Internal Report 2010–19 December 2010

Universiteit Leiden
Opleiding Informatica

Discovering Intelligence

in a

Natural Environment

Zeki Karaca

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Discovering Intelligence in a Natural Environment

By

Zeki Karaca

Supervisor: Dr. Walter Kosters

December 2010

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University

Abstract

This experiment was performed to find intelligence in a natural environment. We used a game setting with pawns belonging to players. Pawns used programs which defined their behaviour during the game. Programs could be changed by players to create better playing pawns. We compared different syles of cloning and mutations to increase the performance of pawns. We found an increase in performance during several experiments. With mutations disabled these performances stagnated to a certain value. Even when using mutations by deleting parts of a program resulted in stagnated performances. Mutations by adding random commands to a program showed the best results.

Contents

1 Introduction **1**

2 Related work **3**

2.1 Core War 3

2.2 RoboCom 4

3 Methods **6**

3.1 The gameboard 6

3.2 Players 7

3.3 Pawns 8

3.4 Programs 9

3.4.1 Cloning 10

3.4.2 Mutating 11

3.5 Moves 12

3.5.1 Move 13

3.5.2 Attack 13

3.5.3 Boolean test 16

3.5.4 Jump 17

3.6 Scores 17

3.6.1 Distance Only 18

3.6.2 Distance and Power 19

3.6.3 Distance and Power reversed 20

3.6.4 Bonus scores 21

4 Results and Experiments **22**

4.1 Game domination by cloning pawns 22

4.2 The effect of mutation by deleting 27

4.3 Comparing different muation styles 30

4.4 Finding good performing programs 31

5 Conclusions **35**

Bibliography **37**

1 Introduction

Artificial Intelligence is often defined as the study and design of *intelligent agents* (see [1] and [2]). An intelligent agent is a system that learns from experience and adjusts its behaviour to perform better. The overall aim of this case study is to define a natural environment, where simple programs can collaborate and compete. We study their behaviour, and hope to discover intelligence in the evolving structures.

An intelligent agent usually observes its environment and acts upon it. The agents during our research act on a fully deterministic environment. Each state of the environment depends on the preceding state of the environment and the actions taken by the agent. No interference or uncertainty will occur.

We will be using a kind of game to do our research where different players can compete with each other. For the sake of simplicity, we use a small square tiled board to play the game. We will let several simple agents compete with each other and give them the possibility to adjust their behaviour to perform better. One of the questions we hope to answer during this case study is whether we will find an increase in performance for some of the agents based on the things they have "learned".

Assume we have an $n \times n$ gameboard with a number of pawns on it, each belonging to a specific player. Players possess power which is spread amongst their pawns. All players start with an equal number of pawns and an equal amount of power. In agent related research, power is often referred to as energy or strength. Each pawn has a program containing a sequence of moves it will make during one round of the game. A move can be simplistic, e.g., just moving a pawn in a certain direction, or it can be more complicated, like attacking another pawn by taking some of its power.

At start, all the players' pawns are lined up at one side of the squared board. Their goal is to reach the opposite site. When moving across the board, pawns can be eliminated if their power becomes zero or less. This will obviously decrease a players' total power. After every round of moves, the players can change the programs of their pawns to perform better during the next round. An overall score will be calculated for each player, depending on the number

of pawns that reached their goal, their total amount of power, and the number of pawns that are still "alive". The player with the highest score will be the winner of the round. Eventually, an increase in score during consecutive rounds could mean that a player has "learned" to perform better.

This Bachelor's thesis is conducted under supervision of dr. W.A. Kosters at the Leiden Institute of Advanced Computer Science (LIACS) from Leiden University. This report shows our findings during our research. First, we will specify the game environment in greater detail. Then we will briefly explain some of the design and implementation decisions that were made. Finally, we evaluate some of our experiments and discuss the obtained results.

2 Related work

2.1 Core War

In 1984 A.K. Dewdney, a computer scientist and former columnist for Scientific American, wrote an article about a game called *Core War* (see [3, 4, 5, 6]) a programming game in which two or more programs compete for the control of a virtual computer. These programs, known as warriors, are written in *Redcode*, a low-level language similar to assembly. The goal of a warrior is to eliminate all opponents by terminating their processes and gaining sole control over the virtual memory.

The most important parts of Redcode are the easiest ones. The simplest, and probably the first, Core War program is the *Imp*, published by A. K. Dewdney in the original 1984 Scientific American article that first introduced Core War to the public. Figure 1 shows this program.

```
MOV 0, 1          ; a simple move instruction
```

Figure 1: Simplest Redcode example; The Imp

As we can see, the Imp consists of just one MOV which copies an instruction forward, just after itself as illustrated in Figure 2. The instruction it just wrote will be executed. Then it will once again copy itself one instruction forward, execute the copy, and continue to move forward repeatedly.

```
MOV 0, 1          ; this was just executed
MOV 0, 1          ; this instruction will be executed next
```

Figure 2: The Imp copies itself

2.2 RoboCom

Dennis Bemmann developed *RoboCom* (see [7]), a kind of simplified version of Core War, where programs (robots) compete on a $n \times n$ chess-like board. Like in Core War, the goal of a robot is to eliminate the opponents' robots while staying functional. There are no weapons: the only way a robot can manipulate other robots is by transferring code to them.

A robot uses an instruction set indicating which instructions it can execute. There are three instruction sets: *basic*, *advanced* and *super*. The instructions in each set are listed below. Note that the super set contains all possible instructions.

- **Basic (0)** ADD, BJUMP, COMP, DIE, JUMP, MOVE, SET, SUB, TURN
- **Advanced (1)** Basic instructions plus SCAN and TRANS
- **Super (2)** Advanced instructions plus CREATE.

The basic instruction set contains all the elements necessary for a program to run, plus arithmetic operations and movement routines. This set contains almost all instructions but robots using this instruction set cannot communicate with other robots.

The advanced instruction set contains all instructions of the basic set with the addition of instructions to communicate with *the reference field*, i.e., the field a robot is facing. Each robot has a direction which determines which field is the reference field. If the reference field contains another robot then communication with that robot is possible. The additional instructions in this set are SCAN and TRANS.

The super instruction set contains all possible instructions. In addition to the advanced set it contains an extra instruction, i.e., the CREATE instruction, which allows a robot to build more robots.

The program of a robot is organized in so-called banks. A code bank can contain any number of statements and each robot can possess up to 50 banks.

```
; RoboCom program
Name Mine

Bank Mine          ; First bank, containing the program
@Loop              ; Label at the beginning of the loop
Turn 1             ; Turn to right
Set %Active, 0     ; Deactivate potential opponent
Scan #1            ; Scan reference field
Comp #1, 1         ; Actually an opponent?
Jump @Loop         ; No, keep turning
Trans 2, 1         ; Yes, transfer bank 2!
Set %Active, 1     ; Re-activate and let it run
                   ; Auto-reboot, keep turning
Bank Poison        ; Whichever robot executes this bank
Die                ; ...commits suicide!
```

Figure 3: An example Robocom program with 2 banks, see [7]

Execution starts with the first bank and auto-reboots to this bank if there are no more statements to execute. Figure 3 shows an example of a RoboCom program with 2 banks.

During this project we developed a game to do our research where small programs can compete with each other and evolve to perform better. The programs use a small instruction set inspired by games like CoreWar and RoboCom. The way in which simultaneous actions are handled is loosely inspired by games like Diplomacy (see [8]). Care has been taken to ensure a deterministic behaviour. The next section describes the game we developed in large detail.

3 Methods

As stated in the introduction of this report, we will be using a gamelike environment to do our research. In the sections that follow, we will briefly discuss the different elements in our environment.

3.1 The gameboard

The game will be played on an $n \times n$ gameboard (with $n \geq 2$) and a minimum of 2 and a maximum of 4 players. This is due to the fact that we use a square gameboard and each player resides at one side of the board aiming to reach the opposite site. The gameboard is a tiled board with *fields*. A field can be

- **a wall** if it is an obstacle that is unreachable for players
- **empty** if it is not a wall and no player has put a pawn on it
- **occupied** if a player has put a pawn on it

Walls are colored black and empty fields are plain white fields on the gameboard. All goals of a player, i.e., the empty fields at the opposite site of the board, are colored according to the players' color to make it easy to identify the goals. If a player moves one of its pawns to a field, then that field is not empty anymore and will be colored according to the color of the player. Figure 1 shows an example of a 20×20 gameboard with 4 players and 10 pawns for each player. Player 1 is colored pink, player 2 is green, player 3 is orange and player 4 is cyan. This example with given players, their starting positions and colors will be used throughout this report. For example, if we use an orange color for a pawn in an example then this means player 3 of Figure 4, unless explicitely stated otherwise.

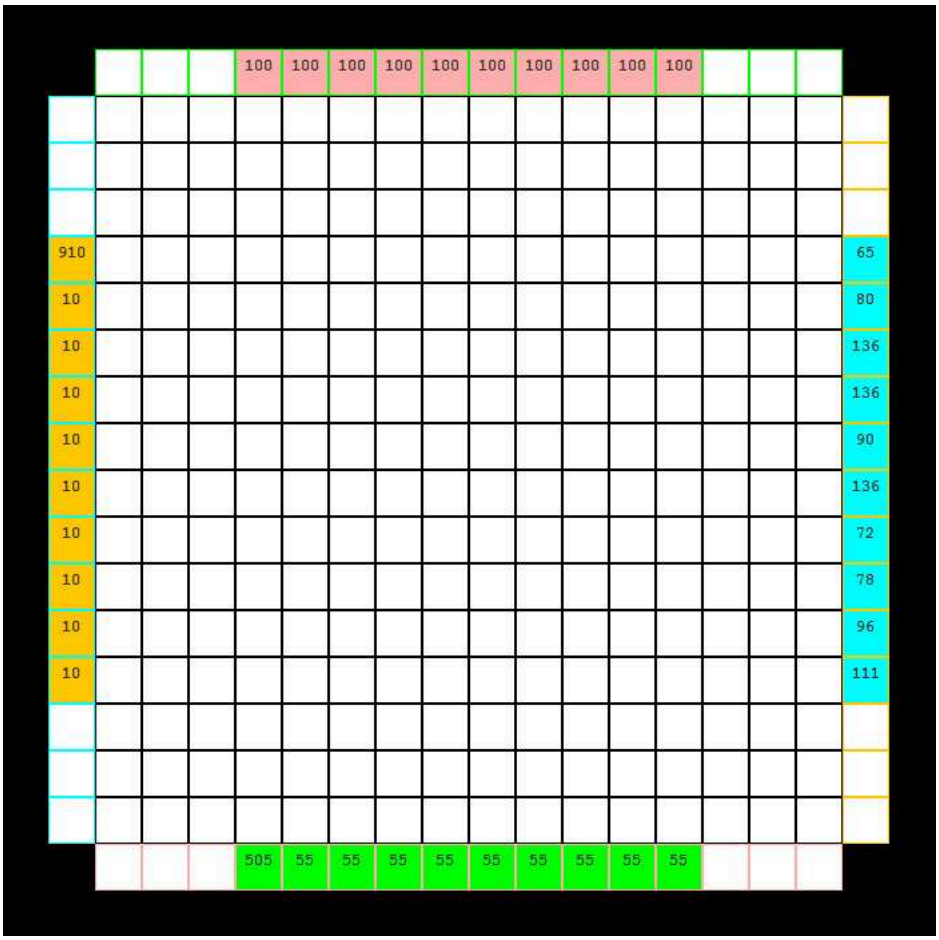

Figure 4: A 20×20 gameboard; initial position

3.2 Players

The "intelligent agents" in our project are the players of the game. The number of players may differ between 2 and 4, and is determined while configuring the game settings. All players have different colors to identify them. The players have a predetermined position at start. Player 1 always starts at the north side of the board and player 2 at the south side. If there is a third player then it will start at the west side and an optional fourth player will start at the east side.

Players have pawns that are placed on the board. The number of pawns p for each player is the same and, like the number of players, it is determined while configuring the game settings before the game starts. We have $p \leq n$.

Players possess power. At the start all players receive an equal amount of power which they spread amongst their pawns. This spreading of the total power is different for each player. Each player has its own strategy for distributing the power to its pawns. In the example from Figure 4 we have the following. Player 1 divides its power equally amongst the pawns. Player 2 gives half of its power to one of the pawns and divides the remaining part equally amongst the rest of the pawns. Player 3 gives one of its pawns a relatively large amount of power and distributes the remaining part equally. Player 4 spreads its power randomly. Table 1 shows the distribution of the powers for the example in Figure 4. The numbers in the first row indicate the pawns of the players.

Player	1	2	3	4	5	6	7	8	9	10	Total
1	100	100	100	100	100	100	100	100	100	100	1000
2	505	55	55	55	55	55	55	55	55	55	1000
3	910	10	10	10	10	10	10	10	10	10	1000
4	65	80	136	136	90	136	72	78	96	111	1000

Table 1: Distribution of powers for Figure 4

3.3 Pawns

We already mentioned that all players start with a predetermined number of pawns. Each pawn has its own individual power. When a pawn is put on a specific field of the board, the color of that field will be changed to match the color of the player that owns the pawn, and the power of the pawn will be shown on that field. For example, the numbers on the fields in the example of Figure 4 correspond to the power of each pawn. At the start, the sum of powers for all pawns of a player is equal to the total power of that player.

A pawn finishes if it reaches its goal, i.e., the opposite side of their starting point. All empty fields at the opposite side are goals for the pawn. It does

not matter on which of these empty fields a pawn finishes. Note that these goals can also be occupied by pawns of other players. A finished pawn will not participate anymore in the game during the current round.

3.4 Programs

All pawns have small programs which determine how a pawn should act on its environment. A program is a sequence of actions that should be taken by the pawn during each step of the game. Programs are plain text files with a simple format to describe the actions. The format of the progams is inspired by the programs used in RoboCom (see [7]). Figure 5 shows part of an example program. The structure of the programs must be exactly like shown. Each line denotes a single action or move that must be taken during one step of the game and starts with a label number followed by a semicolon. Label numbering should start from zero. The rest of the line should contain the action itself which can consist of several parts. Each part of an action or move should be separated by a blank space. If the end of a program is reached, the program will auto-reboot itself to start over again from label 0. The different actions that can be used will be discussed in the next section.

```
0: MOV N
1: MOV N
2: SCN W
3: IF W ENEM GOTO 4 ELSE GOTO 5
4: ATT W
5: MOV N
6: MOV N
7: ...
```

Figure 5: An example program

Programs are controlled by the players. When launching the game, a random pool of programs is generated. Players randomly choose a program for each of its pawns from this pool of available programs. Each pawn can have a different program but it is also possible that a single program is used by several pawns.

After each round, a player can adjust the programs of some of its pawns to perform better during the next round of the game. This stage of the game is called the *transfer stage*. A pawn can change its program by cloning it or by mutating it.

3.4.1 Cloning

The first action that will be performed during the transfer stage is cloning. Each player, starting with the worst, will change the program of its worst pawn by cloning *the program of the overall best performing pawn* or *the best program* that is not cloned yet. We will explain this in a minute.

Recall that different pawns can have the same program when launching the game. This means that the best performing pawns might use the same program. However, these programs do not need to be identical. When we say that pawns might use the same program, we mean that when launching the game, these pawns used the same program file as their initial program. But the programs might change after consecutive rounds because players can mutate the programs of their pawns.

There are two possible types of cloning strategies which can be chosen before starting the game:

- *Cloning Pawns*: When cloning pawns we make sure that each pawn can be cloned only once during a round. When using this strategy, we don't look at the programs but instead we look at the indivual pawns. This means that it would be possible that the same program used by different pawns could be cloned more than once during a round. With this strategy, one program could eventually dominate the whole because several pawns could clone the same best performing program. Of course, although the programs were the same during the first round, they might have changed at a later stage due to mutation.
- *Cloning Programs*: When cloning programs, we make sure that each program can be cloned only once during a round. When using this strategy, it is not possible to clone a program more than once, even if it belongs to

another pawn that changed it due to mutation. As a side-effect, we need to make sure that there are enough programs that can be cloned. While the game progresses, certain programs will be used more often because they perform well. The total number of programs used during a round decreases. We make sure that there are at least n different programs, with n the number of players, that can be cloned during every round. If this was not the case, some players would not be able to clone anything because of the restriction that a program can only be cloned once during a round.

3.4.2 Mutating

After the cloning process, each player, again starting with the worst, will change the program of its second worst pawn by mutating it. A random command from the program will be selected and used for mutation. There are four types of mutations possible:

- *Mutate by Editing*: Selected command will be edited by replacing it with a new randomly generated command.
- *Mutate by Adding*: A new randomly generated command will be added after the selected command.
- *Mutate by Deleting*: Selected command will be deleted. To avoid getting empty programs, a command will only be deleted if there will be at least one command left after deleting it.
- *Random mutation*: Selected command will be mutated by one of the above methods.

The type of mutation that will be used is set before starting the game and doesn't change until the end of the game. Note that a player has to have at least one pawn left for cloning and at least two pawns for mutation. This is because the worst pawn will be used for cloning and the second worst pawn will be used for mutation. If there is just one pawn, only cloning will be applied.

3.5 Moves

Programs can contain several moves or actions which tell the pawns what to do in each consecutive step of the game. There are several commands that can be used when defining actions. Following is a list of possible commands and their actions:

- **MOV** move to a field in a given direction
- **ATT** attack the field in a given direction
- **IF TEST GOTO X ELSE GOTO Y** a boolean test
- **JMP** jump to a label

All pawns and their programs are considered at the same moment in time. This means that simultaneous actions are possible, e.g., multiple attacks to the same pawn. To make sure that this simultaneity is performed correctly we adhere to a standard sequence for actions as listed below:

1. All boolean tests are evaluated and executed.
2. All attacks are computed.
3. All attacks are effectuated. Dead pawns (with power ≤ 0) are removed.
4. All other moves are executed.

Boolean tests are evaluated and executed before any other action in a program. This is to make sure that the first boolean test evaluation does not cost a move. The boolean test will be evaluated and executed. If a boolean test jumps to another boolean test, the test that is jumped to will not be executed immediately and has to wait for the next move. After the initial boolean tests are performed, all attacks are computed and effectuated. We first compute all attacks before effectuating them to ensure the simultaneity of the attacks. All other moves are executed after these steps.

All these possible commands will be explained in the following sections.

3.5.1 Move

A *move* is just a basic movement of a pawn in a certain direction. The program command used for this action is "MOV X", where X denotes the direction to move in. Directions are given by the four cardinal directions north, east, south and west. We use the first letter of these directions in commands, i.e., N, E, S and W. All directions are relative to the starting position of the players. A step forward literally means a step forward. Let's take a look at an example. If player 3 in Figure 4, i.e., the orange player, reads a "MOV N" command it will move its pawn one field to the right on the board. Figure 6 shows a simple move command to an empty field. This is not a move to the east! It is a move to the north for player 3.

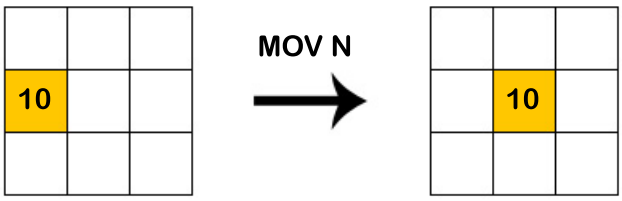

Figure 6: A move

A move is only possible if the target field is empty. If the target field is a wall or (still) contains a pawn of another player, then the move is not possible and the pawn stays at its current position. When several pawns want to move to the same empty field at the same time, none of the pawns will move and all stay in their current position.

3.5.2 Attack

An *attack* move is a move where a pawn diminishes a certain amount of power from another pawn. But, instead of taking, it can also give power to another pawn. In both situations its own power will decrease with the same

amount of power that is taken or given. The difference is between enemy and friendly pawns. An enemy pawn is a pawn that belongs to another player. A friendly pawn is a pawn that belongs to the same player. If a pawn attacks an enemy pawn, it will diminish the power of the enemy pawn. If a pawn attacks a friendly pawn, it will give power to the friendly pawn. This can be quite confusing to understand at first because of the term we use for this action. A friendly attack could be described as a kind of transfer of power from one pawn to another whereas an enemy attack is based on bad intentions trying to diminish power of the enemy pawn. Figure 7 shows an enemy attack and Figure 8 shows a friendly attack. Note that in enemy attacks power gets lost, while in friendly attacks the total amount of power remains the same.

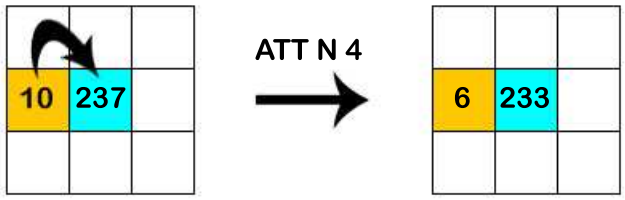

Figure 7: Attacking an enemy

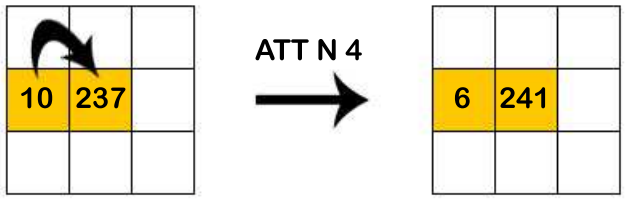

Figure 8: Attacking a friend

The program command used for attacks is “ATT X P”, where X denotes the direction to attack and P is the amount of power that will be used for the

attack. For example, in Figures 7 and 8 we used ATT N 4 to attack the field in the north direction with a power of 4. We can however omit the value for the power, using a command of the form "ATT X". In this case we do not explicitely specify the power to be used for the attack so we will use all of the attacking pawns' power. This means that after such an attack, the attacking pawn has a power of zero, so it will cease to exist and will be eliminated! Figure 9 illustrates this situation which we call a "full attack".

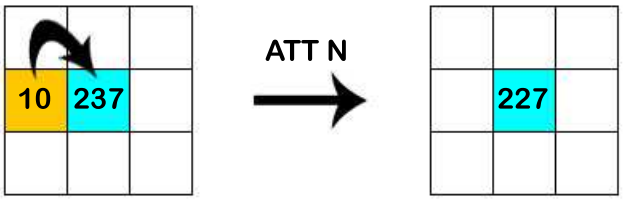

Figure 9: A full attack

A pawn could be attacked by several friendly or hostile pawns simultaneously. While being attacked by these pawns, the pawn itself can attack another pawn too. All incoming attacks and the outgoing attack are calculated to determine the resulting power of the pawns. Figure 10 shows an example of this situation.

Figure 10: Simultaneous attacks

If the power of a pawn is ≤ 0, it is removed from the board. Note that we

consider the total amount of power for this pawn after all attacks have been considered, see e.g., the pawn with power 8 in Figure 10.

3.5.3 Boolean test

A *boolean test* can be used within programs to determine which command should be executed depending on a boolean condition. The program command for a boolean test is "IF TEST GOTO L1 ELSE GOTO L2", where TEST is the boolean condition to test, L1 is the label to jump to if the condition is true and L2 is the label to jump to if the condition is false. The TEST part always contains the direction as the target for the test evaluation, i.e., N(North), E(East), S(South) or W(West) and can additionally use one of the following elements:

- **FRND** a friendly pawn
- **ENEM** an enemy pawn
- **WALL** a wall
- **EMPT** an empty field
- **GOAL** a goal

An example of a boolean test is given in Figure 5 on label 3. It states that "IF W ENEM GOTO 4 ELSE GOTO 5", where we first check if there is an enemy at the west side of this pawn. If true, we jump to label 4 and immediately execute the command on label 4. Otherwise, we jump to label 5 and immediately execute that command. If the command on label 4 or 5 is another boolean test, then this new test is not immediately executed and the pawn waits and executes this boolean test command during the next move. If one of the label numbers does not exist, the program reboots by starting again at the first label.

The FRND and ENEM conditions can have optional elements used for relational tests. These are:

- **GEQ** relational operator $\geq$ (greater than or equal)
- **LEQ** relational operator $\leq$ (less than or equal)

A boolean test of this extended form could be like "IF W ENEM GEQ 17 GOTO 4 ELSE GOTO 5". This example is the same as we saw before with one major difference: we added a relational operator to test the power of the enemy pawn. If the target is an enemy and its power is greater than or equal to 17, the boolean test will evaluate to true and control goes to label 4. Otherwise we continue at label 5.

3.5.4 Jump

A *jump* is basically a movement of control in the sequence of actions listed in a program file. Each line of action starts with a label. A jump command tells the program to jump to a specific label. The program command for a jump is "JMP L", where L stands for the label number. If a given label number does not exist, the program reboots by starting at the first label.

3.6 Scores

To measure the performance of a player or pawn we give them scores. The score of a player is the sum of the scores of their pawns. After each round the scores are evaluated for each player to determine the best players. These scores are written to a file before they are reset for the next round.

There are three ways of calculating scores for players. Before starting the game we determine how the scores will be calculated by selecting one of these three possible methods. Each of them will be discussed in the next sections.

3.6.1 Distance Only

The *distance only* method is the simplest method for calculating scores. When this method is used, we only look at the absolute distance a pawn has travelled from its starting position. Each step towards its goal, i.e., a move to the North, will increase the score by adding 1 to it. Each step back, i.e., a move to the South, will decrease the score by 1. Moves to the East or West will not effect the score at all. As discussed in Section 3.5.1 these directions are all relative to the starting point of a pawn. In the following sections we will use the term *distance* to indicate the distance that a pawn has travelled towards its goal, i.e., the total number of moves to the North. The score of a player according to this method is calculated by:

$$Score = \sum_{i=1}^{n} D_i$$

with n the total number of pawns for a player and D_i the distance travelled by pawn i.

We illustrate this by taking a look at Figure 11. It shows a small 8×8 board with one pawn on it. The starting position of this pawn is indicated by the orange dot at the left. Goals for the pawn are as usual at the opposite side of the starting point and are shown with an orange border. The movement path of the pawn during each sequential move is indicated by the black arrows. As we can see, this pawn has made 4 moves. But the distance from the starting point to its current position is merely 2 as indicated by the red arrows. The pawn made 4 moves but scored merely 2. If this pawn would have moved in a straight line towards the goals, it would only need 2 moves to reach the same distance from its starting point.

The maximum score a pawn can get with this method is the shortest distance from the starting point to the goals which is $n - 1$ for an $n \times n$ board. In the example of Figure 11 this would be 7. However, as we will see in Section 3.6.4, a pawn can get a much higher score by reaching its goal.

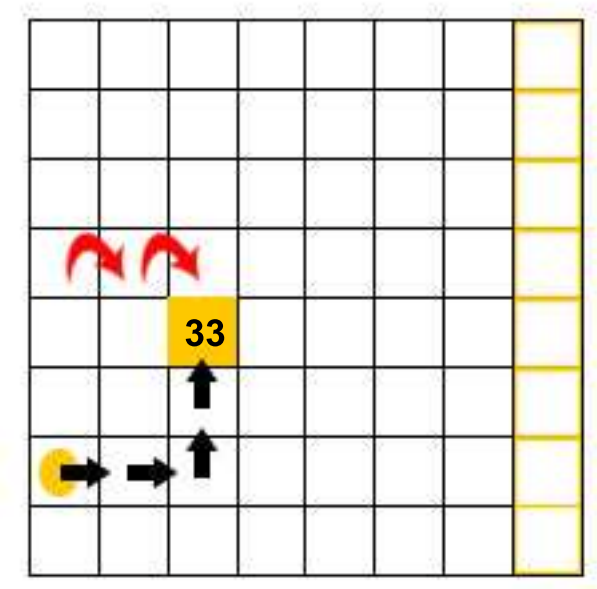

Figure 11: Pawn made 4 moves but has a distance of 2

3.6.2 Distance and Power

Assume that two or more pawns have travelled the same distance from the starting position. If these pawns have different powers, would it be fair to give them all the same score? The *distance and power* method uses the same score calculation as the distance only method but in addition multiplies that score by the power of a pawn. The score of a player is calculated by:

$$Score = \sum_{i=1}^{n} D_i \times P_i$$

with n the total number of pawns for a player, D_i the distance travelled by pawn i and P_i the power of pawn i.

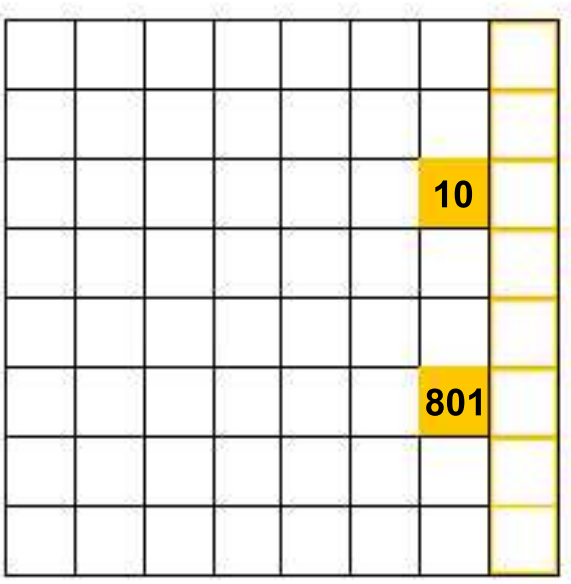

Figure 12: Equal distances, different powers

A pawn with more power will have a higher score if the distance from the start is the same. We will use Figure 12 to illustrate this. Take a look at the two pawns with powers of 10 and 801 which are almost finished. With the distance only method, both pawns would have the same score of 6. But with the distance and power method we take the power of the pawns into account. The pawn with power 10 will get a score of 60 (6×10) and the pawn with power of 801 will get a score of 4806 (6×801).

3.6.3 Distance and Power reversed

As you can probably guess by the name of this method, it is similar to the distance and power method with one difference: the lower the power of a pawn, the higher the score and vice versa. That is why we added the keyword *reversed*. For example, in Figure 12 the pawn with a power of 10

would receive a higher score than the pawn with a power of 801. The idea behind this method is that a pawn with less power has to struggle harder to reach that far and should be rewarded better. A pawn that has much power would have to take less effort in reaching the same distance.

The score of a player for this method is calculated by:

$$Score = \sum_{i=1}^{n} D_i \times (P_{start} - P_i)$$

with n the total number of pawns for a player, D_i the distance travelled by pawn i, P_{start} the initial total power of a player at the start of the game and P_i the power of pawn i.

3.6.4 Bonus scores

A pawn that finishes receives a bonus score as a reward. This reward is predefined in the game configuration before the game starts and is equal for all pawns. When using the distance only method the bonus is 1000 points. When using the other two score calculation methods, pawns can get very high scores if there power is high so the bonus for these methods is raised to 5000 points. Recall that finished pawns do not participate in the game anymore. This means that the score of a pawn can not change after it has finished.

4 Results and Experiments

This section analyses the results of the experiments we did. Before we show the different results we explain the game settings that are used while experimenting. Some of these settings are used as a standard for most of our experiments. If not explicitely mentioned otherwise, the basic settings for experiments are as follows:

Standard Settings	
Gameboard size	20×20
Number of players	4
Number of pawns	10 for each player
Number of rounds	100
Number of moves	100 for each round
Initial power per round	1000 for each player

There are three other settings that can be configured before starting a game. Two of them are discussed in Section 3.4 as the *cloning style* and *mutation style* options. The last setting is the *score calculation* method and was discussed in Section 3.6. These three settings will be our *variable settings* while doing the experiments and will be mentioned separately for each experiment.

4.1 Game domination by cloning pawns

We mentioned earlier in Section 3.4.1 that if we would clone pawns, a situation might occur where one program could dominate the whole game. This is due to the fact that one program could be used by several best performing pawns. By cloning pawns instead of programs we could end up in a situation where one program would be cloned several times and if this repeated itself during consecutive rounds, every pawn would end up using this sole program. We could say that all pawns learn to use this best performing program. This is related to the notion of *premature convergence* in genetic algorithms where a population for an optimization problem converges too early.

During this experiment we tried to find out if we could indeed replicate this situation. The variable settings used for this experiment are as follows:

Variable Settings	
Cloning style	Cloning pawns
Mutation style	None
Score calculation	Distance only

Note that we turned off mutations for this first experiment because mutations could cause uncertainties to happen and for this first experiment we just want to know if we can confirm our assumptions. We will however try the same experiment with mutations turned on later.

After six trials we were indeed able to replicate the desired state. Just after only nine rounds we reached a state where one program was used by all pawns. Figure 13 shows this program which turned out to be a program with a single command.

```
0: MOV N
```

Figure 13: The dominating program

Figure 14 shows the scores of all players during consecutive rounds. After round 9 all players reached the same score of exactly 50 over and over again. The positions of the pawns on the gameboard formed an X-like shape where no pawn could move anymore. This final state is shown in Figure 15.

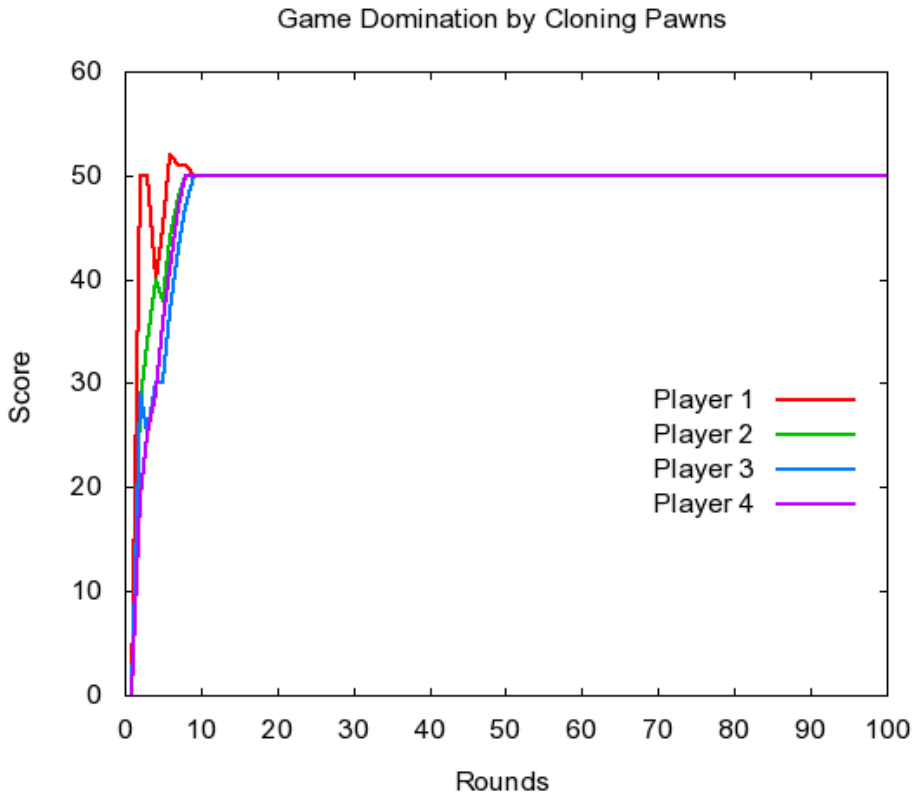

Figure 14: Game domination by single program

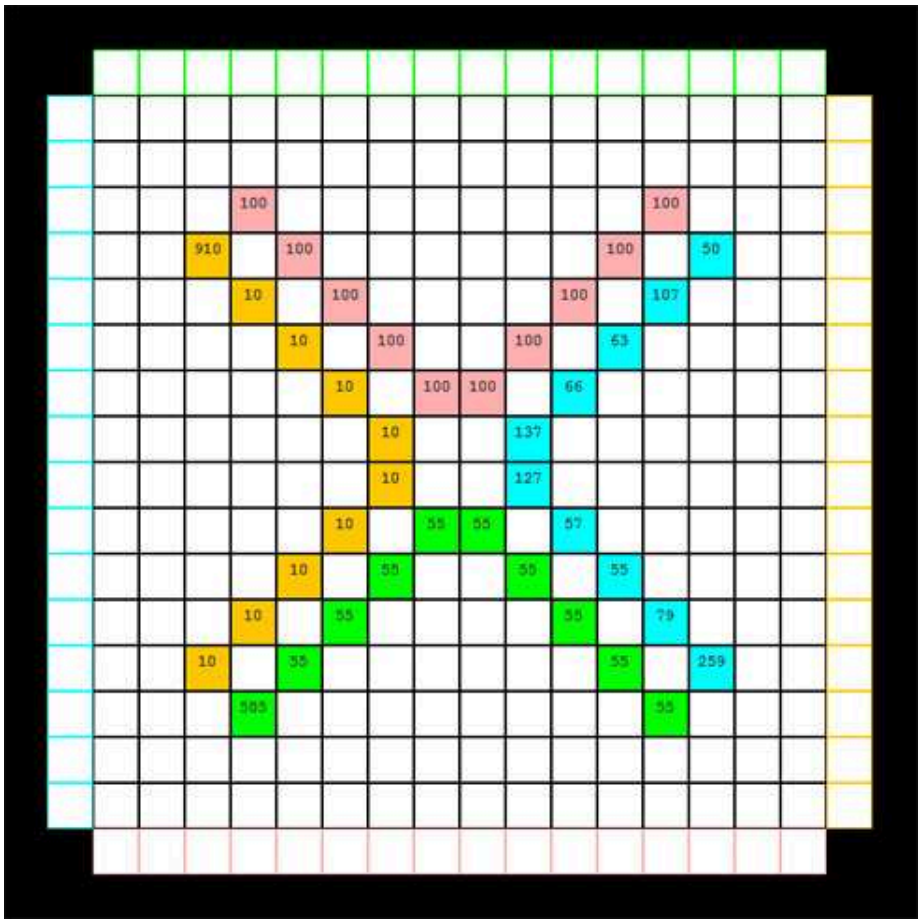

Figure 15: All pawns using same program

As noted before, we did the same experiment with mutations enabled. The settings are equal to the previous experiment but during this experiment we enabled random mutations which are applied to some of the programs after every round. Figure 16 shows the changes in scores during this experiment.

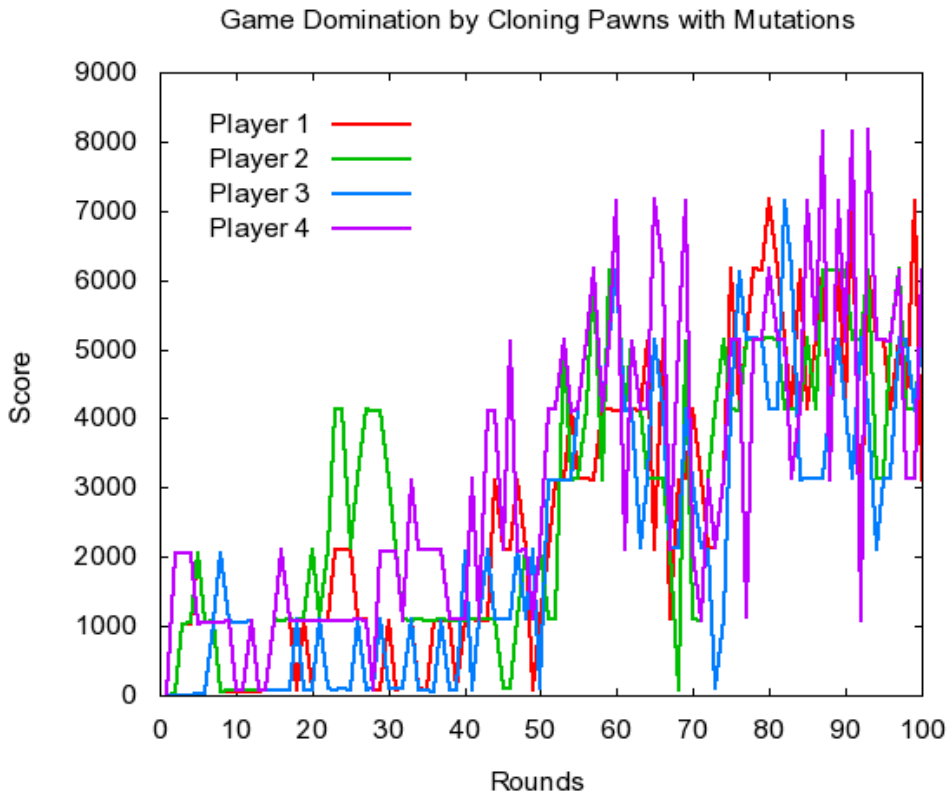

Figure 16: Game domination with mutations enabled

There are moments when the scores are rising up or falling down very quickly. Take a look at players 2 and 3 between rounds 65 and 75 for example. The scores of both players dive towards zero before they rise again. This is due to the fact that their well performing, finishing pawns, are not reaching their goals anymore. The programs of these pawns are probably mutated or they have been attacked by other pawns and died.

Mutations can change a pawns behaviour with just the slightest change in a program. As we can see, the scores of the players are rising as the game is played but there is no convergence in the scores. This means that the pawns of the players are indeed learning to play better but they use different programs due to the many mutations they suffered from.

4.2 The effect of mutation by deleting

In Section 3.4.2 we mentioned a method to mutate programs by deleting a random command from the program. A constraint for this mutation style was that after deletion at least one command should be left within the program. Otherwise we would end up with an empty program for a pawn. But what if we keep on deleting one command at a time for all programs that are used by pawns. Would we reach a state where all programs consisted of just a single command? How would such a state affect the performance of the players? Each program with just one command would result in a round where pawns using this program would make the same move over and over again until the end of the round. During this experiment we tried to answer these questions and examined the effect of mutating programs by deleting commands.

The variable settings used for this experiment are as follows:

Variable Settings	
Cloning style	Cloning programs
Mutation style	Mutate by deleting
Score calculation	Distance and power

Figure 17 shows the results of this experiment. As we can see, all players reach a certain score and their performance stagnates till the end of the game. If we look at the figure, we see that the performances of players 1, 2 and 4 stagnate aproximately after round 45 of the game. For player 3 the scores do not change anymore after round 36. This is due to the fact that pawns are making the same moves over and over again just like we predicted. Our assumption was right for this particular experiment.

We tried another experiment with the same settings. The stagnation of scores during this experiment was a little bit different than we assumed. It was not because *all* pawns use single-command programs. In fact, quite a few pawns still use programs that contain a lot more commands.

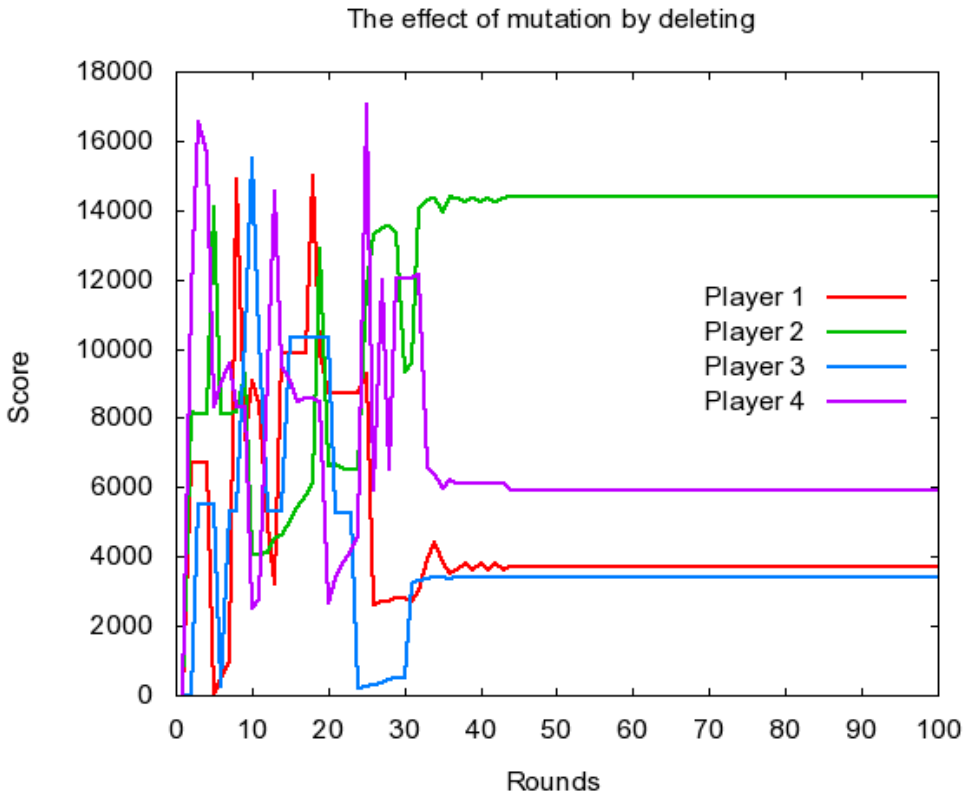

Figure 17: Stagnating performance with mutation by deleting

The number of commands in programs used by each pawn after the game has finished is shown in Figure 18. It shows that indeed a lot of programs use just one command instruction, probably due to mutation by deleting. But it also shows us that there are still pawns that are using programs with more than one command, in paticular, there are 2 pawns that have programs with 8 commands.

Pawns that perform good will not be mutated. If a pawn is performing bad,

at some point its program will probably be subdued to cloning and mutation. Recall that mutation by deleting was only possible if after deleting, a program still had at least one command left. This means, that none of the single-command programs will be mutated. At some point, there will be no mutations at all anymore and the player performances will not improve nor reduce. It stays the same because the pawns do exactly what they did the previous rounds.

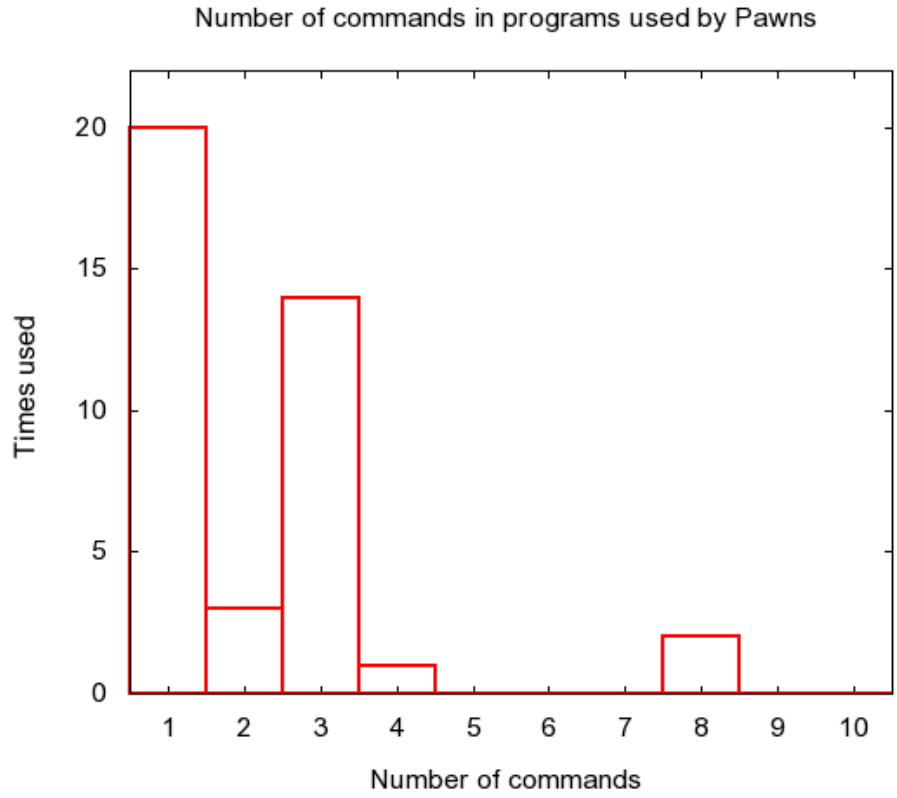

Figure 18: Not just single-command programs after mutation by deleting

4.3 Comparing different muation styles

During this experiment we only viewed the performance of one player under different styles of mutations. We tried to find out if there is a substantial difference in performance when a player uses different mutation styles. We used cloning by programs as cloning style and the score calculation is set to the distance and power strategy. We evaluated the performance of only player 1.

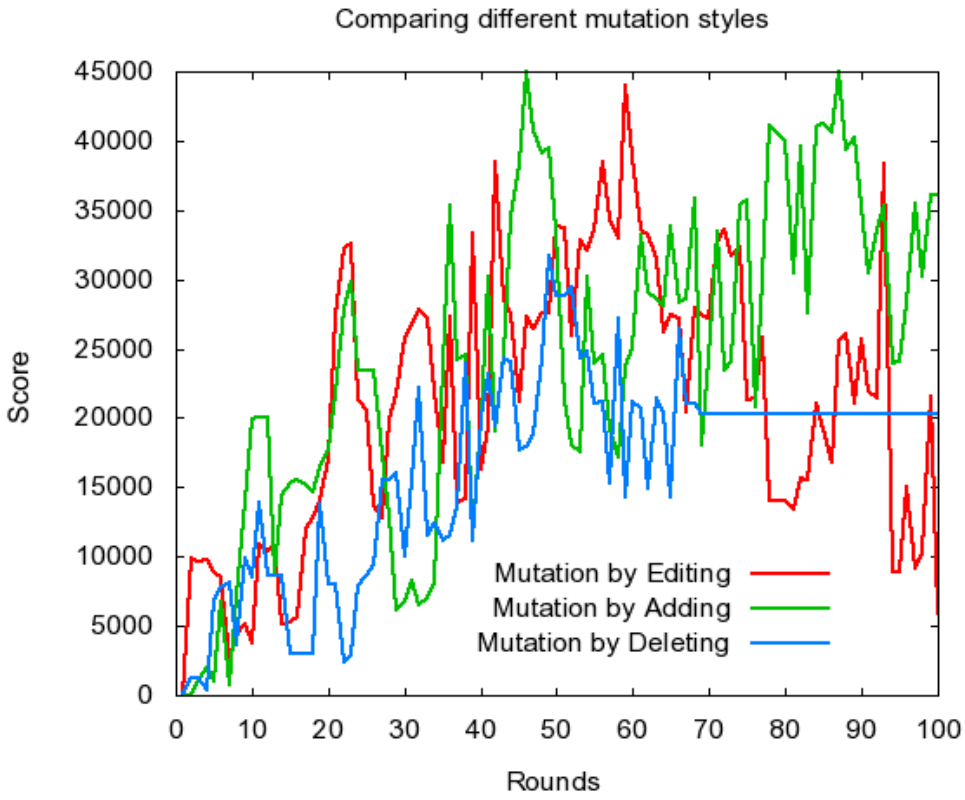

Figure 19: Comparing different mutation styles for player 1

Figure 19 shows the results of this experiment. It is obvious that there is quite a difference in performance and learning for player 1 when using the different mutation styles. Player 1 performs best when using mutation by

adding. The performance curve is gradually ascending. When mutation by editing is used, the performance is quite unpredictable and according to our findings this is the worst mutation strategy to use. It starts out very well by performing better and better until around round 60 it starts dropping down. This is probably due to the fact that changing a random command of a program can dramatically change the behaviour of the pawn using that program and results in unpredictable behaviour. The performance curve when using mutations by deleting is exactly as expected and stagnates to a certain value. We already saw this happening during our previous experiment which is discussed in Section 4.2.

4.4 Finding good performing programs

Players clone and mutate programs of their pawns to learn and to perform better. The aim is to create programs that perform better. But how does a good performing program look like? During this experiment we tried to compare the best performing programs that were generated after several test runs of the game.

The variable settings used for this experiment are as follows:

Variable Settings	
Cloning style	Cloning programs
Mutation style	Random mutations
Score calculation	Distance and power

Table 2 shows the final scores of the players after several test runs. The table shows some interesting results. For example, run 3 shows that all players performed very well which resulted in very high scores. This is a good example of what we have been trying to achieve or at least what we wanted to see in an ideal situation. That is that all players are learning from each other by cloning and are indeed becoming more and more "intelligent". But this is not true for all the runs we did. Take a look at run 5 for example. It shows that player 3 scored very bad while the other players performed well.

The mutation factor probably influenced the behaviour of the pawns for this player in a negative way.

	Player 1	Player 2	Player 3	Player 4
Run 1	45500	16340	30720	32979
Run 2	34400	40955	26600	29564
Run 3	51632	51330	50948	55424
Run 4	25232	16210	36530	27378
Run 5	21700	29195	8610	39698

Table 2: Final scores of players after several runs

Now let us take a look at the best performing programs during these runs. Figure 20 shows these best performing programs for all five runs. We will try to explain why these programs are good by evaluating them step by step.

The first program is making a diagonal move towards the goal by moving north and east repeatedly. Every second move is a move forward and thus a really good move. Pawns using this program will travel far and might even reach their goals.

The second program starts with an if statement. The if statement will go to statement 4 if there is no friendly pawn ahead. Statement 4 tells the pawn to move forward. The next command is another if statement and will, in most cases, default to the first command because label 8 does not exist. This sequence is repeated and results in a good strategy because the pawn is moving north.

The third program tells the pawn to move east, north and west repeatedly. Of every three moves, a pawn will move one step towards its goal.

The fourth program is in essence just a single move command because if we look at the first if statement, it refers to labels 8 and 9 which do not exist. Thus every time this if statement is called, it will default to the first command of the program which is a move to the north. Control goes from the first command to the second and returns to the first again. This means that of every 2 moves the pawm makes, one is always a step towards the goal.

The fifth program has 7 command statements but the last two are unreachable because of the first if statement at label 4. A pawn makes two moves west, two moves north and repeats these steps depending on the evaluation of the boolean condition of label 4. Like all other good performing programs, this is also making a pawn move towards its goal as much as possible.

```
Run 1
0: MOV N
1: MOV W

Run 2
0: IF N FRND GOTO 7 ELSE GOTO 4
1: IF S FRND GEQ 82 GOTO 9 ELSE GOTO 1
2: MOV E
3: MOV W
4: MOV N
5: IF W ENEM GEQ 59 GOTO 4 ELSE GOTO 8
6: JMP 5
7: MOV E

Run 3
0: MOV W
1: MOV N
2: MOV E

Run 4
0: MOV N
1: IF S FRND GOTO 9 ELSE GOTO 8
2: IF E ENEM GOTO 7 ELSE GOTO 8
3: IF N FRND GOTO 4 ELSE GOTO 5

Run 5
0: MOV W
1: MOV W
2: MOV N
3: MOV N
4: IF S ENEM GOTO 1 ELSE GOTO 2
5: IF E FRND GEQ 15 GOTO 2 ELSE GOTO 8
6: IF E WALL GOTO 6 ELSE GOTO 2
```

Figure 20: Best performing programs during several runs

5 Conclusions

In this research we developed a game system, where the individual players are represented by teams of autonomous agents, the so-called pawns. Every pawn acts according to its own (simple) program or strategy. Several mechanisms were introduced that guide the evolution of the strategies involved.

During this research we tried to find out if we would find an increase in performance for some of the pawns based on the things they learned. Our findings in Section 4 show clearly that we did indeed find an increase in performance during our experiments. Agents "learned" to play better by cloning and mutating their programs.

Cloning programs turned out to be a good learning scheme for pawns. All our experiments showed that there was a rising curve when viewing the scores of players. But there are some minor drawbacks to this strategy as we saw in Section 4.1 where all pawns started to use the same program. It showed that all pawns learned to play at their best and there was no more learning possible. Of course, this is not what we would want because our main goal was to find increasing intelligence. This is why we added some kind of evolutionary factor which we naturally called mutations.

Mutations of programs create unpredictable behaviour. Some mutations can dramatically change a pawn's behaviour with just a small change in the program of the pawn. This could turn a good program into a bad one and vice versa with just one small change in the commands. This is exactly what we observed during our experiments. When viewing the different performance plots in Section 4 it is evident that mutations are present because the performance rises and drops very quickly in some cases. Our experiment in Section 4.3 showed that it is also important to determine how we will apply mutations and what strategy we will use. An interesting twist for future research could be to constrain mutations to be allowed only if they increase the performance of a pawn.

As we saw during our experiments in Section 4.4, a good program is a program that makes a pawn move as far as possible towards its goal. This is quite obvious because the score calculation depends on the distance travelled by

the pawn, i.e., the number of moves to the North. But is this a good measure to calculate the performance? Imagine a pawn that has attacked and killed several pawns but did not travel that much. Would this pawn be a good pawn? Should it rank high in our top scores list? Let us assume that is has just made one move towards its goal. In our experiment this pawn would not be qualified as a good pawn, however, by eliminating several other enemy pawns it probably did make life easier for its fellow friendly pawns. The way we determine the performance of an agent or pawn could have a huge impact on the results we observe. It might be interesting to do more research with these kind of questions in mind.

References

[1] Poole, D.; Mackworth, A.; Goebel, R. (1998). Computational Intelligence: A Logical Approach. New York: Oxford University Press.

[2] Russell, S.; Norvig, P. (2010). Artificial Intelligence: A Modern Approach. Third edition. Englewood Cliffs, NJ: Prentice-Hall.

[3] Dewdney, A.K. (May 1984). In the game called Core War hostile programs engage in a battle of bits. Scientific American, 1984, v.250, N 5, pp. 15–19.

[4] Dewdney, A.K. (March 1985). A Core War bestiary of viruses, worms and other threats to computer memories. Scientific American, 1985, v.252, N 3, pp. 14–19.

[5] Dewdney, A.K. (January 1987). A program called MICE nibbles its way to victory at the first Core War tournament. Scientific American, 1987, v.256, N 1, pp. 8–11.

[6] Core War website, `http://www.corewars.org/` [retrieved Feb 8, 2010].

[7] RoboCom website, `http://robocom.rrobek.de/` [retrieved Feb 8, 2010].

[8] Diplomacy website, `http://www.diplomacy-archive.com` [retrieved Feb 10, 2010].