranoid algorithm chooses an even different principal variation. It chooses a-d-i with value (4, 4, 5), which seems an even better choice for player 1 in this case.

With this new evaluation function pruning can take place if there are at least two lower bounds known. Generalizing: in an *n*-player game the Not-So-Paranoid player can prune when there are *n*–1 lower bounds known and the evaluation values are below the lower bounds. You could also prune when any lower bound is exceeded because it is not very common (in Cartagena for instance) that there is still a better possibility for the deciding player in which the lower bound is no longer exceeded. That is probably game-dependent

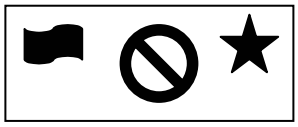
but it is included in the results. To distinguish the pruning differences in the Not-So-Paranoid algorithms the following name convention is introduced:

- Not-So-Paranoid$_{np}$: Not-So-Paranoid without pruning, using the first mentioned evaluation function (with the influence function).
- Not-So-Paranoid$_{p}$: Not-So-Paranoid with pruning, using only the relative differences of the utilities and pruning when all the lower bounds are exceeded.
- Not-So-Paranoid$_{ep}$: Not-So-Paranoid with extended pruning, again when only using the relative differences, and pruning when only one lower bound is exceeded as is described above.

5.2 Monte Carlo algorithm

Monte Carlo search is used in many games where there is a lot of unknown information, like the rolling of dice, unknown cards in someone's hand or cards on a card stack. In games the results are usually quite good. Therefore we considered this algorithm in Cartagena. For some more information see [CSEP, 1995] or search on the Internet where there is a lot of information about Monte Carlo available.

At first it was not quite obvious how Monte Carlo can be used in Cartagena. How Monte Carlo works is best explained with this very simple example:



You have a picture of which you want to calculate the size of the black areas. Monte Carlo takes a number of (for instance one thousand) random points in the picture and calculates the percentage of the random points that lie in a black area. It then calculates the total area and estimates the black area by taking the calculated percentage of the total area.

The use of Monte Carlo in Cartagena will be to pick a couple of possible moves and from there on play out a hypothetical game a number of times using a low-ply search. The move that results in the most wins is assumed to be the best move. But in Cartagena the results of a search are deterministic: there is always only one best move for one situation. This results in the fact that the hypothetical game always ends the same. To use Monte Carlo in search some non-determinism has to be introduced.

To introduce this non-determinism and to still be able to compare the algorithm to the other algorithms, the Monte Carlo algorithm first calculates the best 27 moves that are possible according to the evaluation function, just like the other algorithms. From those states hypothetical games are played out ten times. Each move in those hypothetical games is calculated by calculating the best three moves (1-ply search) according to the evaluation function (just like the other algorithms) and by picking one of them randomly. Then the games are almost always played differently.

The number of times a game is played out can be made a variable. The algorithm becomes faster when the game progresses further, so increasing the number of times a game is played out when the game progresses is a way to improve the algorithm. Using more moves to choose from can also improve the results but then the algorithms are more difficult to compare, so this was not implemented.

A good thing about the use of Monte Carlo is that it performs well when there is a lot of unknown information (non-determinism). In the Jamaica version of Cartagena there is a lot of unknown information, so this algorithm would probably outperform the other algorithms in that type of a game. Unfortunately there was no time to verify this.

5.3 Neural networks algorithm

In Cartagena the search depth is not very high, so the idea was developed to use neural networks to improve this search.

The idea is as follows: Assume that there is a method that can search *d*-ply deep, and that method returns the best move according to its evaluation function. A training set is generated by playing a number of games, and representing all the game states and their best moves as pairs. Then a neural network is constructed that these pairs are taught to. This neural network has as input the state of the game and as output the best move.

The neural network becomes a *d*-ply search algorithm estimator, and if this neural network learns well it can be used to improve the search depth of the learnt algorithm. With this neural network a new search algorithm can be constructed that first calculates all the possible moves for instance 1-ply deep and then uses the neural network to calculate the best moves the players can do from that point on. This results in an algorithm that can search $d+1$ ply deep. This new algorithm could then again be taught to a neural network etc. Note that this neural network will never be able to outperform the algorithm it tries to learn. A better solution is to collect data from real human players.

This algorithm could improve the search depth of the current algorithms considerably. The main problem is the learning. If the neural network is not capable to learn the algorithm well then the estimations will be poor and the idea will not work. But when the neural network learns well then the neural network can be used to give a good estimation of the best move, which can be used by new algorithms or as improvements of other algorithms.

Learning performance

The network was taught data generated by the Not-So-Paranoid$_{ep}$ algorithm in a 3-ply search. The data was represented in the ‘board representation’ way, as is described in section 4.2.3 and 4.2.4. See Table 5.1 for the results:

Number of actions checked	681
Number of nul-actions outputted (only zeros)	69
Number of actions outputted (number of ones)	1222
Number of outputs that contain the best action	144
Number of outputs that only contain the best action	98

Table 5.1: Neural network output results

Table 5.1 shows that the output of the neural network is in 10% of the times only zeros, which is not very bad. Furthermore, the network generates on average two actions and includes the action that it should output in 25% of the times. The question is whether the output is still good when the best move is not in the output. Further evaluation of the generated actions showed that in 53% of the times a generated action is an invalid action, which greatly reduces the usefulness of the networks.

From this we can conclude that these neural networks have a very bad chance of finding the best move. They reduce the branching factor but they increase the calculation time and greatly reduce the quality of the results. The question is whether the generated actions are as well as the actions that the network should have generated. This can only be discovered by evaluating games between the neural network algorithm and the algorithm it learnt from. Those results are presented in chapter seven.

6. Implementation

This chapter describes the implementation in Cartagena. The implementation includes the relations between the used classes, a user interface and a version that uses no interface, which will be further denoted as the stand-alone version.

The game is implemented in Delphi 5.0, because we wanted a programming language that was not too slow and not too hard to program in. Moreover, we wanted to build an object-oriented program (see below), for which Delphi is a good choice. The idea was to first implement the Tortuga version (with complete information) and later on see what the results are in the Jamaica version. Unfortunately, due to time shortage, the Jamaica version was never implemented.

The card row for the Tortuga version is implemented differently than is mentioned in the rules. The rules state that when the card row is depleted it is refreshed again, but that lessens the complete information of the game. Therefore, in the implementation, the card row is continually refilled to its full length and its size is reduced to 10.

Then there was the need to introduce a third type of action: the 'do nothing', because it is not practical to work with moves of varying length (1-3 actions). All the other rules are implemented in the same way as the rules state in section 2.1.

6.1 Cartagena classes

To enable testing and playing capabilities, and because in the beginning it was not yet known what had to be implemented in the end, an object-oriented program was constructed. The programming started with the construction of object-oriented classes that represented Cartagena.

To implement Cartagena in an object-oriented fashion the game had to be split up into all the different objects used in the game. First of all there is the Cartagena class that has all the methods involving game play like moving a pirate and taking a card. It also contains all the objects that the game uses. These are (in alphabetic order): Action, Card, CartagenaBoard, Pirate, Player and Space. These classes (objects) will shortly be described in this section.

Not all the methods and variables are mentioned (for instance the constructors and destructors are left out) to make everything easier to read and understand. The methods and variables that were left out are only interesting when reading the code, not when one wants to understand the structure.

Figure 6.1 shows a UML class diagram that reflects the relations between the Cartagena classes.

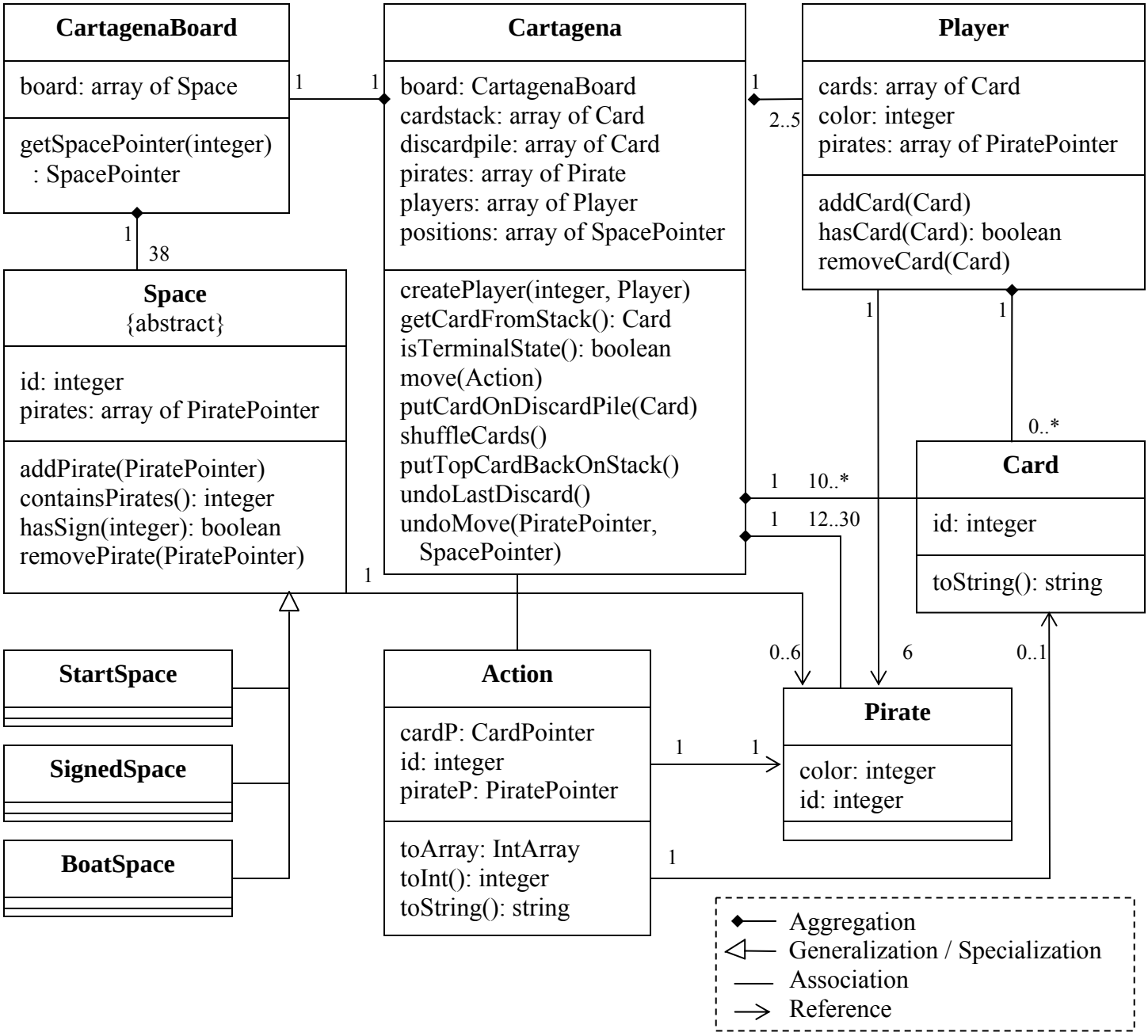


Figure 6.1: A UML class diagram of the Cartagena classes. For UML details see for instance [Eriksson and Penker, 1999].

The Action class

This class represents an action taken by a player. A player can take up to three actions in a round, so an action is either 'move pirate *p* forwards by playing card *c*', 'move pirate *p* backwards' or 'do nothing'. Whether the actions are valid is not relevant here.

The class contains three variables:

- *cardP*: a pointer to the card that will be played
- *id*: to identify the action. (0:'do nothing', 1:'move forwards' and 2:'move backwards')
- *pirateP*: a pointer to the pirate that has to be moved

At first glance the only useful methods here seem the constructor and the destructor, but there are also some useful methods that represent an action in another way. The methods:

- toArray(): returns a bit representation of the action.
- toInt(): returns an integer representation of the action.
- toString(): returns a string representation of the action.

The Card class

This class is an object that represents a Cartagena card. Such a card has a sign on it that can be a bottle, pistol, hat, skull, dagger or key. These signs are all represented by an integer (0=bottle, 1=pistol, 2=hat, 3=skull, 4=dagger, 5=key). The methods:

- toString(): returns a string representation of the card.

The CartagenaBoard class

The board is a collection of spaces, which will be described later in this section. We use the CartagenaBoard class to represent the board. It has the following method:

- getSpacePointer(*id*: integer): returns a pointer to the space with id *id*.

The Cartagena class

This class contains the functionality to play a game. It does not contain a user interface or any other medium to let someone really play. This class contains many variables (in alphabetic order):

- *board*: the playing board
- *cardstack*: the stack of cards from which the players take their cards (+ some indices)
- *discardpile*: the stack of cards where the discarded cards are put (+ some indices)
- *pirates*: the pirates of the players
- *players*: the players (+ numberofplayers)
- *positions*: an indication where the pirates are on the board

The methods are all the ingredients that are needed to enable play. Internally there are many methods for taking a card, moving a pirate, shuffling the deck and many more things, but only some relevant methods are mentioned here:

- createPlayer(*number*: integer; *p*: Player): changes player with id *number* to *p*. This is useful for changing a computer player to a different type of computer player.

- getCardFromStack(): returns the first card on the stack
- isTerminalState(): returns whether the game is in a terminal state
- move(*a*: Action): perform action *a*. If the move is illegal it won't be performed.
- putCardOnDiscardPile(*c*: Card): puts card *c* on the discard pile.
- shuffleCards(): shuffles the discard pile back into the card stack.

It turned out that it was also useful to include methods that can undo certain actions. Therefore the next couple of methods were included:

- putTopCardBackOnStack(): as long as shuffleCards() is not called this method undoes the last getCardFromStack().
- undoLastDiscard(): as long as shuffleCards() is not called this method undoes the last putCardOnDiscardPile().
- undoMove(*pp*: PiratePointer; *sp*: SpacePointer): puts pirate *pp* back on space *sp*. It does not undo a move, just the moving itself. The card handling is done with the putTopCardBackOnStack() and undoLastDiscard() methods.

The Cartagena class does not care whether someone performs ten actions in a row or not as long as the actions are valid. The user interface and the testing program both have to make sure that those rules of the game are not broken.

The Pirate class

This class represents a pirate. Pirate is a class that has only a constructor, an id and a color. A pirate has no link with his owner (a player), but the owner can identify which pirates are his. Moreover, a pirate knows nothing about the game it is playing: it is just a pawn. Variables:

- *color*: the color value of the owner of the pirate
- *id*: unique id of the pirate

The Player class

A player moves his pirates around and plays and collects cards. His variables are:

- *cards*: the cards that the player has
- *color*: the color of the player
- *pirates*: the pirates of the player

Because card handling is not entirely done by Cartagena the player also has to do some card handling. The player cannot move his own pirates by himself but has to call the move method of Cartagena with his pirates and cards as parameters. The methods are:

- addCard(*c*: Card): adds a card to the player's cards.
- hasCard(*c*: Card): returns whether the player has a card with the same sign as card *c*.
- removeCard(*c*: Card): removes card *c* (a card with the same sign as *c*).

The Player class is the base class for all the players, which are all the computer players. The first computer player was implemented in the CPlayer1 class (also a base class) and is described later in this section.

The Space class

The Space class represents a space on the board. The space can be a StartSpace, a SignedSpace or a BoatSpace. These three types of spaces all inherit from Space. Each space has an *id* and a collection of pointers to Pirates that are on the space. It also has three methods to identify what kind of space it is (these are overridden by its children), and, of course, the methods to add and remove a pirate. The methods are:

- addPirate(*p*: PiratePointer): Adds PiratePointer *p* to the space
- containsPirates(): returns the number of pirates on the space
- hasSign(*i*: integer): returns whether this space has sign *i* (see the Card class description for the sign encoding)
- removePirate(*p*: PiratePointer): remove PiratePointer *p* from the space

The Space does not check whether his Pirate limit has been exceeded, but according to the rules a SignedSpace, a space that is not the start or the sloop, can contain a maximum of three pirates. This rule is upheld by the move method of Cartagena.

6.2 Computer player classes

To implement various search algorithms the CPlayer1 class was constructed. That class is the base class for all the computer players, and thus the search algorithms. The CPlayer1 class evolved into many other CPlayer classes, all of which either added new useful methods or implemented a different kind of search.

The Node class was constructed as the basic element of which the search tree was constructed. A Node is a copy of the game state at a particular moment. Together with the CPlayer classes these are all that is required to implement a search algorithm.

When the speed and memory usage became bottlenecks a new Cartagena and CPlayer class were implemented called Cartagena_Int and CPlayer_Int. As the names already indicate these classes only use integers to represent all the elements of the game. Because of the use of integers, the extended use of objects is reduced to the changing of the length of integer arrays and the (much lower amount of) needed memory can be easily allocated beforehand. This results of course in much faster base classes for the search algorithms.

The Cplayer1 class

This class inherits from the Player class, and adds some statistical variables. These statistical variables are calculation times and the number of calculated moves. Therefore the important methods in CPlayer1 and its subclasses are listed here:

- calculateUtility(*np*: NodePointer): returns a value that represents how good the current state is of the player on the board. The higher the utility, the better the position.
- nextStep(*np*: NodePointer, *lastAction*: Action, *n*: integer): a recursive function that calculates the best action at a particular moment in a Move, and returns a move (or a part of it). The *n* indicates which action it is (the first, second or the third) and the *np* refers to the current state. The *lastAction* was added to prevent the calculation of moves that end in the same state.
- playturn(*np*: NodePointer): This method decides what actions are to be undertaken at a particular Node (by search) and plays them. Later the playturn was extended by adding search depth and width as parameters.

Some of the methods of the children of CPlayer1: (listed in the order of the time that they were created)

- calculatePossibleActions(*np*: NodePointer; *lastAction*: Action; *n*: integer): returns all the possible actions at Node *np* (the *n*th action, with *lastAction* as the last action).
- calculatePossibleMoves(*np*: NodePointer; *lastAction*: Action; *n*: integer): returns all the possible moves that can be performed at Node *np*. A *width* factor was added to bound the search width. calculatePossibleMoves uses calculatePossibleActions recursively.
- getBestActions(*aa*: ActionArray; *number*: integer): returns the *number* best actions of *aa* (as an ActionArray).
- createNode(*np*: NodePointer; *a*: Action): creates a copy of *np* and performs action *a* on it. The new Node is returned.
- undoMove(*np*: NodePointer; *a*: Action; *sp*: SpacePointer): undoes action *a* on state *np* by undoing all the Card changes and by returning the used Pirate to *sp*.
- move(*cp*: CartagenaPointer; *a*: Action): the same as createNode but then on a Cartagena object. This method was implemented to avoid the use of Nodes.
- undoMove(*cp*: CartagenaPointer; *a*: Action; *sp*: SpacePointer): the same as the undoMove with a NodePointer, but then on a Cartagena object.
- calculateOrder(*aa*: ActionArray; var *order*: IntArray): calculates the order of the elements in *aa* by utility and puts the order in *order*. This method was implemented to reduce possible memory loss and sorting time.

The Node class

A Node represents a state of the game. The Node is used to build up and destroy search trees. Its variables are:

- *cartaPointer*: a pointer to a Cartagena object

- *ootc*: the one or two cards that were received by a 'move backwards'. This is stored as a safety precaution.
- *previousNode*: a pointer to the previous game state

The only useful methods of Node are the constructors:
- constructor(*cp*: CartagenaPointer): creates a Node that has the copy of the state of *cp*. For this Node a copy of *cp* is made and the cartaPointer of the Node refers to this copy.
- constructor(*np*: NodePointer; *a*: Action): creates a new Node from state *np* by performing action *a* on it.

The Cartagena_Int class

This class was implemented in the last period of the research to increase the speed and reduce the memory usage of the algorithms. This class represents, as previously mentioned, a Cartagena class but then with only integers and no objects. The computer players that use this class are derived from CPlayer_Int, which was implemented purely for the use of Cartagena_Int.

This integer version of Cartagena uses the following variables:
- *boardSigns*: integer representations of the signs on the board
- *cardsOfPlayers*: 2-dimensional array with the number of cards of all players of every cardtype
- *cPointer*: a pointer to a 'real' Cartagena object, which is the original of this representation
- *cardstack*, *stackpointer*, *stacksize*: integer representation of the card row
- *discardpile*, *discardSize*: integer representation of the discard pile
- *numberofplayers*: the number of players of this game
- *positions*: id values of the space on which every pirate currently is located
- *spaceContainsPirates*: an array with the number of pirates on all the spaces

The methods are:
- calculateUtility(*weights*: UtilityWeights): calculates the utility of the game state. The *weights* are the weights for all the variables, and are player-dependent. constructor(*cp*: CartagenaPointer): creates a Cartagena_Int representation of *cp*.
- isTerminalState(): returns whether the game is in a terminal state.
- move(*actionID*, *n*: integer): performs the action with integer representation *actionID* at depth *n*.
- undomove(*actionID*, *lastposition*, *n*: integer): undoes the action *actionID* at depth *n* and moves the pirate back to the position with id *lastposition*.
- shuffle(): shuffles the discard pile into the card stack.

The Cplayer_Int class

This is the computer player class that uses the Cartagena_Int class instead of the Cartagena class. It inherits all the variables from CPlayer1 (just like a 'normal' computer player would), but it has extra methods that only use Cartagena_Int objects. The methods are:

- calculatePossibleActions(*cp_int*: Cartagena_intPointer; *lastMove*, *n*: integer; var *actions*, *utilities*: IntArray; var *numberOfActions*: integer): calculates all the possible actions at state *cp_int* and puts them in *actions*.
- calculatePossibleMoves(*cp_int*: Cartagena_intPointer; *lastMove*, *n*, width: integer; var *indexinmoves*: integer; var *moves*, *actions*, *actionUtilities*: doubleIntArray; var *numberOfActions*, *indexInActions*, *moveUtilities*: intArray): calculates the best moves searching with width *width* by recursively calling calculatePossibleActions.
- calculateUtility(*cp_int*: Cartagena_intPointer): calculates the utility for this player of *cp_int*.
- sort(var *actions*, *utilities*: IntArray; *numberOfActions*: integer): sorts the *actions* by their *utilities*.
- sort(var *moves*: DoubleIntArray; var *utilities*: IntArray; *numberOfActions*: integer): sorts the *moves* by their *utilities*.

6.3 User interface

The base classes of Cartagena enable play but do not implement it. A user interface was implemented to evaluate the results and to enable a human player to play against the computer.

The user interface enables to play or playtest a game with 2 to 4 players. The type of players can be selected when starting a new game. You can let the computer play for any player but only with a pre-programmed algorithm, and at any time you can force moves for anyone because the program was meant as a testing environment (and not a commercial game).

The only possible play-type is the Tortuga version, so all the cards of the players are visible in a table, and the card stack is displayed below the board. To move a pirate you only have to click on it and then click on the place where it should go. If the move is illegal it will not be performed and an error message will be displayed. See figure 2.2 for a screenshot of the interface:

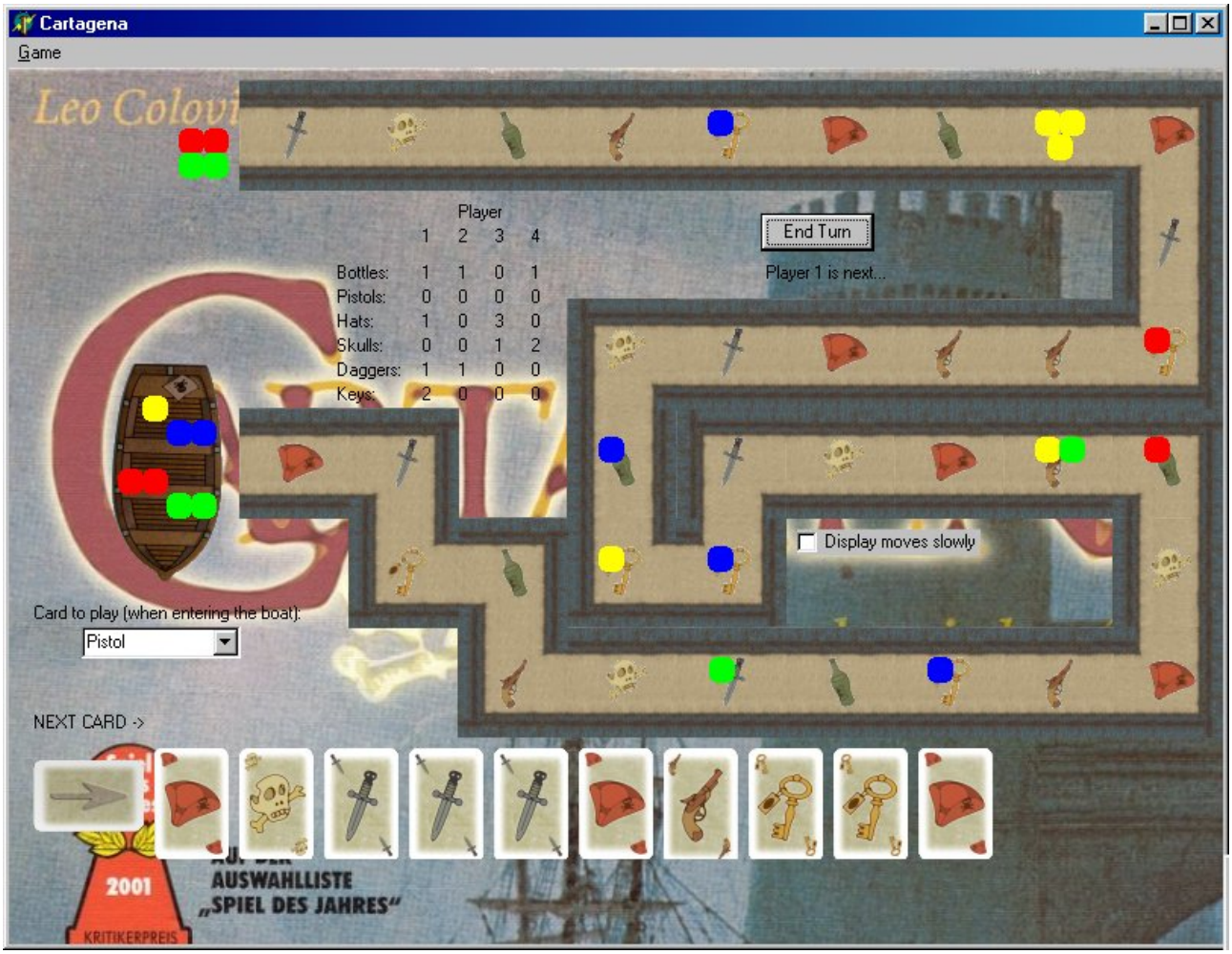


Figure 2.2: a screenshot of the user interface

There is also the possibility to let only computer players play so you can evaluate their strategies. The screen refresh takes a lot of time and it is unnecessary when you want to run games during the night. For that purpose a so-called stand-alone version was implemented.

6.4 Stand-alone version

For fast game play of hundreds of games a stand-alone version was implemented. It is called PlayCartagena and has several options. It can:

- write what player won a game to a file
- write all the actions played in a game to a file
- write the number of moves and calculation times to a file
- limit the number of moves that are played in a game (to avoid infinite loops)
- write the evaluation values of a game to a file
- play with 2 to 5 computer players
- compare neural network output to the moves of a player (e.g. the player to learn from)

The stand-alone version reads the command line parameters and creates a game with these parameters. Examples of some common parameters are ‘paranoid’ and ‘maxn’ for a Paranoid and Max^n player respectively, a number to limit the number of moves in a game per player and ‘-xt’ to write all the game state and move pairs to a file so it can be fed to a neural network (in Matlab).

This version was implemented purely for the play of hundreds of games without the use of an interface, and to have a program that can easily generate all kinds of statistical data about the players. All the results in the next two chapters were created with the stand-alone version of Cartagena.

7. Experimental results

This chapter describes the efficiency and performance results of the Max^n, Paranoid, Not-So-Paranoid, Monte Carlo and Neural Network algorithms. The results include the calculation times, the number of calculated moves, evaluation function behavior and the number of times an algorithm won.

7.1 Max^n, Paranoid and Not-So-Paranoid

In this section the results of Max^n, Paranoid and Not-So-Paranoid algorithms are described. These three algorithms are well comparable due to the fact that they all build up a search tree. Note that these algorithms use the same search width (maximum branching factor) of 27 and that they all use the same utility function to determine the best three actions at each part of a move. The only difference is the choice of the best move (the evaluation function).

7.1.1 Efficiency

Efficiency is very important in search. When an algorithm is fast it can search deeper and give better results. The Max^n algorithm is slower than the Paranoid algorithm because it has to expand the entire search tree. The Not-So-Paranoid$_n$ algorithm is as fast as the Paranoid algorithm, even though it uses more values in its evaluation function. Figure 7.1 displays the calculation times of the Max^n, Paranoid and Not-So-Paranoid$_p$ algorithms.

Figure 7.1 shows that the Max^n and Not-So-Paranoid$_p$ algorithms take about the same time in 3-ply, but the Paranoid algorithm already prunes a lot which reduces his search time. In 4-ply the pruning improves the calculation times for the Paranoid and Not-So-Paranoid algorithms. Because the Not-So-Paranoid$_p$ algorithm uses three values in its evaluation function instead of one for the Paranoid, it is still slower than the Paranoid in 4-ply. When searching more than 4 ply deep the calculation time of the Not-So-Paranoid$_p$ algorithm increases more then the calculation time of the Paranoid algorithm.

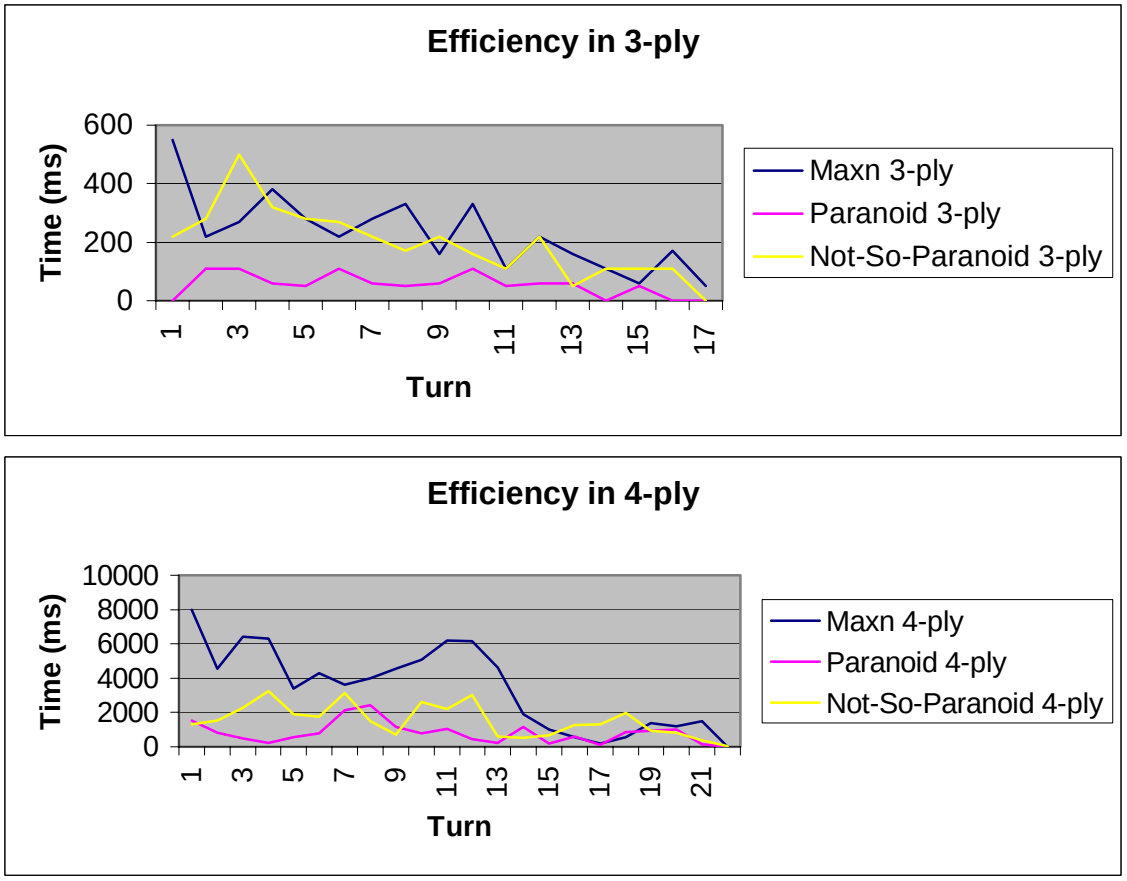


Figure 7.1: Calculation times in a 3-ply and 4-ply search for Max^n, Paranoid and Not-So-Paranoid$_p$ in a typical game

To see how efficient the algorithms are on average, the algorithms are compared in Table 7.1 in a 3-ply and 4-ply search.

	3-Ply		4-Ply	
	Moves	Time	Moves	Time
Max^n	521	206	11834	4320
Paranoid	159	71	3671	1391
Not-So-Paranoid$_p$	531	179	3807	1490

Table 7.1: Average number of calculated moves and calculation times over several games for Max^n, Paranoid and Not-So-Paranoid$_p$ in 3-ply and 4-ply

Table 7.1 shows the result of several 3-player games between a Max^n, Paranoid and Not-So-Paranoid$_p$ algorithms. The number of calculated moves for the Paranoid and Not-So-Paranoid$_p$ algorithms clearly decreases in 4-ply compared to the Max^n algorithm. The Max^n algorithm approaches the maximum number of calculated moves, as is expected because it does not prune, while the Paranoid and the Not-So-Paranoid$_p$ algorithms calculate about one third of the number of moves that Max^n calculates.

The Not-So-Paranoid$_{ep}$ algorithm was mentioned briefly in section 5.1 and is of an entirely different class. The Not-So-Paranoid with extended pruning prunes about 95 percent of the moves that the Not-So-Paranoid$_p$ algorithm calculates. See table 7.2 for the results of a game of three Not-So-Paranoid players with different levels of pruning.

	3-Ply		4-Ply	
	Moves	Time	Moves	Time
Not-So-Paranoid$_{np}$	521	175	12241	4678
Not-So-Paranoid$_p$	497	185	4156	1805
Not-So-Paranoid$_{ep}$	72	32	183	101

Table 7.2: Average number of calculated moves and calculation times for the Not-So-Paranoid algorithm without pruning, with pruning and with extended pruning

The data in table 7.2 is the result of several 3-player games between the different Not-So-Paranoid players. As can be easily seen in table 7.2, the pruning of the Not-So-Paranoid$_{ep}$ algorithm is much better than the pruning of the normal Not-So-Paranoid algorithm. The extended pruning version prunes more than 98 percent of the tree while the normal version prunes about 70 percent in a 4-ply search. The search time reductions are similar.

7.1.2 Performance

Better calculation times are always nice, but the more important thing is whether the algorithm performs well in a game. To test this we first let the Paranoid, Not-So-Paranoid$_p$ and Max^n algorithms play against one another in 3-player games with 3-ply search.

	Performance
Max^n	25%
Paranoid	29%
Not-So-Paranoid$_p$	46%

Table 7.3: Performance (chances of winning) of 123 games between Max^n, Paranoid and Not-So-Paranoid$_p$ using a 3-ply search

As you can see in Table 7.3 it does not pay off to only look at yourself as Max^n does. Playing paranoid is a little bit worse but improving your relative position offers the best chances of winning. The Not-So-Paranoid$_p$ algorithm is the best one (of the three above) and the fastest when searching deeper. The Paranoid algorithm may seem worse than the Max^n algorithm but it is faster so it can look deeper in the same time and thereby improve the results, as can be seen later in this section.

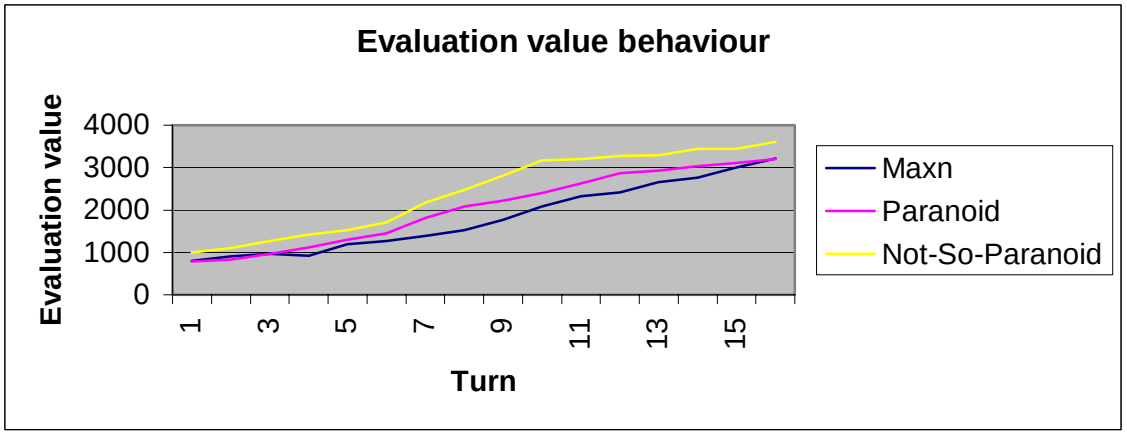


Figure 7.2: The evaluation value in a game between Max^n, Paranoid and Not-So-Paranoid$_p$ (using 3-ply search)

In figure 7.2 the evaluation function values in a game are shown. These are not the evaluation values of the players, but the utility values as described in section 5.1. The evaluation value of the Not-So-Paranoid$_p$ player is always higher than the values of the other players. This means that his position is always better than the position of the other players (assuming that the evaluation function is not very bad), which increases his chances of winning.

We will now look at the different pruning versions of the Not-So-Paranoid algorithm. The Not-So-Paranoid$_{ep}$ is much faster than Not-So-Paranoid$_p$, but the question is whether the Not-So-Paranoid$_{ep}$ does not sometimes mistakenly prune candidates for the best move. To test this we let the Not-So-Paranoid algorithms play against each other in 3-player games.

	3-players (3-ply) Performance	2-players (4-ply) Performance
Not-So-Paranoid$_{np}$	38%	-
Not-So-Paranoid$_p$	31%	50%
Not-So-Paranoid$_{ep}$	31%	50%

Table 7.4: Performance (chances of winning) of 236 games between the Not-So-Paranoid algorithms in 3-player games (the 2-player games were actually 3-player games with only always 2 algorithms of the same type, and only the algorithms with pruning)

Table 7.4 shows the result of several hundreds of games between the three Not-So-Paranoid algorithms, and on the right side 3-player games with only the Not-So-Paranoid algorithms with pruning. We can see that the Not-So-Paranoid algorithms with pruning are beaten by the non-pruning version. This is not caused by over-pruning but by the difference in the evaluation functions. The evaluation function of the non-pruning version just happens to be better than the evaluation function of the other two.

When comparing the two Not-So-Paranoid algorithms with pruning one can see that they perform equally well. In the 3-player games they both win 31% of the games and when

playing against each other they both win 50% of the games. The Not-So-Paranoid$_{ep}$ algorithm is therefore not pruning too much, plays as well as the Not-So-Paranoid$_{p}$ and is much faster. The performance of the algorithms when giving them the same amount of time is predictable, but is shown in table 7.5:

	Performance
Max^{n} 4-ply	12%
Paranoid 5-ply	19%
Not-So-Paranoid$_{ep}$ 7-ply	69%

Table 7.5: Performance of the Max^{n}, Paranoid and Not-So-Paranoid$_{ep}$ algorithm in 3-player games when they use about the same amount of time

The Not-So-Paranoid$_{ep}$ algorithm is clearly the best algorithm when building up a search tree using a bounded amount of time (as was to be expected from the results in table 7.3). The algorithm prunes so much that it can search much deeper in the same time as the other algorithms. The Not-So-Paranoid$_{ep}$ algorithm therefore outclasses the Max^{n} and Paranoid algorithms in terms of performance, efficiency and quality.

In the next section the Not-So-Paranoid$_{ep}$ algorithm will be competing against the Monte Carlo algorithm and the algorithm that uses neural networks with a bounded calculation time.

7.2 Monte Carlo and the Neural Networks

The previous section described the Max^{n}, Paranoid and Not-So-Paranoid$_{n}$ algorithms. In this section the results of the Monte Carlo and Neural Network algorithms compared to the Not-So-Paranoid$_{ep}$ algorithm will be given.

7.2.1 Efficiency

The (efficiency) behavior of the Monte Carlo algorithm can best be shown in figure 7.3, which is the result of a 3-player game with the Monte Carlo algorithm and a 5-ply and 6-ply Not-So-Paranoid$_{ep}$ algorithm.

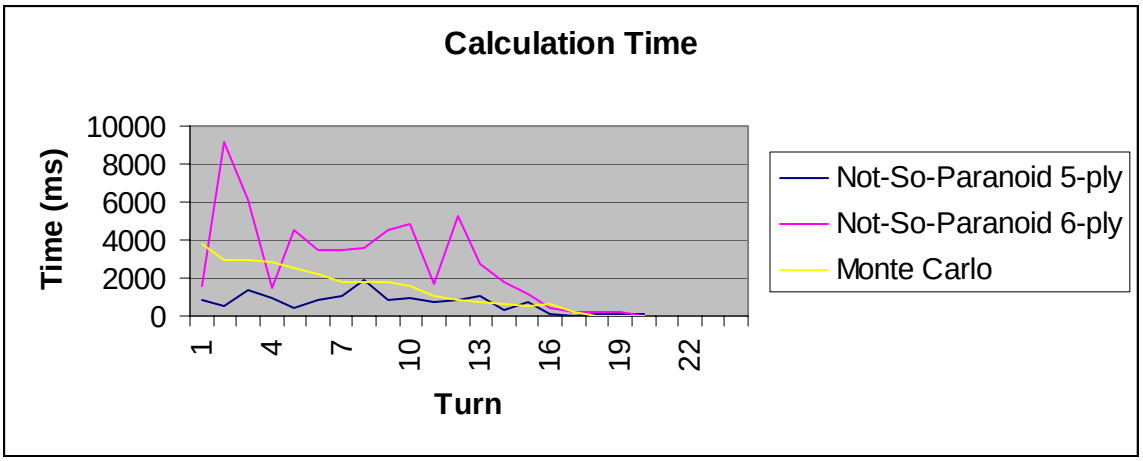


Figure 7.3: Monte Carlo calculation time in a 3-player game together with a 5-ply and 6-ply Not-So-Paranoid player with extended pruning

The calculation time of the Monte Carlo algorithm is linear decreasing while the calculation time of the Not-So-Paranoid$_{ep}$ algorithm is (almost) constant. The peek in figure 7.3 around turn 2 is the result of the enormous amount of possible actions at that point in the game, and the fact that the extra ply deeper means 27 times more move calculations. The algorithm reduces the amount of moves to choose from, but all the possible actions still have to be calculated.

We trained the Neural Network algorithm to learn the moves generated by the Not-So-Paranoid algorithm. Because the neural networks sometimes output only zeros, the algorithm that was mentioned in chapter 6 was slightly adjusted. It now uses the Not-So-Paranoid algorithm to generate the best possible actions when a neural network outputs nothing or only illegal moves. This increases the search time but enables the algorithm to always output useful moves. The efficiency results are shown in table 7.6.

	Efficiency (3-ply)	Efficiency (6-ply)
Not-So-Paranoid$_{ep}$	35 ms	2916 ms
Neural Networks	953 ms	20812 ms

Table 7.6: The average move calculation time of the Not-So-Paranoid$_{ep}$ algorithm and the Neural Network algorithm in a 3-ply and 6-ply search

The idea behind the Neural Network algorithm was that its speed could be used to increase the search depth. Unfortunately it is too slow to be properly used for that purpose. If the algorithm's performance is nearly as good as its teacher's then the algorithm can possibly still be used when searching without a limited search width and a large branching factor. The performance is presented in the next subsection.

7.2.2 Performance

To check whether the Neural Network algorithm performs as well as the Not-So-Paranoid$_{ep}$ algorithm that it learnt from, we let them play against each other. The results are shown in table 7.7.

	Performance (3-ply)
Not-So-Paranoid$_{ep}$	96%
Neural Networks	4%

Table 7.7: Performance of the Not-So-Paranoid$_{ep}$ algorithm and the Neural Network algorithm in 3-ply games, when the neural network was taught the moves of the Not-So-Paranoid$_{ep}$ opponent

The neural networks clearly do not learn good enough to be used properly. The Neural Network algorithm is outperformed when playing against its teacher, while searching to the same depth. Unfortunately, the performance and the efficiency of the neural networks are just too bad to enable the algorithm to be used as a way to increase the search depth of a search algorithm.

This leaves us with the Monte Carlo algorithm. The Monte Carlo algorithm is very efficient and performs better than the Paranoid and Max^n algorithms, as can be seen in table 7.8. Because the Monte Carlo algorithm has a linearly decreasing search time we increase during the number of Monte Carlo simulations while the game progresses, in order to have a more-or-less constant computation time.

	Percentage of games won
Max^n (3-ply)	8%
Paranoid (4-ply)	9%
Monte Carlo (adjusted)	83%

Table 7.9: Performance of the Max^n, Paranoid and Monte Carlo algorithms when playing 118 3-player games, when using about the same calculation time.

The Monte Carlo algorithm performs even better against the Max^n and Paranoid algorithms than the Not-So-Paranoid$_{ep}$ algorithm does. We now let the Monte Carlo algorithm play against the Not-So-Paranoid$_{ep}$ algorithm using a bounded search time. Table 7.9 shows the results.

	Percentage of games won
Not-So-Paranoid$_{ep}$ 6-ply	60%
Monte Carlo (adjusted)	40%

Table 7.9: Performance of the Not-So-Paranoid$_{ep}$ algorithm and the Monte Carlo algorithm in 3-player games when using about the same amount of time

The Not-So-Paranoid$_{ep}$ algorithm is clearly better than the Monte Carlo algorithm. When playing 3-player games with only Not-So-Paranoid$_{ep}$ and Monte Carlo players, a Not-So-Paranoid$_{ep}$ player wins 60% of the games. The Monte Carlo algorithm is a little bit faster,

but performs worse. This makes the Not-So-Paranoid$_{ep}$ algorithm the best performing algorithm of all the algorithms mentioned in this document.

The reason of the good results of the Not-So-Paranoid$_{ep}$ algorithm are twofold: It is the most efficient algorithm so it can search deeper than the other algorithms, and its strategy is better than those of the other algorithms that build up a search tree.

8. Conclusions

This document describes five types of multi-player search algorithms in a (board) game with perfect information. Two of them were already published and three of them were developed during the course of this thesis. The developed Not-So-Paranoid algorithm turned out to be a very good algorithm. It outperformed the other algorithms in the fields of efficiency, performance and quality. The developed algorithm that uses Monte Carlo search also performed quite well even though it does not really use a strategy to search like the other search algorithms. It will probably function better in a game with no perfect information.

The search improvement by using neural networks was unfortunately a failure. The neural networks cannot learn well and take a lot of time to calculate a move. The neural networks need to learn very well to be able to be used, and are potentially useful when a large branching factor is involved and not a limited (or small) branching factor, as was done in our case for comparison reasons.

Future research could prove whether the Not-So-Paranoid algorithm performs as well in other games as it did in Cartagena. Also the performance of the Not-So-Paranoid algorithm in games with no perfect information and its performance compared to the Monte Carlo algorithm in such circumstances could be investigated. In the case of the neural networks, a study could be done to determine when it pays off to use neural networks to learn the best move in games, or to prove that they are unsuited to be used in multi-player games.

References

[CSEP, 1995]: 'Introduction to Monte Carlo Methods', Computational Science Education Project 1995. http://csep1.phy.ornl.gov/mc/mc.html.

[Eriksson and Penker, 1999]: Hans-Erik Eriksson and Magnus Penker: 'De UML Toolkit' (Dutch). Copyright 1999 Academic Service, ISBN 9039510156.

[Marsland, 1986]: Tony Marsland: 'A review of game-tree pruning'. ICCA Journal vol. 9, no. 1, pp. 3-19, 1986.

[Reed and Marks, 1999]: Russell D. Reed and Robert J. Marks II: 'Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks'. Copyright 1999 Massachussetts Institute of Technology, ISBN0262181908

[Ross, 2002]: Don Ross: 'Game Theory', Stanford Encyclopedia of Philosophy, Copyright 2002. http://csep1.phy.ornl.gov/mc/mc.html.

[Russell and Norvig, 1995]: Stuart Russell, Peter Norvig: 'Artificial Intelligence, a Modern Approach'. Prentice Hall 1995, ISBN0133601242

[Sturtevant, 2002]: Nathan Sturtevant: 'A comparison of Algorithms for Multi-Player Games'. UCLA, Computer Science Department.

[Sturtevant and Korf, 2000]: Nathan R. Sturtevant and Richard E. Korf: 'On Pruning Techniques for Multi-Player Games'. Computer Science department of the University of California, Los Angeles. Copyright 2000, American Association for Artificial Intelligence.