Solving Nim by the Use of Machine Learning

Exploring how well Nim can be played by a computer

Mikael Nielsen Røykenes

Thesis submitted for the degree of
Master in Informatics: Programming and Networks
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Autumn 2019

Solving Nim by the Use of Machine Learning

Exploring how well Nim can be played by a computer

Mikael Nielsen Røykenes

Solving Nim by the Use of Machine Learning

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Contents

1 Intro

The game of Nim is rather old, how old is uncertain, but certainly over a hundred years old. It is also a pretty simple game, and it has a complete mathematical theorem, see "Nim, a game with a complete mathematical theorem".[1] There is a fairly simple algorithm for making winning moves, so simple that a human can learn it by heart and use it. The author of this paper being an example of this. Machine learning is a type of stochastic algorithms that try to find a solution based upon statistical data. A stochastic algorithm is an algorithm with some random elements, a deterministic one does not have any random elements. It can often find good solutions quite quickly, certainly faster than conventional deterministic algorithms. Machine learning as well as other stochastic types of algorithms can often be good at finding good enough solutions, that is solutions which arent perfect, yet still good. In this paper the problem which will be explored is how proficient it will be at playing perfectly, that is, having a chance at winning against a human player that knows how to play perfectly.

1.1 My Goal with This Thesis

What I am trying to do is to have a computer play Nim perfectly, with as little fore-knowledge as possible. This is reminiscent of computers playing other games, like chess. For example the chess-match between the Grandmaster Garry Kasparov playing against the computer Deep Blue. While having problems at first, it would eventually win a game, and then a full match against Kasparov. I am going to use stochastic methods to achieve this. This is different from Deep Blue, which used deterministic algorithms to choose moves.

1.2 Hypothesis

Making a program that uses reinforcement learning to play Nim should be able to do quite well, given enough time. I expect this because Nim, like other games, can be expressed in the form of a tree or a graph, and reinforcement learning is usually good at navigating graphs. On the other hand I believe it will be significantly slower than the pre-existing algorithm, at least in the worst case[2]. Furthermore, I do not believe that the algorithm will be able to train enough with a static amount of episodes.

2 Ethics

Clearly, solving nim with for example, reinforcement learning, does not in itself have much of an impact on anything, ethically. This is because there is already

[1] Charles L. Bouton. "Nim, A Game with a Complete Mathematical Theory". In: *Annals of Mathematics* 3.1/4 (1901), pp. 35–39. ISSN: 0003486X. URL: `http://www.jstor.org/stable/1967631`.

[2] Worst case time use does not make much sense in the case of stochastic algorithms, like machine learning. I believe the difference here will be so large it is significant anyway.

an algorithm even a human could easily learn how to use. It would not change anything about the world, as there isn't a competitive Nim scene or anything like that, unlike for example chess. If one was able to solve chess, which is possible given the finite amount of positions, that could have a serious impact on professional chess-players. Not so here. Because of that, a look into possible indirect consequenses of finding a solution here is warranted.

The long and short of it is that a solution to this problem might be a solution, or might inspire a solution for some unforseen other problem. The traits of the problem of solving Nim, requiring perfection from the training, as well as finding the correct move in all cases. Not just playing the game from the starting state perfectly. There might some other problem which share those traits.

3 The Game Nim

3.1 Rules and How to Play It

> "The game is played by 2 players, A and B. Upon a table are placed three piles of objects of any kind, let us say counters. The number in each pile is quite arbitrary, except that is well to agree that no two piles shall be equal at the beginning. A play is made as follows:–The player selects one of the piles, and from it takes as many counters as he chooses; one, two, ..., or the whole pile. The only essential things about a play are that counters shall be taken from a single pile, and that at least one shall be taken. The players play alternately , and the player who takes up the last counter of counters from the table wins."[3]

A game of Nim shown as an algorithm Assuming A is the first player.

1. Player A removes some amount of counters from a pile. If there are still counters left, go to 2., if not, go to 3.
2. Player B removes some amount of counters from a pile. If there are still counters left, go to 1., if not, go to 4.
3. Player A removed the last counter/s and is the winner.
4. Player B removed the last counter/s and is the winner.

When playing Nim there are several ways to represent it, like in the movie "Last Year at Marienbad" where the main character shows how to play the game with cards, putting them in a pyramid fashion, and having the piles be the rows of cards. Alternativly literal piles of counters can also be used. What all of these have in common is that they are unary in representation, that is, they have n objects for a pile of size n.

[3] Bouton, "Nim, A Game with a Complete Mathematical Theory", op. cit., p. 35.

3.2 Nimrod

During the "Festival of Britain" in 1951 in the UK, in the science exhibition, there was a computer called Nimrod with which visitors could play a game of Nim. This might very well be the first time that a Digital computer(as they called it) could play Nim. It is also probably[4] one of the first computer games to exist. It was a computer with 4 rows of lights indicating the size of 4 heaps. The computer also had a control panel which allowed human players to play with the computer. It had controls for different modes of play, as well as the same set of lights as on the computer, with companion buttons used to indicate the move the player wishes to make. An informational booklet was sold, which contained information about the computer and computers in general(at the time), the game Nim and how to play it, as well as an explanantion on how the computer itself plays Nim. It contains the very algorithm on how to play Nim that will be presented later in this paper. The booklet was sold for 1 shilling and sixpence.[5]

3.3 Impartial Games

An impartial game is defined as a game where:

- There are 2 players.
- The players take alternate turns to make moves.
- There are a finite amount of unique legal moves for a player in any given state of the game, and there are a finite amount of possible moves in a game.
- All moves must be possible for both players to make. That is, in a given state, no matter which players turn it is, the same moves are possible for them.
- When a player cannot make a move, one of the players win.
- There is complete information, that is, both players know everything about the current state of the game.
- There are no random elements, like the use of dice or shuffled cards.

Nim is an example of such a game, because it fulfills all those requirements. It has, for example, always a finite amount of legal moves, because for each pile there are a finite amount of counters to take. This means there are only so many moves that can be made. Since counters have to be removed for each move, the game cannot go on forever, as there are a finite amount of counters to take.

[4] It is hard to say excactly when and what the first computer game was, depending on several factors including the very definition of computer game. Considering it among the first seems likely however, given when it was available to the public. The question of what was the first computer game is a question worth a full article by itself.

[5] *The Ferranti NIMROD Digital Computer*. URL: `https://www.goodeveca.net/nimrod`.

The longest possible game one could have is the game where each player takes one counter every turn, there will then be as many turns as there are counters. When a player takes the last counter they win, and the next player cannot make any moves, so that criteria is fulfilled as well. A player's moves are decided by the state of the game, not by the player whose turn it is, and so the requirement for moves being possible for both players is fulfilled as well. It seems obvious that Nim fulfills the rest of the requirements.

Games That Are Not Impartial Chess, for example, has the players control different pieces on the board which break the fourth rule, since each player can only move their own pieces. In Ludo there are more than 2 players, each player has their own pieces, and a random element, a die, is used. All of which individually puts it out of the category of impartial games. In Poker there is not complete information, because the cards in each hand are hidden. Furthermore the shuffling and dealing of the deck is a type of random element, if done correctly[6].

3.4 The Sprague-Grundy Theorem

The Sprague-Grundy Theorem states that all impartial games are equivalent with a nimber. A nimber is defined as the value of a pile in a game of Nim.[7] The Nimber of a game of Nim is simply the Nim-sum of the game, which will be explained later. Because of this theorem, it seems easy to conclude that whatever solutions are found, they should be equally as good at playing other impartial games.

4 Machine Learning

The book "Machine learning: an algorithmic perspective" states that "Machine learning, then, is about making computers modify or adapt their actions (whether these actions are making predictions, or controlling a robot) so that these actions get more accurate, where accuracy is measured by how well the chosen actions reflect the correct ones".[8] In this paper only reinforcement learning and supervised learning will be used, and as such those are the only parts of machine learning that will be discussed here. Note that this will not be an exhaustive explanation, only on the parts of the subjects which are relevant, mostly those in use.[9]

[6] Shuffling cards in such a way that the cards end up in a predetermined way is possible, but this would be considered cheating. You'd get shot for less in Texas.

[7] John H. Conway, Elwyn R. Berlekamp, and Richard K. Guy. *Winning Ways for Your Mathematical Plays*. Second. Vol. 1. CRC Press, 2001, p. 56.

[8] Stephen Marsland. *Machine learning: an algorithmic perspective*. Second. CRC Press/-Taylor & Francis Group, 2015, p. 4.

[9] The information in this section is taken freely from ibid.

4.1 Reinforcement learning

Reinforcement learning is a type of machine learning where a program tries to find a solution for a problem, and is taught how to do this, by getting rewards and punishments. It is based upon how one might train an animal with treats and such, to do a trick.

4.1.1 The Principle

The logic behind reinforcement training is simple enough. When training a dog, for example, teaching it to do a trick, you might follow the following procedure:

1. Tell the dog how to do the trick, likely showing it what it is supposed to do, somehow.
2. Give the order for the trick. If the dog does it well enough, it gets a reward. This reward could be a treat, for example.
3. Repeat the previous step until the dog manages to do it consistently.

This might teach a dog to sit, or roll around. For such a simple trick, that is fine, but what if you wanted to teach a dog to navigate a maze? Dogs have memories, so even if you only give them treats when they reach the end, they might remember which route they took, and take it again to get treats. All this is well and good, but a program is not a dog, so its methods will differ from a dogs.

4.1.2 The Proper

The way a program learns in reinforcement learning, is by trying somewhat randomly, and then compile statistics about what reward it can expect from a certain action. It might have, for example, a matrix containing the expected reward from moving between states. The algorithm works with the concept of episodes, which are the program finishing a run, when the program has found its way through a maze. A more proper procedure might look like this:

1. The amount of episodes is, or has been chosen.
2. Repeat for each episode.
   (a) The program starts in some start position, like in the middle or at the entrance of a maze.
   (b) A move is chosen, usually either a random move, or the move the program thinks is best. The one with the highest expected reward.
   (c) The move is made, and the expected reward of the move is updated based upon the viability of the possible next move. If the program now sees the exit of the maze it was a good move.

(d) Now the program is in a new state. If the program has reached the end, escaped the maze, then the episode is done, if not, the program is back to step (b) again.

4.1.3 Action Selection/Policy

The action selection, or policy, is how the program chooses which move to make when training. There are several ways to do this, but two of them are:

- Greedy, where the program always picks the move, or action, which gives the highest expected reward.
- ϵ-greedy, which is the same as greedy, except there is a chance to instead make a random move.

If only the Greedy selection is used then the program might get stuck at a less than optimal solution, because it never tries something other than what it knows. This is why ϵ-greedy is the one used further on in this paper.

4.1.4 Sarsa and Q-learning

There are two relevant specific algorithms for training in reinforcement learning, the two used in this paper. All of the algorithms are based upon the so called value function, of which there are also two types, the state-value function $V(s)$ which finds the average expected reward for a state, and the action-value function $Q(s, a)$ which finds the expected value of an action, or a move from a state s to state a. Both algorithms use the action-value function.

Q-learning First we have to Q-learning algorithm, which looks like this:[10]

- **Initialize**
  - set $Q(s, a)$ to small random values for all s and a
- Repeat:
  - initialize s
  - repeat:
    - select action a using *epsilon*-greedy or another policy
    - take action a and receive reward r
    - sample new state s'
    - update $Q(s, a) \leftarrow Q(s, a) + \mu(r + \gamma max_{a'} Q(s', a'))$
    - set $s \leftarrow s'$
  - For each step of the current episode
- Until there are no more episodes

[10] Ibid., p. 242.

Sarsa The algorithm assumes that the optimal move will be made in the next state, unrelated to what the actual policy in use is. It is thus considered off-policy. If one wants an algorithm which is instead on-policy, it must be modified slightly, into one which is called Sarsa:[11]

- **Initialize**
  - set $Q(s, a)$ to small random values for all s and a
- Repeat:
  - initialize s
  - choose action a using the current policy
  - repeat:
    - take action a and receive reward r
    - sample new state s'
    - choose action a' using the current policy
    - update $Q(s, a) \leftarrow Q(s, a) + \mu(r + \gamma Q(s', a') - Q(s, a))$
    - $s \leftarrow s'$, $a \leftarrow a'$
  - For each step of the current episode
- Until there are no more episodes

It is called Sarsa because the algorithm use variables: s, a, r, s', and a', thus spelling Sarsa.

4.2 Supervised Learning

Supervised learning is a type of machine learning that relies on the use of training data, that is data that states input and output. This means the program knows what it needs to answer for many inputs, and for it to work it needs to be able to generalize on that data, use it to make a model that will hopefully apply in all cases. The focus will be on neural nets, because that is what is used later on in this paper.[12]

Neurons A neuron is, put simply, a kind of valve for signals. It takes the sum of all its incoming signals, inputs, and if the sum is above some threshold it fires off, sending out a signal of its own, if the sum of the input is below the threshold, it does not fire. McCulloch and Pitts made a simple mathematical model of a neuron[13], which is used as a basis to make more advanced neural nets. It looks like Figure 1, and calculates output in the following procedure:

[11] Ibid., p. 243.

[12] The information in this section is taken freely from ibid.

[13] Note that it is a simple mathematical model, it isn't particularly accurate, and doesn't emulate the more strange behaviours of nerve-cells.

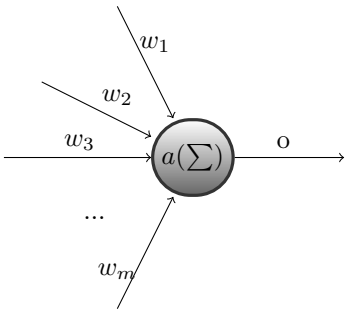

Figure 1: A model of a McCulloch Pitts neuron.

1. get the sum of the inputs times their weight. $\sum_{i=1}^{m} w_i x_i = s$
2. apply the activation function with the sum as input. $a(s) = o$
3. The result, o, is output through the synapse.

The activation function is typically a threshold function, that is, a function which returns almost exclusivly a number close to either 1 or 0. This models biological neurons, which can either send a signal, or not send a signal. This activation function does not have to be a threshold function, however, in which case the neuron might behave somewhat differently than a biological neuron, for example sending output like 0.5, which does not make sense from a biological perspective.

4.2.1 The Perceptron

The perceptron is a simple neural net, that is it is a network of neurons. It essentially is a row of neurons, that all get the same inputs, but the weights are all different[14]. Figure 3 shows a model of a perceptron. Note that there is just 1 layer of neurons, the darker nodes, but those neurons are the McCulloch Pitt neurons mentioned previously.

Bias If the input for a perceptron was all 0, the output would be the same no matter what the weights were, since $x * 0 = 0$. The bias is included so the perceptron can change the output for input of all 0's. The bias is simply another input which always has the same value. The value can be anything non-zero, but 1 or -1 are usually chosen for simplicity's sake, though it does not matter as the weight could be adjusted accordingly.

Training The perceptron can be trained, for each weight it should be updated like in equation 1, where x_i is the input from node i, y_j is the output from neuron

[14]Usually. They can in theory all be the same, or at least some be the same, it is just rather unlikely unless they are initialized as such

j, and t_j is the expected output from neuron j. η is the training variable, which is just adjusted as needed to train as one wish.

$$w_{ij} \leftarrow w_{ij} - \eta(y_j - t_j) * x_i \tag{1}$$

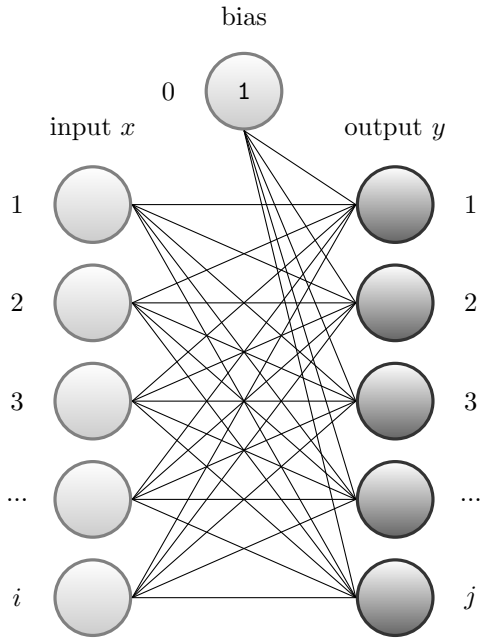

Figure 2: A model of a single layer Perceptron. Note that only the darker nodes are actual neurons, the lighter ones are there to show how the inputs are given to all neurons.

Limitations A perceptron can find patterns, and fire for something that belongs to a class, and not fire for something that doesn't belong to a class. It is constrained to separating them with a straight line, if in 2D, and thus is quite constrained in what it can identify.[15] Suffice to say that it isn't very powerful, and not very useful. There is a solution to this however, simply have more layers. That solution does have some problems, specifically, how to train a so called Multilayer perceptron.

4.2.2 The Multilayer Perceptron

To make the perceptron more useful, we now have another layer. The problem with doing this is that the algorithm needed to train it is rather more complicated than before. The proper equations will be the same used later on when reviewing the code, so for now a more abstract explanation should suffice:

[15]Marsland, *Machine learning: an algorithmic perspective*, op. cit., p. 55.

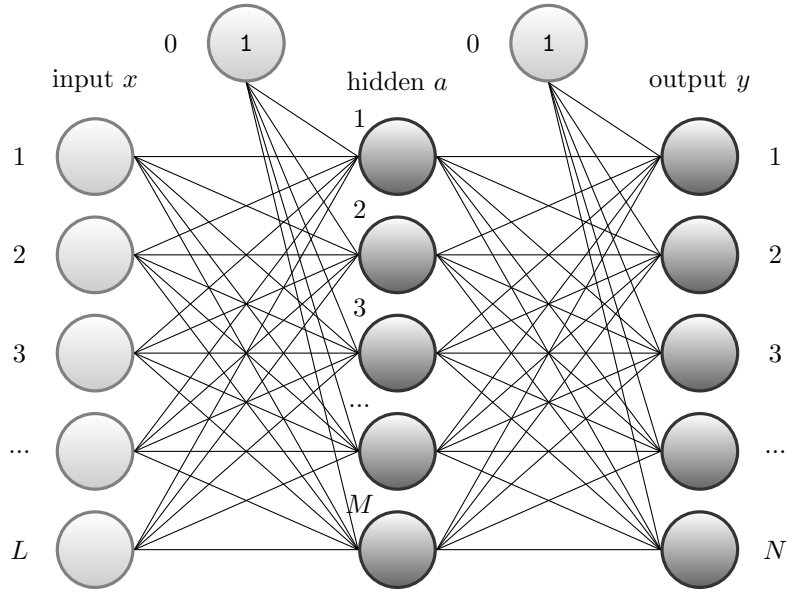

Figure 3: A model of a MultiLayer Perceptron.

1. An input is given to the input nodes.
2. The inputs are fed forwards through the network.
   (a) The hidden layer get the inputs, and like the single layer perceptron, neurons take the inputs times the weigths, and applies that sum to the function. The result is given as input to the output layer.
   (b) The output layer does the same, only with the data from the hidden layer instead.
3. Calculate the error of the output.
   (a) The weights between the output and hidden layer are updated, based upon the error, and their input.
   (b) Then the weights between the hidden and the input layer are updated, using error inherited from the output layer, modified by the weights. The bigger value a neuron gave, the bigger amount of error it inherits.

The proper algorithm[16] The actual algorithm is somewhat more complex, and use the following symbols: i, j, k is the indices of the layers respectivly, their sizes being L, M, N respectivly, with the corresponding greek letter ι, ζ, κ for fixed indices. Weights between the input and hidden layer is called v, and the

[16] Ibid., p. 77.

weights between the hidden and output layer is called w. The input nodes are x_i, the bias node is x_0, the hidden nodes are a_j, the bias node a_0 and the output layer is y_k. The targets for the outputs, what the mlp should output, is t_k. The proper algorithm for the Multilayer perceptron will then be:

The Multilayer Perceptron Algorithm[17]

- **Initialization**
  - initialize all weights to small (positive and negative) random values
- **Training**
  - repeat:
    - For each vector:
      Forwards phase:
      - compute the activation of each neuron j in the hidden layer(s) using:

$$h_\zeta = \sum_{i=0}^{L} x_i v_{i\zeta} \tag{2}$$

$$a_\zeta = g(h_\zeta) = \frac{1}{1 + exp(-\beta h_\zeta)} \tag{3}$$

      - work through the network until you get to the output layer neurons, which have activations[18]

$$h_\kappa = \sum_{j=0}^{M} a_i w_{i\kappa} \tag{4}$$

$$y_\kappa = g(h_\kappa) = \frac{1}{1 + exp(-\beta h_\kappa)} \tag{5}$$

[17] Ibid., p. 78.

[18] The actual equation in the source is $h_\kappa = \sum_j a_i w_{i\kappa}$ but that is lacking a couple of symbols, as j needs constraints.

Backwards phase:

· compute the error at the output using:

$$\delta_o(\kappa) = (y_\kappa - t_\kappa)y_\kappa(1 - y_\kappa) \tag{6}$$

· compute the error in the hidden layer(s) using:

$$\delta_h(\zeta) = a_\zeta(i - a_\zeta)\sum_{k=1}^{N} w_\zeta \delta_o(k) \tag{7}$$

· update the output layer weights using:

$$w_{\zeta\kappa} \leftarrow w_{\zeta\kappa} - \eta\delta_o(\kappa)a_\zeta^{hidden} \tag{8}$$

· update the hidden layer weights using[19]:

$$v_{\iota\zeta} \leftarrow v_{\iota\zeta} - \eta\delta_h(\zeta)x_\iota \tag{9}$$

∗ (if using sequential updating) randomise the order of the input vectors so that you don't train in exactly the same order each iteration

– until learning stops

- **Recall**
  - use the Forwards phase in the training section above

5 The Algorithms for the Problem

5.1 Preexisting Algorithm

This is the algorithm which already exist for solving Nim, and this is how it works. This algorithm is deterministic, that is, it will give the same result and follow the same steps each time, if given the same input twice.

5.1.1 Nim-sum

The algorithm uses an operation called Nim-sum, which is really equivalent to the exclusive-or operation if the numbers are converted to binary. I shall use $\oplus$ as the symbol for this operator for every digit in the two numbers, the corresponding digit in the Nim-sum will be 1 if they are different, and 0 if they are the same. Table 1 is an example of a nim-sum operation. When finding the Nim-sum of more than 2 numbers all that is done is to get the Nim-sum of the first 2 numbers, then find the Nim-sum of that and the third number, continuing that until finished.

The book "Winning ways for your mathematical plays"[20] has the following suggestion for how to calculate a nim-sum if you are a human, not a computer:

[19] The actual equation in the source is $v_\iota \leftarrow v_\iota - \eta\delta_h(\kappa)x_\iota$ but that doesn't make sense, as a weight is between 2 nodes, but here the weights aren't like that.

[20] Conway, Berlekamp, and Guy, *Winning Ways for Your Mathematical Plays*, op. cit.

Table 1: A Nim-sum example

Binary	**base10**
00100110⊕	38⊕
00001101	13
=	
00101011	43

The Basics of Nim-addition You can use these two basic properties to find the nim-sum of any collection of numbers by writing each of them as a sum of distinct powers of 2 and then cancelling repetitions in pairs. For example,

$$5 \oplus 3 = (4+1) \oplus (2+1) = 4 \oplus \cancel{1} \oplus 2 \oplus \cancel{1} = 4 \oplus 2 = 4 + 2 = 6,$$

$$11 \oplus 22 \oplus 33 = (8 + \cancel{2} + \cancel{1}) \oplus (16 + 4 + \cancel{2}) \oplus (32 + \cancel{1}) = 8 + 16 + 4 + 32 = 60.$$

Table 2: This table shows the decision process for each digit in the nim-sum operation.

only one digit	only one digit	different digits	same digits	different digits	same digits	
1	0	0	1	0	1	
		1	1	1	1	⊕
1	0	1	0	1	0	=

5.1.2 Nim-sum of the State

the crux of the algorithm is to find the Nim-sum of the current state of the game. It will then be used to figure out whicht move is the correct to make, unless the Nim-sum is 0, because then there is no good move. The Nim-sum of a game-state is simply the Nim-sum of all the sizes of the heap's in the game. For the game-state 3 5 7 the Nim-sum is $3 \oplus 5 \oplus 7 = x$. Table 3 shows the calculation with the answer: 1.

Table 3: Nim-sum of the state 3 5 7

Binary	**base10**
011⊕	3⊕
101⊕	5⊕
111	7
=	
001	1

5.1.3 Finding the Correct Move

To find the correct move one should get the nim-sum of the state, and find the Nim-sum of that and the size of a heap. If the resulting sum is less than the heaps size, the correct move is the move reducing the heap to the Nim-sum. If the sum is not smaller than the heap size, the same procedure should be done with another heap. As long as the Nim-sum of the stat is not 0, a move can be found. In Table 4 a correct move is found in the case of game-state 3 5 7 which has a Nim-sum of 1. The Nim-sum is 2, which means a correct move would be to reduce the heap of size 3 to size 2. There are other correct moves here too, however. For example: $1 \oplus 5 = 4$, so reducing the heap of size 5 to size 4 is also a correct move.

Table 4: Nim-sum of Table 3s Nim-sum and the first heap in the game-state.

Binary	**base10**
001$\oplus$	1$\oplus$
011	3
=	
010	2

5.2 Reinforcement Learning

This second algorithm is a machine learning algorithm, and it is, as opposed to the previous one, stochastic.

5.2.1 Representation of Nim as a graph

Reinforcement learning usually works with a graph, to find the best way through the graph. Therefore the game of Nim should be represented as a graph so that the algorithm might be more easily applied to this problem. All games can easily be represented as trees, where all the children of a node represents all the legal moves possible to make from the parent node. See the first graph in Figure 4. This is not very space-efficient however, as there can be several nodes which are identical, which could be eliminated for a slimmer graph. As small a graph as possible is desired, specifically fewer edges, so less training is needed. The amount of nodes will mainly determine the space-use of the program. The graph is directed and has nodes and edges, where the a node is a state the game can be in, that is a set of heaps, each with some amount of counters in them, including 0 counters. There is an edge from one node to another, if there is a legal move in the first state which results in the second state. The node which does not have any edges going into it is the node containing the start state, the state which is given as the input for the algorithm. The node with only incoming edges is the winning condition, that is, no counters left in any heap.

This is the graph which the first iteration of the algorithm uses, see the second graph in Figure 4.

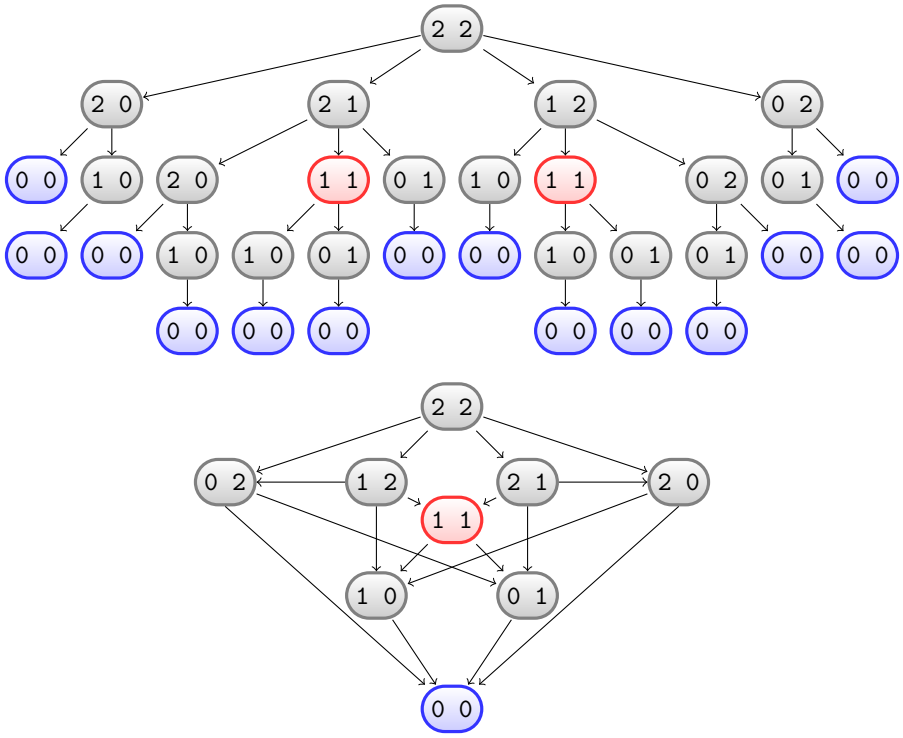

Figure 4: First graph is the graph for the game tree of a game of Nim, starting with the state 2 2. Notice how many times state 0 0 (blue) occurs. State 1 1 (red) occurs twice as well. The second graph is the resulting graph if all equal state-nodes in the game tree is combined. Notice how the states 0 0, and 1 1 occur just once in this graph.

In the first graph in Figure 4 the node 1 1, in red, occurs twice, and both have the same amount of edges going out from them. In the second graph in Figure 4, where there is only one occurance of 1 1, there are also less edges in total going out of occurances of 1 1, as well as there being less occurances overall. This does not apply to all nodes however; node 0 0, in blue, never has any edges going away from it, so even though it is the most common node in the first graph Figure 4 removing duplicates of it will not as such remove edges, it will still make the total memory requirement smaller.

Amount of nodes the amount of nodes in a graph is calculated using equation 10 where h_k is the size of the kth heap

$$(h_0 + 1) * (h_1 + 1) * (h_2 + 1) * ... * (h_k + 1) = amount\ of\ nodes \tag{10}$$

5.2.2 How the Graph Is Represented in the Program

This is a matrix showing all legal moves from a state in the vertical list of states, to a state in the horizontal list of states. The program gets the second graph in figure 4, modelled as a matrix, where the indices on both axis corresponds to one state of the game, or one node of the graph. Index 0 is the starting position which the program gets as an input, while index 1 is the state you get when one counter is removed from the last heap of the state in index 0. Each field represents whether there is an edge from the state on the index on the horizontal axis, to the state on the index in the vertical axis. the field is of the value 1 if there is an edge, and of value 0 if there isn't an edge. The program also has a second matrix of the same size, which will be used to record expected reward for a certain move.

Table 5: The graph as a matrix

states	2 2	2 1	2 0	1 2	1 1	1 0	0 2	0 1	0 0
2 2	0	1	1	1	0	0	1	0	0
2 1	0	0	1	0	1	0	0	1	0
2 0	0	0	0	0	0	1	0	0	1
1 2	0	0	0	0	1	1	1	0	0
1 1	0	0	0	0	0	1	0	1	0
1 0	0	0	0	0	0	0	0	0	1
0 2	0	0	0	0	0	0	0	1	1
0 1	0	0	0	0	0	0	0	0	1
0 0	0	0	0	0	0	0	0	0	0

5.2.3 How the Graph Is Made Smaller

The graphs are much better than the trees, but there is still overlap between the nodes. For example the nodes 2 1 and 1 2 are essentially the same, the only difference is that they are mirrored. So there is still something that can be done to make the graph even smaller, by only having one ordering of any state, that is, only 2 1 and not 1 2. I have decided that the correct ordering is from biggest to smallest. So 2 1 1 is allowed, but not 1 1 2, nor 1 2 1. The graph for 2 2 will then look like the right graph in Figure 5.

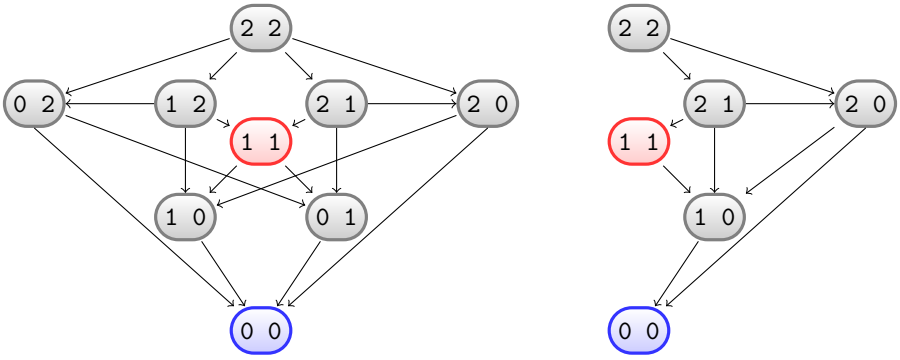

Figure 5: Graph to the left is the untrimmed graph, the one to the right the trimmed one. See how the trimmed one has almost been cut in two? It is much smaller, another improvement to the graph.

Equation 11 is used to calculate the amount of possible states reachable in a game of nim, provided all the heaps are of the same size. n is the size of the heaps, and k is the amount of heaps.

Amount of nodes in a trimmed game-tree

$$\frac{(n+k)!}{n! * k!} \tag{11}$$

Combinations with repetition

$$\frac{(n+k-1)}{k! * (n-1)!} \tag{12}$$

This eqation is almost the equation for finding combinations with repitition, which it is based upon. This makes sense, because of the number needed can be considered to be the following: for each heap draw a number between and including the size of the heaps and 0. If the order is not important, how many different combinations of these are there. Since the amount of things to draw is actually 1 more than the size of the heaps, because 0 can also be drawn, the equation is changed a little, because n is $amount\ of\ heaps + 1$ and n is always preset with a -1 which nullifies each other.

Problems with the trimmed graph The big problem with the algorithms for the trimmed graph is that they can actually only take start-states with heaps of the same size, as it needs to know the size of the graph from the start. This is not so bad, however, the benefits of using this trimmed graph far outweighs any restrictions, see section 8.

5.2.4 Training

The algorithm learns what moves to make by training, which consists of trying moves, and updating the expected rewards for those moves. The algorithm essentially plays games of Nim against itself, and modifies the afformentioned second matrix to get a better estimation of how good certain moves are, but there is a difference from the common algorithm for reinforcement learning. The difference is that while usually the expected reward for a move is modified by adding it to the expected reward of some possible next move(depending on the variant), because moving to a good position is usually desired for it moves you closer to some goal. In this situation, and for a similar algorithm for other similar games, a move to a good position is not desirable, because that means the opponent gets to move from a good position. In this case instead of adding the expected reward for some next move, it is subtracted. This would only apply to games where both players can make the same moves, but not games like chess where each player have different pieces. The assignment looks like this: $Q(s,a) \leftarrow Q(s,a) - \mu(r + \gamma Q(s',a') - Q(s,a))$

To make it more clear, here is what happens, in the correct order.

1. The amount of episodes is, or has been chosen. For example 1000 episodes.
2. An episode is started, and the number of episodes is incremented.
   (a) An episode starts on the predefined start position. In Table 5 it is 2 2.
   (b) With the current state of the game, a move is chosen. It will usually pick the best move, that is the one with the highest expected reward, but there is a chance for it to be picked randomly.
   (c) The move is made, and the expected reward of the move is updated, based upon the move the next player is expected to make. If that next move is good, the reward will be decreased if not appropriate.
   (d) The new state, which is reached by the chosen move, is now considered the current state. If the current state is of a finished game, that is, it consists of only empty heaps, then the episode is done. If not, go back to step (b) with the new current state.
3. If the number of episodes has reached the predefined amount, in this case 1000, the program is done. If the amount is not reached, go back to step 2.

Terminology From this point on the programs that use the untrimmed graph, the one that combined the nodes in the game-tree that were the same,will be refered to as the untrimmed programs. The graph will also merely be called the game-tree. The programs that use the trimmed graph where the order is not enough for difference are called the trimmed programs. Any other further programs will be named for their unique features.

5.2.5 Terminating the Program

The program runs a finite amount of episodes before it terminates. The amount of episodes is configurable in the code. This wasn't the only possibility however, the program could also have ran on until it was sufficiently trained(whatever that is). These are the options of a logical or stochastic termination, the logical is predetermined, and the same each time the program is run, and the stochastic is(likely) different each time the program is run. The logical termination was chosen because of its predictability, as well as the sizes of the game-trees for larger games: the amount of episodes of training that is necesarry gets so large the program runs for an impractical amount of time. Since I wish to easily make statistics, logical termination seems appropriate, with the added benefit of the amont of episodes being controlled as well. A test-program that features stochastic termination will be featured later in the paper, and the results should support my decision.

5.2.6 One-dimensional Program

Since the states in Nim have the same legal moves for both players, and the goal for finding a good move is to find a good state to move to, perhaps an improvement would be to base a program on the use of the state-value function $V(s)$ instead of the action-value function $Q(s, a)$. Since it uses the one-dimensional $V(s)$, some parts of the code must be changed, but overall the program should be quite similar. It shall be called OneDSarsa, as it uses the Sarsa algorithm adapted to $V(s)$, which is one-dimensional.

5.3 Supervised Learning

This algorithm relies on using data on correct moves to learn how to play.

5.3.1 Acquiring Data

Acquiring data is simple enough, since there exists an algorithm for making correct moves, so a way would simply be to have a list of states, and record what the algorithm outputs for those states. Another way would be to use the reinforcement algorithm to get some positions.

5.3.2 Training

The program trains by modifying the weights in a graph, or network, called a neural network, as have already been explained. The algorithm works somewhat like this, see figure 6 for an example neural net:

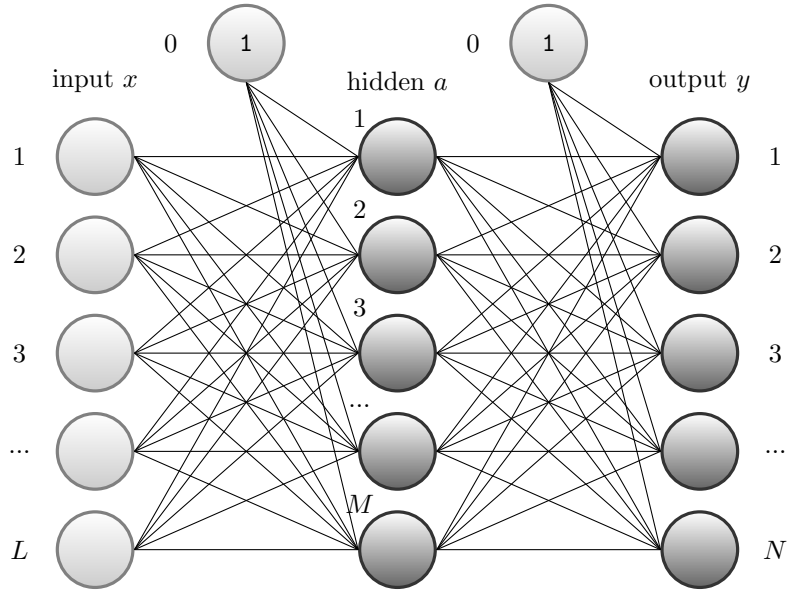

Figure 6: A simple model of a Multilayer Perceptron.

1. The weights are initialized with small random positive or negative numbers.
2. Repeat:
   (a) Repeat once for each line of data in the training set:
      i. Give the current line of training data as input.
      ii. Calculate inwards to the hidden layer, through each synapse. From each synapse multiply the input value with the weight, and add it to the output node.
      iii. Calculate inwards to the output layer, through each synapse. From each synapse multiply the input value with the weight, and add it to the output node.
      iv. Compare the resulting output with the expected output and calculate error.
      v. Go backwards from the output layer to the hidden layer and adjust weigths based on their error.
      vi. Go backwards from the hidden layer to the input layer and adjust weights based on their error.
   (b) Check the total error of the neural net.
3. If the error is higher than last iteration, do not repeat. Training is done.

Total error To reiterate, the total error of the neural net is checked by comparing the result of some input and the expected correct result. Equation 13 is the equation for the total error of the output layer of a neural net, where N is the amount of nodes, or perceptrons in the output layer.

$$\sum_{k=1}^{N}(y_k - t_k)^2 \tag{13}$$

6 Describing the Code

Having given a more abstract explanation on how the algorithms work, it is time to take a more indepth look into the actual programs i have written[21].

6.1 The Deterministic Algorithm

The code for the deterministic algorithm is simple enough. It takes the state of the game as input, and returns the correct move as an array of size 2 containing firstly how many counters should be removed, and secondly from which heap the counters should be removed from.

```
class PerfectPlayer:
  #This method applies the deterministic algorithm for making
    perfect moves in Nim, and returns the index of the move and the
     amoiunt of counters removed.
  def makeMove(self, state):
    nimSum = self.findNimSum(state)
    for i in range(len(state)):
      remove = nimSum^state[i]
      if(remove < state[i]):
        result = [state[i] - remove, i]
        return result
    for i in range(len(state)):
      if(state[i] != 0):
        result = [1, i]
        return result
    return -1

  #Since finding nimsums is relevant in the program Run.py this
    program finds the nimsum of the input state.
  def findNimSum(self, state):
    nimSum = 0
    for i in range(len(state)):
      nimSum = nimSum^state[i]
    return nimSum
```

Listing 1: Algorithm for finding the correct move from a certain state in Nim

[21] There are some parts of the code where some outside inspiration has had its part, though those parts will be mentioned as such, when relevant.

The program could have returned the new size of the heap instead, however that seemed less natural, as it feels like a move should be considered to remove a certain amount of counters, as that is what one would do when playing the physical game.

6.2 Programs for Reinforcement Learning.

I have written five programs that use reinforcement learning. Two that use the untrimmed game-tree, and two that use the trimmed tree, and one that has a one-dimensional array for the expected rewards, that is it has an expected reward for a state, rather than a move. It also uses the trimmed game-tree. There are some functions they have in common, so they will be mentioned. I will mention them in the same order as here.

6.2.1 Initialization

There are two versions of the initialization. One in the programs using the untrimmed tree, and one in the programs using the trimmed tree.

Untrimmed The initialization for these programs looks like this:

```
8  def __init__(self, startBoard):
9    self.sum = startBoard[0]+1
10   self.board = startBoard
11   for i in range(1, len(self.board)):
12     self.sum *= self.board[i]+1
13   self.listOfStates = [0] * self.sum
14   self.listOfStates[0] = self.board[:]
15   self.T = []
16   self.Q = []
```

Listing 2: Initialization of the programs using the untrimmed game-tree

The important part is to find the sum, which is really the amount of nodes in the game tree, or how many possible states can be reached from the starting state. In the untrimmed game tree it is easy to calculate, as shown before. see Equation 14. T and Q are the matrices for legal moves and expected rewards from moves repsectivly, and listOfStates is appropriatly enough the list of possible states, with the rule that any move that can be made from a state, is only to states below it in the list.

$$(h_0 + 1) * (h_1 + 1) * (h_2 + 1) * ... * (h_n + 1) = amount\ of\ nodes \tag{14}$$

Trimmed This only accepts input boards of the type where all the heaps are of the same size. The initialization for these programs looks like this:

```
def __init__(self, startBoard):
  self.board = startBoard
  self.sum = self.getSum()
  self.board.sort(reverse = True)
  self.listOfStates = []
  self.listOfStates.append(self.board[:])
  self.T = []
  self.Q = []
```

Listing 3: Initialization of the programs using the trimmed game-tree

This is almost the same as the last one, expcept that the sum is calculated differently, it has its own function. the function just uses Equation 15 to calculate the sum.

```
def getSum(self):
  return int(math.factorial(len(self.board)+ self.board[0])/(math.
    factorial(len(self.board))*math.factorial(self.board[0])))
```

Listing 4: A function for applying Equation 15 to find the amount of states reachable from the start state.

$$\frac{(n+k)!}{n! * k!} \tag{15}$$

6.2.2 Setup

The setup is also divided by the game-trees, but the program with the one-dimensional array for expected rewards are also separate here.

Untrimmed The setup is for properly making and filling the matrices T and Q, so that training can start. It runs the function to start training, it also records the time setup and training use for statistical purposes.

The List of States All the programs use a list of states for several purposes, for example, when it is filled, that also functions as a method for filling T with proper data on the legal moves. It is also necessary for when you make moves. The problem is that in the setup of all these programs, the list is filled by making all legal moves from each state, by making every legal move from each state. This also fills the list of states, which is the source of the problem. The problem is whether, when some index of the list is reached, has that state been filled in? The following theorem shall show that with a certain way of ordering the list, this is the case. The list shall be ordered in such a way that each heap, except the right most, the order is so that for state $h_0, ..., h_i, ..., h_n$, for each state h_i to decrement, state h_{i+1} must decrement until it is empty. since there is no state h_{n+1}, state h_n can decrement without any restrictions. The order created by that, when decrementing h_0 is the order list of states should have. In the case of the trimmed tree, the order is the same, except with any state out of order removed. Another way to state to question is whether some state does not have

states above it in the list, that has legal moves to it. The only exception would be the start state. I can do better, it can be proven that no state is, in the list, under any state it has a legal move to.

Theorem: All States Are Above Any State to Which They Have a Legal Move This can be proven by proof of contradiction. Let us assume that some state has a legal move that ends up in a state above it in the list. In that case we have a state s: $h_0, ..., h_i, ..., h_n$, and the result of the move, state s' $h_0, ..., h_i - x, ..., h_n$, where $x > 0$ & $x =< h_i$, which is earlier in the list. This is a contradiction, because the order is decided by decrementing heaps, and since h_i is bigger than $h_i - x$, and all the other heaps are the same, state s must by definition be above state s'. This is a contradiction, thus the assumption is wrong. For when the list is the trimmed one, the same proof almost holds. The difference is that the result of the move does not have the heaps in the same place as they had earlier, for example the move from 2 2 1 to 2 1 0. So the logic is almost the same, you have state s and s', in the formats $h_0, ..., h_i, ..., h_n$ and $h_0, ..., h_i - x_0, h_{i+1} - x_1, ..., h_n - x_{n-i}$ respectivly, where $0 < \sum_{y=0}^{n-i} x_y <= h_i$. This causes a contradiction, because there will be at least one case of $h_j > h_j - x_{j-i}$, which like the previous time, means the state s should be earlier than s' according to the predefined order.

```
23 def setup(self, type):
24   timeStart = time.time()
25   for i in range(self.sum):
26     self.T.append([0] * self.sum)
27     for x in range(len(self.listOfStates[i])):
28       for y in range(1, self.listOfStates[i][x] + 1):
29         tmpValue = self.newIndex(x, y)
30         self.T[i][i+tmpValue] = 1
31         if(self.listOfStates[i+tmpValue] == 0):
32           tmpList = self.listOfStates[i][:]
33           tmpList[x] -= y
34           self.listOfStates[i+tmpValue] = tmpList[:]

36     self.Q.append([])
37     for j in range(self.sum-1):
38       if(self.T[i][j] == 1):
39         self.Q[i].append(random.random())
40       else:
41         self.Q[i].append(-np.inf)
42     if (self.T[i][self.sum-1] == 1):
43       self.Q[i].append(100.0)
44     else:
45       self.Q[i].append(-np.inf)
46   timeEnd = time.time()
47   setupTime = timeEnd-timeStart
48   if(type == 0):
49     epsilon = 0
50     mu = 0.5
51     gamma = 0.5
52     amountEpisodes = 1000
53     timeStart = time.time()
54     self.train(amountEpisodes, epsilon, mu, gamma)
```

```
55      timeEnd = time.time()
56      trainTime = timeEnd-timeStart

58      times = []
59      times.append(setupTime)
60      times.append(trainTime)

62      return times
63    return -1
64


66  def newIndex(self, heapIndex, amount):
67    newMove = 1
68    for i in range(len(self.board)-1, heapIndex, -1):
69      newMove *= self.board[i]+1
70    newMove *= amount
71    return newMove
```

Listing 5: The setup function for the untrimmed programs.

It works by going through the list of all states, adding all states directly reachable from the current state in 1 move, and adding that as a legal move in T, and filling in the new states in the list listOfStates. Since all states in the list except the first one has to be reachable from some other state in the list, we can conclude that when an arbitrary states index is reached, it has been added by some previous state. knowing which index a state belongs to is quite simple, and works as shown in Equation 16 where k is the index of the last heap, b is the size of a heap before, and a is the size after the move has been made. s is a heap as it is in the startstate. The function *newIndex* simply gets a move as input, and calculates what the index of the new position is, relative to the state the move was made from. That is used to calculate where the new states should be, and if that position is empty, that is the state has not already been added, it will be added.

$$i = (b_k - a_k) + (b_{k-1} - a_{k-1}) * (s_k + 1) + ... + (b_1 - a_1) * (s_k + 1) * (s_{k-1} + 1) * ... * (s_1 + 1) \tag{16}$$

When the matrix T is made, the program iterates through it, and makes the Q matrix based upon it. For any legal move, a small random number between 1 and -1 is chosen, and for illegal moves the value negative infinity is chosen[22]. The expected reward to move to the winning state starts off as 100, so that the program knows to go there. The setup function then sets the values for the training variables mu and gamma used by the training function, as well as epsilon for the ϵ-greedy choice, and lastly the amount of episodes.

[22] Negative infinity is a feature of numpy, it is only used so that when picking the move with the highest expected reward, an illegal move is never picked, as negative infinity is always smaller than any number.

Trimmed The setup function for these programs are similar, from the last line in the function in Listing 6 to the end of the function they are the same. It uses a function called *newState*, which takes a state and a move, and gives the resulting state, and gives it back, ordered correctly.

```
def setup(self):
  timeStart = time.time()
  emptyBoard = [0] * len(self.listOfStates[0])
  for i in range(self.sum):
    self.T.append([0] * self.sum)
    for x in range(len(self.listOfStates[i])-1, -1, -1):
      atEnd = 0
      curIndex = i+1
      for y in range(1, self.listOfStates[i][x]+1):
        newState = self.newState(x, y, self.listOfStates[i])
        if(atEnd):
          self.T[i][curIndex] = 1
          self.listOfStates.append(newState)
          curIndex += 1
        else:
          atEnd = 1
          for j in range(curIndex, len(self.listOfStates)):
            if (newState == self.listOfStates[j]):
              self.T[i][j] = 1
              curIndex = j+1
              atEnd = 0
              break
          if (atEnd):
            self.T[i][len(self.listOfStates)] = 1
            self.listOfStates.append(newState)
            curIndex = len(self.listOfStates)
    self.Q.append([])


def newState(self, heapIndex, amount, state):
  curState = state[:]
  curAmount = amount
  for i in range(heapIndex, len(state)-1):
    dif = curState[i] - curState[i+1]
    if (dif >= curAmount):
      curState[i] -= curAmount
      curAmount = 0
      break
    else:
      curState[i] -= dif
      curAmount -= dif
  if (curAmount > 0):
    curState[len(curState)-1] -= curAmount
  return curState
```

Listing 6: The setup function for the trimmed programs.

Since at the time of writing the code I knew of no way to calculate the index of a state when using the trimmed game-tree[23], the creation of the matrix T

[23]See Equation 17 for how the calculation is done.

works a bit differently. It works like this:

- Repeat for all possible states:
  - Repeat for all heaps in current state, starting at the last heap, keeping track of the current index of where the possible moves could go, it is reset for each heap to be the index of the current state + 1:
    - Repeat for all counters in current heap:
      1. Use the function *newState* to find what the resulting state of the new move will be.
      2. Is there a defined state at the current index in the current column in T? if no, add the resulting state at the end of listOfStates, and update the spot for the move in T to 1, if yes, go through the states on and after the index, to see if any of the states are the same as the resulting state. if they are, update T to 1 on the relevant spot. If no such state is found, the state is added at the end of listOfStates, and T is updated, once again in the same fashion.

The function *newState* could be made simpler by just having it make the move as if it was the untrimmed game-tree, and then sort the new state.

One-dimensional This program only has a small difference to the previous programs, since they both use the trimmed game-tree, as well as the same matrix T. Listing 7 shows the part of the code which is different from the trimmed programs, the first and last line it has in common.

```
70              curIndex = len(self.listOfStates)
71      if(i == (self.sum-1)):
72        self.Q.append(100.0)
73      else:
74        self.Q.append(random.random())
75    timeEnd = time.time()
```

Listing 7: The part of the setup function which is different in the One-dimensional program.

6.2.3 Training

Here the difference is not in the game tree, as the important part there is how the matrices T and Q are made, but rather the algorithm used to train, which is unrelated to how you make the matrices.

An Error in the Code There is an error in the code for the training, which is that for the variable r, which is the actual reward for making a move, the programs instead use the expected reward. This is not a big problem, because the actual reward for all moves except the winning one is 0, so the result is

just that some rewards are somewhat exaggerated, but it shouldnt impact the training. Having done some tests, the difference seems negligible in this case. This could easily hinder training in other problems, however, so I shall strive not to make that error again.

Q-learning These programs use this algorithm, as expected, except for one thing, that is the very adjustment of the expected reward. It isnt added to the current value, it is subtracted. This is based upon the logic that giving your opponent the option of a good move is bad, and forcing the opponent to make a bad move is good.

```
95  def train(self, amountEpisodes, epsilon, mu, gamma):
96    for neverUsed in range(amountEpisodes):
97      curState = 0
98      playing = 1
99      while(playing):
100       curAction = self.epsilonGreedyChoice(curState, epsilon)
101       curReward = self.Q[curState][curAction]
102       nextState = curAction
103       nextAction = np.argmax(self.Q[nextState])
104       self.Q[curState][curAction] -= mu * (curReward + gamma*self.Q
      [nextState][nextAction] - self.Q[curState][curAction])
105       curState = nextState
106       if (nextAction == self.sum-1):
107         playing = 0
108
109
110 def epsilonGreedyChoice(self, state, epsilon):
111   if(random.random()<epsilon):
112     tmpList = []
113     for i in range(state, self.sum):
114       if(self.T[state][i] == 1):
115         tmpList.append(i)
116     return tmpList[random.randrange(0, len(tmpList))]
117   else:
118     return np.argmax(self.Q[state])
```

Listing 8: The code for the Q-learning algorithm. Note that linenumbers are from the untrimmed program.

Sarsa This is the code for the Sarsa algorithm, but like the Q-learning programs it subtracts, instead of adding the adjustment of expected reward.

```
95  def train(self, amountEpisodes, epsilon, mu, gamma):
96    for neverUsed in range(amountEpisodes):
97      curState = 0
98      curAction = self.epsilonGreedyChoice(curState, epsilon)
99      playing = 1
100     while(playing):
101       curReward = self.Q[curState][curAction]
102       nextState = curAction
103       nextAction = self.epsilonGreedyChoice(nextState, epsilon)
104       self.Q[curState][curAction] -= mu * (curReward + gamma*self.Q
      [nextState][nextAction] - self.Q[curState][curAction])
```

```
105       curState = nextState
106       curAction = nextAction
107       if (curAction == self.sum-1):
108         playing = 0
109
111 def epsilonGreedyChoice(self, state, epsilon):
112   if(random.random()<epsilon):
113     tmpList = []
114     for i in range(state, self.sum):
115       if(self.T[state][i] == 1):
116         tmpList.append(i)
117     return tmpList[random.randrange(0, len(tmpList))]
118   else:
119     return np.argmax(self.Q[state])
```

Listing 9: The code for the Sarsa algorithm. Note that linenumbers are from the untrimmed program.

One-dimensional This program uses the sarsa algorithm, but the code is slightly different, as the Q is One-dimensional. Thus for each occurance of Q it only uses one index, which is the difference. For the sake of completeness, Listing 10 has the code.

```
144 def train(self, amountEpisodes, epsilon, mu, gamma):
145   for neverUsed in range(amountEpisodes):
146     curState = 0
147     curAction = self.epsilonGreedyChoice(curState, epsilon)
148     playing = 1
149     while(playing):
150       curReward = self.Q[curAction]
151       nextState = curAction
152       nextAction = self.epsilonGreedyChoice(nextState, epsilon)
153       self.Q[curAction] -= mu * (curReward + gamma*self.Q[
      nextAction] - self.Q[curAction])
154       curState = nextState
155       curAction = nextAction
156       if (curAction == self.sum-1):
157         playing = 0
158
162 def epsilonGreedyChoice(self, state, epsilon):
163   if(random.random()<epsilon):
164     tmpList = []
165     for i in range(state, self.sum):
166       if(self.T[state][i] == 1):
167         tmpList.append(i)
168     return tmpList[random.randrange(0, len(tmpList))]
169   else:
170     highVal = -np.inf
171     highValIndex = -1
172     for i in range(state+1, len(self.Q)):
173       if(self.T[state][i] == 1 and np.greater(self.Q[i], highVal)):
174         highVal = self.Q[i]
175         highValIndex = i
```

```
176     return highValIndex
```

Listing 10: The code for the Sarsa algorithm, in the One-dimensional program

6.2.4 Making Moves

The programs all have a function called *makeMove*. It takes a state, and returns how an array with 2 values, the first for how many counters should be removed, and the second is from which heap they should be removed from. There is a difference between the programs, once again based upon their use of game-trees, as well as the one-dimensional one being different.

Untrimmed The function in these programs is quite simple, the code in Listing 11 works like this:

1. If the input state has fewer heaps than the start-state the program was trained from, extra heaps of size 0 are added at the end.
2. Find the state we wish to make a move from in the list of states.
3. Find the move with the highest expected reward.
4. Find the index from which the counters were removed, and put the index, as well as the amount of counters removed, into an array which is then returned.

```
76 def makeMove(self, state):
77   curState = state[:]
78   nextState = []
79   move = []
80   if (len(state)<len(self.listOfStates[0])):
81     dif = len(self.listOfStates[0])-len(state)
82     for i in range(dif):
83       curState.append(0)
84   for i in range(len(self.listOfStates)):
85     if (curState == self.listOfStates[i]):
86       nextState = self.listOfStates[np.argmax(self.Q[i])]
87   for i in range(len(curState)):
88     if (curState[i] != nextState[i]):
89       move.append(curState[i] - nextState[i])
90       move.append(i)
91       return move
```

Listing 11: The *makeMove* function in the untrimmed programs.

Trimmed The function is somewhat different in these programs, as finding the correct move is simple enough, but the input state is sorted to fit the program, so finding the correct index for moving is somewhat more difficult, so it is longer and more complex. In addition, finding the state in the list of states is done by calculation, which is faster, though uses more code. The process is this:

Figure 7: The equation for finding the index of a state, in relation to the start state, for a specific move. n is the size of the heaps in the starting state, x is the amount of counters being removed, and k is the amount of piles right if the pile being removed, which would be $length(state) - k'$ where k' is the pile being removed from.

$$x * (\frac{1}{n!} * (k+1) * (k+2) * ... * (k+n)) \tag{17}$$

1. Sort the input state, biggest first, then if the input state has fewer heaps than the start-state the program was trained from, extra heaps of size 0 are added at the end.

2. Find the state we want to move from in the list of states, by calculating its index. this is done by using Equation 17.

3. Figure out the how many counters were removed, and from which index.

4. Find the correct index to be output. Since the input state was sorted, the index is not the same as it was, which means an index of the same value needs to be found in the proper input state.

```
103 def makeMove(self, state):
104   curState = state[:]
105   curState.sort(reverse = True)
106   nextState = []
107   move = []
108   if (len(state)<len(self.listOfStates[0])):
109     dif = len(self.listOfStates[0])-len(state)
110     for i in range(dif):
111       curState.append(0)
112   goal = 0
113   progress = self.board[0]
114   for i in range(len(self.board)):
115     heaps = len(self.board)- (i+1)
116     for j in range(progress, -1, -1):
117       if(j == curState[i]):
118         break
119       else:
120         if(heaps == 0):
121           goal += j-curState[i]
122           break
123         result = 1
124         for x in range(1, heaps + 1):
125           result *= (j + x)
126         goal += (result)//math.factorial(heaps)
127         progress -= 1
128   nextState = self.listOfStates[np.argmax(self.Q[goal])][:]
129   numToDec = 0
130   for i in range(len(curState)-1, -1, -1):
131     if (curState[i] != nextState[i]):
```

```
      numToDec += curState[i]-nextState[i]
      newIndex = curState[i]
  move.append(numToDec)
  for i in range(len(state)):
    if(newIndex == state[i]):
      newIndex = i
      break
  move.append(newIndex)
  return move
```

Listing 12: The *makeMove* function in the trimmed programs.

One-dimensional For this program the code might look more complex, but it is really about the same as for the trimmed programs. the only real difference lies in that instead of using numpy's *argmax* function, it needs its own version, as there isnt a matrix Q, it is an array instead, without the holes indicating legal movement. Thus the code for figuring out the highest expected value for a move needs to include T, which *argmax* doesn't do.

```
def makeMove(self, state):
  curState = state[:]
  curState.sort(reverse = True)
  nextState = []
  move = []
  if (len(state)<len(self.listOfStates[0])):
    dif = len(self.listOfStates[0])-len(state)
    for i in range(dif):
      curState.append(0)
  goal = 0
  progress = self.board[0]
  for i in range(len(self.board)):
    heaps = len(self.board)- (i+1)
    for j in range(progress, -1, -1):
      if(j == curState[i]):
        break
      else:
        if(heaps == 0):
          goal += j-curState[i]
          break
        result = 1
        for x in range(1, heaps + 1):
          result *= (j + x)
        goal += (result)//math.factorial(heaps)
        progress -= 1
  highVal = -np.inf
  highValIndex = -1
  for j in range(goal+1, len(self.Q)):
    if(self.T[goal][j] == 1 and np.greater(self.Q[j], highVal)):
      highVal = self.Q[j]
      highValIndex = j
  nextState = self.listOfStates[highValIndex][:]
  numToDec = 0
  for i in range(len(curState)-1, -1, -1):
    if (curState[i] != nextState[i]):
      numToDec += curState[i]-nextState[i]
      newIndex = curState[i]
```

```
134   move.append(numToDec)
135   for i in range(len(state)):
136     if(newIndex == state[i]):
137       newIndex = i
138       break
139   move.append(newIndex)
140   return move
```

Listing 13: The *makeMove* function in the one-dimensional program.

6.2.5 *Run.py*

This is the program that actually runs the previous programs, and takes some inputs as to which of the programs it will run, and for which games of Nim. The program takes the following input:

1. Lower size of the Nim heaps you want to run.
2. Upper size of the Nim heaps.
3. Lower amount of Nim heaps.
4. Upper amount of Nim heaps.
5. Whether the program *ReinforcementSarsa.py* should be run. 0 for no, 1 for yes.
6. Whether the program *ReinforcementQ-learning.py* should be run.
7. Whether the program *SpaceSaveSarsa.py* should be run.
8. Whether the program *SpaceSaveQ-learning.py* should be run.
9. Whether the program *OneDSarsa.py* should be run.
10. Whether the program *SpaceSaveSarsaSpread.py* should be run.

Handling of input This is the code *Run.py* uses to handle inputs and a little other things.

```
1  import time
2  import sys
3  import ReinforcementSarsa as RS
4  import ReinforcementQlearning as RQ
5  import PerfectPlayer as PP
6  import SpaceSaveSarsa as SS
7  import SpaceSaveQlearning as SQ
8  import OneDSarsa as ODS
9  import SpaceSaveSarsaSpread as SSSS

11 if(len(sys.argv) < 11):
12   print("Error: too few arguments. Got " + str(len(sys.argv)-1) + "
       , Expected 10")
13 elif (len(sys.argv) > 11):
```

```
14    print("Error: too many arguments. Got, " + str(len(sys.argv)-1) +
          ", Expected 10")
15  else:
16    #Parsing the input.
17    heapSizeLower = int(sys.argv[1])
18    heapSizeUpper = int(sys.argv[2])
19    heapAmountLower = int(sys.argv[3])
20    heapAmountUpper = int(sys.argv[4])

22    player = PP.PerfectPlayer()

24    ssstatistics = open("ssstatistics.txt", "w")
25
26    #writing the header of the statistics file, to show which numbers
          correspond to which program.
27    header = ""
28    if(int(sys.argv[5]) == 1):
29      header += "\t\t\t\t\t\tRS\t"
30    if(int(sys.argv[6]) == 1):
31      header += "\t\t\t\t\t\tRQ\t"
32    if(int(sys.argv[7]) == 1):
33      header += "\t\t\t\t\t\tTS\t"
34    if(int(sys.argv[8]) == 1):
35      header += "\t\t\t\t\t\tTQ\t"
36    if(int(sys.argv[9]) == 1):
37      header += "\t\t\t\t\t\tODS\t"
38    if(int(sys.argv[10]) == 1):
39      header += "\t\t\t\t\t\tNew Program"
40    header += "\n"
41    ssstatistics.write(header)
```

Listing 14: The code in *Run.py* before the programs are actually run.

Running of the programs This code is repeated 6 times, once for each program.

- Repeat for each state specified in the input:
  - Repeat for each program:
    1. Check the inputs to see if this programs should be run.
    2. Repeat 10 times
       (a) Initialize and run the setup of the program(which also runs the training), taking the time it uses for this.
       (b) Play against the perfect player(which uses the deterministic algorithm), for each state the program can win(that is those with a nim-sum unequal to 0) if it starts, that are less than the state the program was run with.
    3. find the average time of all the programs, and write it into the statistics file.

```
42    #The loop that defines the start-states from which the programs
          will be run.
```

```
43  for heapSize in range(heapSizeLower, heapSizeUpper+1):
44    for heapAmount in range(heapAmountLower, heapAmountUpper+1):
45      state = [heapSize] * heapAmount
46      print(state)
47      line = ""
48      """This repeats for each program. For each start-state they
    are run 10 times, taking their time use, after which they are
    set to play against the perfect player
49      for all starts it can play from, from which it can win when
    starting. That is, the states with a nim-sum != 0. The averages
     of the times and winrate is then
50      written into the statistics file."""
51      if(int(sys.argv[5]) == 1):
52        timeAvg = 0
53        percent = 0
54        setupTime = 0
55        trainTime = 0
56        for i in range(10):
57          start = time.time()
58          rs = RS.ReinforcementSarsa(state)
59          times = rs.setup()
60          end = time.time()
61          timeAvg += end-start
62          setupTime += times[0]
63          trainTime += times[1]
64          listOfStates = rs.getListOfStates()
65          wins = 0
66          losses = 0
67          for i in range(len(listOfStates)):
68            if(player.findNimSum(listOfStates[i]) != 0):
69              playing = 1
70              board = listOfStates[i][:]
71              winBoard = [0] * len(listOfStates[i])
72              while(playing):
73                move = rs.makeMove(board)
74                board[move[1]] = board[move[1]]-move[0]
75                if(board == winBoard):
76                  wins += 1
77                  break
78                move = player.makeMove(board)
79                board[move[1]] = board[move[1]]-move[0]
80                if(board == winBoard):
81                  losses += 1
82                  break
83          percent += (wins/(wins+losses)*100)
84        timeAvg /= 10
85        percent /= 10
86        setupTime /= 10
87        trainTime /= 10
88        line += "setup: " + str(round(setupTime, 3)) + "s\t train:
    " + str(round(trainTime, 3)) + "s\t total: " + str(round(
    timeAvg, 3)) + "s\t" + str(int(percent)) + "%\t"
89      if(int(sys.argv[6]) == 1):
```

Listing 15: This is the code that runs the programs, takes the time they use and creates statistics about it.

6.2.6 *Play.py*

This program lets the user play a game of Nim against one of the reinforcement programs, or get the transcript of a full game against the perfect player written to file. Its inputs are like this:

1. Should the program play agains the perfect player or the user. 0 for the perfect player, any other number for the user.
2. Which reinforcement program should be run? 0 for *reinforcementSarsa.py*, 1 for *reinforcementQ-learning.py*, 2 for *spaceSaveSarsa.py*, 3 for *spaceSaveQ-learning.py*, 4 for *oneDSarsa.py*, and any other number for *SpaceSaveSarsaSpread.py*.
3. The state from which the game starts, in the form: $n_1, n_2, n_3, ..., n_k$.
4. Should the program make the first move, or should the user/perfect player, 1 for the program making the first move, any other number for the user/perfect player making the first move.

It works quite similar to how games between the programs and the perfect player is done in *Run.py*.

```
import time
import sys

import ReinforcementSarsa as RS
import ReinforcementQlearning as RQ
import PerfectPlayer as PP
import SpaceSaveSarsa as SS
import SpaceSaveQlearning as SQ
import OneDSarsa as ODS
import SpaceSaveSarsaSpread as SSSS

if(len(sys.argv) < 5):
  print("Error: too few arguments. Got " + str(len(sys.argv)-1) + "
    , Expected 4")
elif (len(sys.argv) > 5):
  print("Error: too many arguments. Got, " + str(len(sys.argv)-1) +
     ", Expected 4")
else:
  #Parsing the input
  startState = list(map(int, sys.argv[3].split(',')))
  trainingType = int(sys.argv[2])
  if(trainingType == 0):
    ml = RS.ReinforcementSarsa(startState)
  elif(trainingType == 1):
    ml = RQ.ReinforcementQlearning(startState)
  elif(trainingType == 2):
    ml = SS.SpaceSaveSarsa(startState)
  elif(trainingType == 3):
    ml = SQ.SpaceSaveQlearning(startState)
  elif(trainingType == 4):
    ml = ODS.OneDSarsa(startState)
  else:
    ml = SSSS.SpaceSaveSarsaSpread(startState)
```

```
ml.setup()

#The code for the reinforcement program playing against the deterministic program, and getting statistics from it.
if(int(sys.argv[1]) == 0):
  playstatistics = open("playstatistics.txt", "w")
  playstatistics.write(str(trainingType) + "\n")
  player = PP.PerfectPlayer()
  playing = 1
  board = startState[:]
  winBoard = [0] * len(startState)
  #If the reinforcement algo gets the first move it gets to make a move before the loop start.
  if(int(sys.argv[4]) == 1):
    start = time.time()
    move = player.makeMove(board)
    end = time.time()
    moveTime = end-start
    board[move[1]] = board[move[1]]-move[0]
    playstatistics.write("Move by Pre-algo, took " + str(move[0]) + " counters from heap " + str(move[1]+1) + ". New state: " + str(board) + " time used: " + str(round(moveTime, 3)) + "s.\n")
    if(board == winBoard):
      playstatistics.write("Pre-algo is the winner.")
      playing = 0
  #Otherwise the game starts here.
  while(playing):
    start = time.time()
    move = ml.makeMove(board)
    end = time.time()
    moveTime = end-start
    board[move[1]] = board[move[1]]-move[0]
    playstatistics.write("Move by ML, took " + str(move[0]) + " counters from heap " + str(move[1]+1) + ". New state: " + str(board) + " time used: " + str(round(moveTime, 3)) + "s.\n")
    if(board == winBoard):
      playstatistics.write("ML is the winner.")
      break
    start = time.time()
    move = player.makeMove(board)
    end = time.time()
    moveTime = end-start
    board[move[1]] = board[move[1]]-move[0]
    playstatistics.write("Move by Pre-algo, took " + str(move[0]) + " counters from heap " + str(move[1]+1) + ". New state: " + str(board) + " time used: " + str(round(moveTime, 3)) + "s.\n")
    if(board == winBoard):
      playstatistics.write("Pre-algo is the winner.")
      break
#If the player wants to play against the Reinforcement algo.
else:
  player = PP.PerfectPlayer()
  playing = 1
  board = startState[:]
  winBoard = [0] * len(startState)
  #If the player wants to start first, they make a move first.
  if(int(sys.argv[4]) == 1):
```

```
      move = [-1, -1]
      print("Current board is: " + str(board) + "\n")
      move[1] = int(input("Which heap do you want to remove from? "
    ))-1
      while((move[1] < 0 or move[1] >= len(board)) or board[move
    [1]] == 0):
        move[1] = int(input("Illegal heap, choose again. "))-1
      move[0] = int(input("How many counters do you wish to remove?
     "))
      while(move[0] < 1 or move[0] > board[move[1]]):
        move[0] = int(input("Illegal amount, choose again. "))
      board[move[1]] = board[move[1]]-move[0]
      if(board == winBoard):
        print("You are the winner.")
        playing = 0
    #Otherwise the game starts here, with the reinforcement program
     making the first move.
    while(playing):
      move = ml.makeMove(board)
      board[move[1]] = board[move[1]]-move[0]
      print("Move by ML, took " + str(move[0]) + " counters from
    heap " + str(move[1]+1) + ".")
      if(board == winBoard):
        print("ML is the winner.")
        break
      move = [-1, -1]
      print("Current board is: " + str(board) + "\n")
      move[1] = int(input("Which heap do you want to remove from? "
    ))-1
      while(move[1] < 0 or move[1] >= len(board) or board[move[1]]
    == 0):
        move[1] = int(input("Illegal heap, choose again. "))-1
      move[0] = int(input("How many counters do you wish to remove?
     "))
      while(move[0] < 1 or move[0] > board[move[1]]):
        move[0] = int(input("Illegal amount, choose again. "))
      board[move[1]] = board[move[1]]-move[0]
      if(board == winBoard):
        print("You are the winner.")
        break
```

Listing 16: The code in program *Play.py*.

6.3 Programs for Supervised Learning

There are 3 programs used for the supervised learning, as well as the program *PerfectPlayer.py* which is used to ascertain success.

6.3.1 *MakeData.py*

This program makes the data used by the algorithm, for training. It is quite simple, it uses the perfect player, and for all states reachable from the one it is initialized with it writes a list of files and their correct moves to a file.

```
import PerfectPlayer as PP
```

```
class MakeData:
  #This method just makes the list of states accessible from the
    start-state, and then runs the writeToFile method.
  def setup(self, startBoard):
    sum = startBoard[0]+1
    board = startBoard
    for i in range(1, len(board)):
      sum *= board[i]+1
    self.listOfStates = [0] * sum
    self.listOfStates[0] = board[:]
    for i in range(sum):
      for x in range(len(self.listOfStates[i])):
        for y in range(1, self.listOfStates[i][x] + 1):
          tmpValue = self.newIndex(x, y, board)
          if(self.listOfStates[i+tmpValue] == 0):
            tmpList = self.listOfStates[i][:]
            tmpList[x] -= y
            self.listOfStates[i+tmpValue] = tmpList[:]
    self.writeToFile(self.listOfStates)

  # This method exists for finding, given a move, how far down the
    list of states the new state is.
  def newIndex(self, heapIndex, amount, board):
    newMove = 1
    for i in range(len(board)-1, heapIndex, -1):
      newMove *= board[i]+1
    newMove *= amount
    return newMove

  #This file writes each state in the list to the file, and a
    correct move to make when in that state.
  def writeToFile(self, states):
    player = PP.PerfectPlayer()
    f = open("CorrectMoves.txt", "w")
    for i in range(len(states)):
      if (player.findNimSum(states[i]) != 0):
        tmp = [0]*len(states[0])
        result = player.makeMove(states[i])
        tmp[result[1]] = result[0]
        for j in range(len(states[i])):
          f.write(str(states[i][j]) + " ")
        f.write("\n")
        for j in range(len(states[i])):
          f.write(str(tmp[j]) + " ")
        f.write("\n")

  #This method returns the list of states, which is used to test
    programs, by having them play from many different states.
  def getStates(self):
    return self.listOfStates
```

Listing 17: The program *MakeData.py* which creates the data used by the supervised learning algorithm.

6.3.2 The Multilayer Perceptron

This is the code for the MultiLayer Perceptron, in the program *mlp.py*. This code is based upon the code I made for an obligatory task in fall 2018 for the course INF4490. It has been modified to apply better to this subject, yet much of the code is the same, as both are code for Multilayer Perceptrons.

Training The training works like described earlier, except: the program only uses the sigmoid function for the hidden layer, for the output layer it uses the identity function, that is $f(x) = x$. To give an idea of the order functions call things:

1. The program is initialized.
2. The function *earlystopping* is called from the program *RunMlp.py*.
3. *earlystopping* runs the function *train* repeatedly, calling *totalError* for each time to check how well the program has been trained. If the total error stops decreasing for each repeat, the training stops.
4. *forward* is called by *train*, once for each iteration.

```
import numpy as np
import random
from scipy.special import expit

class mlp:
  def __init__(self, nhidden, stateLen):
    self.beta = 1
    self.eta = 0.1
    self.momentum = 0.0
    self.matrix1 = []
    self.matrix2 = []
    self.inputAmount = stateLen
    self.outputAmount = stateLen
    self.hiddenAmount = nhidden
    self.outputA = [0.0] * (self.outputAmount)
    self.outputList = [0.0] * (self.outputAmount)
    self.hiddenU = [0.0] * (nhidden)
    self.hiddenList = [0.0] * (nhidden)
    for i in range(self.inputAmount + 1):
      self.matrix1.append([])
      for j in range(nhidden):
        self.matrix1[i].append(random.random())
    for i in range(nhidden + 1):
      self.matrix2.append([])
      for j in range(self.outputAmount):
        self.matrix2[i].append(random.random())

  def sigmoidDerived(self, x):
    return expit(x)*(1-expit(x))

  def identityFunction(self, x):
    return x
```

```
33
34  def identityDerived(self, x):
35    return 1
36
37  #this function calculates the sum of squares error of all the
      valid inputs and targets, and sums them, and returns that for
      the earlystopping to use for figuring out when to stop.
38  def totalError(self, inputs, targets):
39    accumError = 0.0
40    for i in range(len(inputs)):
41      accum = 0.0
42      self.forward(inputs[i])
43      for j in range(self.outputAmount):
44        accum += (targets[i][j] - self.outputList[j])**2
45      accumError += accum/2
46    return accumError

48  #earlystopping, stops when error increases, or decreases too
      slowly
49  def earlystopping(self, inputs, targets, valid, validtargets):
50    stopTraining = 0
51    lowestError = self.totalError(valid, validtargets)
52    while not stopTraining:
53      self.train(inputs, targets, 10)
54      newError = self.totalError(valid, validtargets)
55      if newError > lowestError or lowestError-newError < 0.5:
56        stopTraining = 1
57      else:
58        lowestError = newError

60  #train trains the neural net for the inputted amount of
      iterations. One pass through the dataset is an iteration.
61  def train(self, inputs, targets, iterations=100):
62    for notUsed in range(iterations):
63      for ite in range(len(inputs)):
64        self.forward(inputs[ite])
65        outDelta = []
66        for i in range(self.outputAmount):
67          outDelta.append((self.outputList[i] - targets[ite][i]) *
    self.identityDerived(self.outputA[i]))
68        hiddenDelta = []
69        for i in range(self.hiddenAmount):
70          accumulator = 0.0
71          for j in range(self.outputAmount):
72            accumulator += outDelta[j] * self.matrix2[i+1][j]
73          hiddenDelta.append(self.sigmoidDerived(self.hiddenU[i]) *
     accumulator)
74        for j in range(self.outputAmount):
75          self.matrix2[0][j] -= self.eta * outDelta[j] * 1
76          for i in range(self.hiddenAmount):
77            self.matrix2[i+1][j] -= self.eta * outDelta[j] * self.
    hiddenList[i]
78        for j in range(self.hiddenAmount):
79          self.matrix1[0][j] -= self.eta * hiddenDelta[j] * 1
80          for i in range(self.inputAmount):
81            self.matrix1[i+1][j] -= self.eta * hiddenDelta[j] *
    inputs[ite][i]
```

```
83    #forward passes the inputs through the neural net to get the
        outputs.
84    def forward(self, inputs):
85      for i in range(len(self.hiddenList)):
86        self.hiddenU[i] = 1*self.matrix1[0][i]
87        for j in range(self.inputAmount):
88          self.hiddenU[i] += inputs[j] * self.matrix1[j+1][i]
89        self.hiddenList[i] = expit(self.hiddenU[i])
90      for i in range(len(self.outputList)):
91        self.outputA[i] = 1*self.matrix2[0][i]
92        for j in range(len(self.hiddenList)):
93          self.outputA[i] += self.hiddenList[j]*self.matrix2[j+1][i]
94        self.outputList[i] = self.identityFunction(self.outputA[i])
```

Listing 18: This is the part of the program *mlp.py* in which the training happens.

Making Moves The program makes moves by running the state through the neural net, and choosing the largest output. It also makes sure the move is legal, so it doesnt add counters, or remove more counters than there is in a heap. The consequences of this is that it might often make small moves when it really hasn't been trained well enough.

```
99  def makeMove(self, state):
100   self.forward(state)
101   if(state[0] < -1):
102     1/0
103   move = [0,0]
104   move[1] = np.argmax(self.outputList)
105   move[0] = int(self.outputList[move[1]])
106   if(move[0] < 1):
107     move[0] = 1
108   elif(move[0] > state[move[1]]):
109     move[0] = state[move[1]]
110   if(state[move[1]] == 0):
111     for i in range(len(state)):
112       if(state[i] > 0):
113         move[0] = 1
114         move[1] = i
115   return move
```

Listing 19: The part of *mlp.py* that is used to get a move from the program.

6.3.3 *RunMlp.py*

This program is similar to *Run.py* except it runs *mlp.py* instead of any of the Reinforcement programs. Its inputs are fewer, as there is only 1 program to run, and they are:

1. Lower size of the Nim heaps you want to run.
2. Upper size of the Nim heaps.
3. Lower amount of Nim heaps.

4. Upper amount of Nim heaps.

```
import MakeData as MD
import PerfectPlayer as PP
import numpy as np
import mlp
import sys
import time

if(len(sys.argv) < 5):
  print("Error: too few arguments. Got " + str(len(sys.argv)-1) + "
    , Expected 4")
elif (len(sys.argv) > 5):
  print("Error: too many arguments. Got, " + str(len(sys.argv)-1) +
     ", Expected 4")
else:
  #Parsing the input.
  heapSizeLower = int(sys.argv[1])
  heapSizeUpper = int(sys.argv[2])
  heapAmountLower = int(sys.argv[3])
  heapAmountUpper = int(sys.argv[4])
  f = open("mlpstatistics.txt", "w")
  filename = "CorrectMoves.txt"
  player = PP.PerfectPlayer()
  #The loop that defines the start-states from which the programs
    will be run.
  for heapSize in range(heapSizeLower, heapSizeUpper+1):
    for heapAmount in range(heapAmountLower, heapAmountUpper+1):
      """The Supervised learning program is run 10 times for each
    start-state, and plays against the perfect player each time,
      to get a winrate for statistics. The time used is also
    written to file as statistics. The data used is the same for a
    given state,
      but it is sorted differently for each time."""
      state = [heapSize] * heapAmount
      print(state)
      md = MD.MakeData()
      md.setup(state)
      timeAvg = 0
      percent = 0
      for i in range(10):
        #sorting and separating the data into the needed categories
    .
        data = np.loadtxt(filename)

        stateLen = len(data[0])

        boards = data[::2]
        moves = data[1::2]

        order = list(range(np.shape(boards)[0]))
        np.random.shuffle(order)
        boards = boards[order,:]
        moves = moves[order,:]

        train = boards[::2]
        trainTargets = moves[::2]
```

```
        valid = boards[1::4]
        validTargets = moves[1::4]

        test = boards[3::4]
        testTargets = moves[3::4]

        hidden = len(state)+10

        start = time.time()
        net = mlp.mlp(hidden, stateLen)
        net.earlystopping(train, trainTargets, valid, validTargets)
        end = time.time()
        timeAvg += end-start
        listOfStates = md.getStates()
        wins = 0
        losses = 0
        for i in range(len(listOfStates)):
          if(player.findNimSum(listOfStates[i]) != 0):
            playing = 1
            board = listOfStates[i][:]
            winBoard = [0] * len(listOfStates[i])
            while(playing):
              move = net.makeMove(board)
              board[move[1]] = board[move[1]]-move[0]
              if(board == winBoard):
                wins += 1
                break
              move = player.makeMove(board)
              board[move[1]] = board[move[1]]-move[0]
              if(board == winBoard):
                losses += 1
                break
        percent += (wins/(wins+losses)*100)
      timeAvg /= 10
      percent /= 10
      line = "Time used:" + str(round(timeAvg, 3)) + "s Winrate/
    sucessrate: " + str(int(percent)) + "\\% "
      line += str(state) + "\n"
      f.write(line)
```

Listing 20: The code for the program *RunMlp.py*

It shuffles the training data for each run so that it is different each time. Only the total time used is taken here, unlike for the reinforcement programs, as setup was a much larger concern in those.

6.3.4 Program With Stochastic Termination

This is the program that runs the programs with stochastic termination, and records the time used, as well as the amount of episodes it ran. The program takes the following parameters:

1. Lower size of the Nim heaps you want to run.

2. Upper size of the Nim heaps.
3. Lower amount of Nim heaps.
4. Upper amount of Nim heaps.
5. The reinforcement program it should run:
    - 0 for *ReinforcementSarsa.py*.
    - 1 for *ReinforcementQ-learning.py*.
    - 2 for *SpaceSaveSarsa.py*.
    - 3 for *SpaceSaveQ-learning.py*.
    - 4 for *OneDSarsa.py*.
    - Any other number for *SpaceSaveSarsaSpread*.

The program reuses much code from *Run.py*, so they are quite similar. The statistics also show the program being run, with the number input for it, as well as the desired percentage.

```
import time
import sys
import ReinforcementSarsa as RS
import ReinforcementQlearning as RQ
import PerfectPlayer as PP
import SpaceSaveSarsa as SS
import SpaceSaveQlearning as SQ
import OneDSarsa as ODS
import SpaceSaveSarsaSpread as SSS

if(len(sys.argv) < 6):
  print("Error: too few arguments. Got " + str(len(sys.argv)-1) + "
    , Expected 5")
elif (len(sys.argv) > 6):
  print("Error: too many arguments. Got, " + str(len(sys.argv)-1) +
     ", Expected 5")
else:
  #Parsing the input.
  heapSizeLower = int(sys.argv[1])
  heapSizeUpper = int(sys.argv[2])
  heapAmountLower = int(sys.argv[3])
  heapAmountUpper = int(sys.argv[4])
  trainingType = int(sys.argv[5])
  player = PP.PerfectPlayer()
  ssstatistics = open("episodestatistics.txt", "w")
  #The desired percentage winrate, which terminates the program.
  goal = 90
  ssstatistics.write("Type: " + str(trainingType) + " ,percentage
    goal: " + str(goal) + "%\n")
  for heapSize in range(heapSizeLower, heapSizeUpper+1):
    for heapAmount in range(heapAmountLower, heapAmountUpper+1):
      state = [heapSize] * heapAmount
      print(state)
      #Settings for the programs to train.
```

```
32      amountEpisodes = 10
33      epsilon = 0.1
34      mu = 0.5
35      gamma = 0.5
36      totalTime = 0
37      start = time.time
38      #The selection of the correct program to run.
39      if(trainingType == 0):
40        ml = RS.ReinforcementSarsa(state)
41      elif(trainingType == 1):
42        ml = RQ.ReinforcementQlearning(state)
43      elif(trainingType == 2):
44        ml = SS.SpaceSaveSarsa(state)
45      elif(trainingType == 3):
46        ml = SQ.SpaceSaveQlearning(state)
47      elif(trainingType == 4):
48        ml = ODS.OneDSarsa(state)
49      else:
50        ml = SSS.SpaceSaveSarsaSpread(state)
51      ml.setup(1)
52      end = time.time()
53      listOfStates = ml.getListOfStates()
54      totalTime += end - start
55      stillTraining = 1
56      totalEpisodes = 0
57      #This loop runs until the percentage winrate is equal or
    higher than the desired goal.
58      while(stillTraining):
59        start = time.time()
60        ml.train(amountEpisodes, epsilon, mu, gamma)
61        end = time.time()
62        totalTime += end-start
63        wins = 0
64        losses = 0
65        """Unlike in the program Run.py the statistics aren't
    repeated ten times. Quite understandably, this would not work
    here,
66        because the desire is one properly trained program, not ten
     alright ones."""
67        for i in range(len(listOfStates)):
68          if(player.findNimSum(listOfStates[i]) != 0):
69            playing = 1
70            board = listOfStates[i][:]
71            winBoard = [0] * len(listOfStates[i])
72            while(playing):
73              move = ml.makeMove(board)
74              board[move[1]] = board[move[1]]-move[0]
75              if(board == winBoard):
76                wins += 1
77                break
78              move = player.makeMove(board)
79              board[move[1]] = board[move[1]]-move[0]
80              if(board == winBoard):
81                losses += 1
82                break
83        totalEpisodes += amountEpisodes
84        percent = (wins//(wins+losses)*100)
```

```
85          if(percent >= goal):
86            stillTraining = 0
87        ssstatistics.write("Time used: " + str(round(totalTime, 3)) +
      "s, Episodes used: " + str(totalEpisodes) + ", " + str(state)
    + "\n")
```

Listing 21: The code for the program *Episodes.py*, that tries to run the reinforcement programs with stochastic termination

7 Comparing the Algorithms with Time Complexity

Usually comparing worst time-complexities is used to compare how good an algorithm is. This does not work properly with stochastic algorithms, because some of them do not have a predictable point of termination, where a deterministic algorithm does. This does not always apply, however. An algorithm that simulates a single die, say one with 6 sides, should always terminate at the same time, but it is still stochastic as the outcome will differ each time it is run. In the case of the algorithm that uses reinforcement learning, it will always run the same amount of episodes, so how long time will use has a clear upper limit but for the same output, it does not always use the same amount of time. What is important is whether an algorithm can go on forever, because if it can, there are never a worst case time use, the worst case would be infinite.

7.1 The Deterministic Algorithm

7.1.1 Time-complexity of the Nim-sum Operation

First one needs to know the time complexity of a Nim-sum operation. In many programming languages the Xor bitwise-operation will do the same, as previously mentioned. That operation uses constant time, that is $O(1)$, but that is because it is not constrained the actual size of the number. If one were to use a Turing machine instead, it would have a complexity of $O(n)$ where k is the amount of digits in the larger of the input numbers. This is simply because there has to be some comparison between the 2 numbers for each digit, with the numbers in binary, of course.

1. Repeat until one of the numbers run out of digits, starting on the smallest digit.
   (a) Compare the current digits of the two numbers.
   (b) If they are equal set the digit in the sum to be 0. If not set the digit in the sum to be 1.
   (c) Move on to the next digit.
2. Now that one of the numbers has run out of digits, repeat the following until the largest number is out of digits.

(a) Set the digit of the sum to be the current digit of the larger number.

(b) Move on to the next digit.

This list shows a suggested algorithm to find the nim-sum, or the exclusive or comparison of 2 numbers. This repeats the three steps as many times as the lesser number has digits, and has to set the rest of the larger number's digits too, so it has the time complexity of $O(n)$ where n is the amount of digits in the larger number.

7.1.2 The Rest of the Algorithm

So for the algorithm it is required to find the nim-sum of the size of sizes of all the heaps. Since the operation only accepts 2 inputs, it will have to be repeated for $k-1$ times, where k is the amount of heaps. Furthermore the algorithm uses the operation again to find the size of the heap which should be reached. not all heaps have a viable move, so if it is unlucky, it will have to find the nim-sum for all heaps. Thus the operation will, in the worst case, be used $(k-1)+k$ times. The best case would be $(k-1)+1$ if the first heap can be reduced to get to a winning position. So then, if k is the amount of heaps, and n is the size of the biggest number, the time complexity in total is $O(k*n)$. To reiterate, this is how the algorithm works, abstractly.

1. Repeat the following until the sum includes the size of the last heap, with the current sum being defined as the size of the first heap at the first iteration.

   - Find the nim-sum of the current sum and the size of the next heap. The result is the new current sum.

2. Starting with the first heap, do the following.

   (a) Find the nim-sum of the total sum and the size of a heap.

   (b) If it is less than the size of the heap, a correct move is found, which is to remove counters from the heap so that it reaches the size equal to the sum found. If the sum is larger, then repeat with the next heap.

7.1.3 Practically, Then?

In practice, the xor operation actually has a complexity of $O(1)$ on a computer, like the sum operation $+$, in which case the full algorithm is $O(k)$ as opposed to $O(k*n)$. This is a clear improvement, however it does not make much of a difference. The other algorithms use the sum operation many times, and that gets the same improvement.

7.2 The Reinforcement Learning Algorithm

This algorithm has several parts, and these will be considered separately. There are three parts. First the setup of the matrices, for legal moves and expected rewards. Second is the training itself, where the algorithm plays against itself an amount of times, to learn which moves are the best moves. Third there is the function that tells the user which move it gets the highest reward from, when given a certain game-state.

7.2.1 Setup of the Matrices

The time this takes is governed by the size of the matrix, which is the amount of possible states reachable in a game from the start-state. Let us call this m. It is clear that m increases at least exponentially when compared to the size of the input, because whenever a counter is added, m increases by at least 1, while the size of the input does not have to increase at all, since a digit might just be replaced by a bigger digit.

Size of game trees The size of a side of the matrices are all states reachable from the start states, which is just a game tree. The equation for finding the size of a certain games gametree, if all the heaps are of the same size is simply the following, or trimmed, where n is the size of the heaps, and k is the amount of heaps:

$$\frac{(n+k)!}{n! * k!} \tag{18}$$

To figure out the time complexity of this part of the algorithm, it is needed to confirm that the size of the game tree does not increase exponentially in relation to the amount of counters in the game. If it does the size of the matrices will be something like 2^{2^N} where N is the size of the input, and that won't do at all. The growth of the tree is as follows when adding new heaps, where x is the amount of heaps, and n is the arbitrary amount in each heap:

$$\frac{1}{n!} * (x+1) * (x+2) * ... * (x+n) \tag{19}$$

The growth of the tree when adding a counter to each pile is like this:

$$\frac{1}{n!} * (x+1) * (x+2) * ... * (x+n) \tag{20}$$

Which is the same, because the equation for finding the size of the game tree is symmetrical, after all. Do note however that this shows the growth when adding either a heap with the same numbers of counters as the other heaps, or adding 1 counter to all heaps, which is usually more than 1 counter, so adding 1 counter will usually cause a growth much less than these show. Nevertheless

this growth is not exponential, and so the size of the matrices compared to the size of the input is $O(2^N)$.

7.2.2 Training the Matrix

The training does not use the same amount of time, each time, however gauging what the worst case time use is, isnt very hard. Every episode happens a finite amount of times, and in each episode the algorithm completes a full game against itself. The longest possible game of Nim from any state is where both players remove one counter each, so the answer is the sum of the size of the heaps. the sum is of course exponentially bigger than the size of the input, since it is basically a comparison between a number and its size. So this part also has a time complexity of $O(2^N)$. It should be mentioned that the training will fail to find relevant data in most cases if it does this, however this is the closest comparison to be made.

7.2.3 Giving an Answer

Finding the move that gives the highest reward, will again be quite slow, compared to the size of the input, because when you find correct index, there needs to be a search through all the expected rewards to find the highest. That might be nearly all possible positions, which means that it will have the time complexity $O(2^N)$ and there is no way around it. Since the sides of the matrices increase exponentially in comparison to the size of the input, we are out of luck. It does not really matter how the correct index for the input state is found, as the worst solution, just searching, will only at most double the time used.

8 Comparing the Algorithms with Actual Time-use

Perhaps their actual average time use is a better measure, so some statistics is in order. For all these values the program was run 10 times, and the average of those values is what is shown, to be sure no one value is strange because of randomness. Any strangeness should be a feature of the algorithm, not just of plain luck[24].

[24] Perhaps lack of luck, being unlucky is more accurate here, as nice and accurate values in statistics would probably be better, though less interesting.

8.1 Reinforcement Learning

Here are the statistics about the programs using reinforcement-learning algorithms.

8.1.1 The naming of the programs in the graph

For smaller tables and readability the programs will be given shortform names, which are found in Table 6

Table 6: The shortform names for the different programs used in the statistics.

Name	Program
RS	Program using the untrimmed game-tree, and uses the Sarsa algorithm.
RQ	Program using the untrimmed game-tree, and uses the Q-learning algorithm.
TS	Program using the trimmed game-tree, and uses the Sarsa algorithm
TQ	Program using the trimmed game-tree, and uses the Q-learning algorithm
ODS	Program using the trimmed game-tree, uses the Sarsa algorithm, and uses the state-value function $V(s)$

8.1.2 The Settings Used by the Programs Running, and Other Relevant Information

When training, the training variables μ and γ are both set to 0.5, and ϵ is set to 0.1, except for ODS, where it is set to 0. The program runs the training through 1000 episodes. The way the winrate is found(The measure of success) is by having the program play against the perfect player(a program using the deterministic algorithm for playing Nim) from all states it knows, and it can win from. That means that for all the states in the programs list of states where the states nim-sum is not 0, it will play a game against the perfect player. The wins and losses are registered, so a percentage can be found. The program is trained, tested, and timed 10 times, and the average of the results are found.

Possible Weakness with the Measure of Success Measuring winrate this way may have a disadvantage. To win a game against the perfect player, every move needs to be a move to a state of the Nim-sum 0. If only one move is wrong, the game is lost. Since the testing is done from many states, many different games are played, but many of the same states are reached. The point is that to win any game it reaches, it must do the winning move, emptying the last heap, while many moves in larger states are just required for a small subset of games. Thus the ability to make the last move is much more important and perhaps over-valued using the current method of measuring success. On the other hand however, the move of removing from the last heap is quicker to learn, since the best move is already defined, and for any move, the closer

it is to a win, the earlier in the training it will be trained, and given a more appropriate expected reward. So the bias for those final moves is justified given their advantage in training. Indeed, that advantage might make a method where you only check if the correct moves are made from winning positions be biased the towards the final moves as well.

8.1.3 General Statistics

Table 7 contains the run-times and winrates of all the pre-mentioned programs for the sake of comparison. The supremacy of the trimmed programs should be immidiatly obvious, as well as ODS doing better at harder states, but actually worse at some easier states, it is perhaps too generalized. The difference between Q-learning and Sarsa seems to be quite small, as they seem to do about as well, with some fluctuations. The biggest difference seem to be that Q-learning has a somewhat higher run-time, likely because for each move it needs to find the best move twice(once for the current action, and once for *argmax* in the assignment), while Sarsa only does so once.

Table 7: The general statistics of the different algorithms.

RS		RQ		TS		TQ		ODS		
Train	Wins	Train	Wins	Train	Wins	Train	Wins	Train	Wins	State
0.014s	100%	0.014s	100%	0.013s	100%	0.013s	100%	0.006s	100%	1 1
0.022s	100%	0.039s	100%	0.021s	100%	0.026s	100%	0.010s	100%	1 1 1
0.030s	100%	0.045s	100%	0.028s	100%	0.039s	100%	0.013s	100%	1 1 1 1
0.041s	100%	0.063s	100%	0.036s	100%	0.055s	100%	0.017s	100%	1 1 1 1 1
0.016s	100%	0.017s	100%	0.015s	100%	0.015s	100%	0.010s	100%	2 2
0.026s	97%	0.036s	98%	0.023s	98%	0.029s	97%	0.016s	100%	2 2 2
0.057s	99%	0.091s	99%	0.036s	100%	0.052s	100%	0.026s	97%	2 2 2 2
0.130s	72%	0.201s	67%	0.046s	92%	0.070s	89%	0.036s	100%	2 2 2 2 2
0.019s	100%	0.022s	100%	0.017s	100%	0.017s	100%	0.014s	100%	3 3
0.036s	87%	0.050s	85%	0.027s	90%	0.034s	93%	0.025s	100%	3 3 3
0.149s	96%	0.251s	96%	0.050s	100%	0.075s	100%	0.051s	89%	3 3 3 3
0.820s	40%	1.099s	42%	0.065s	84%	0.100s	84%	0.075s	90%	3 3 3 3 3
0.023s	100%	0.027s	100%	0.019s	100%	0.020s	100%	0.018s	100%	4 4
0.056s	67%	0.072s	57%	0.032s	84%	0.040s	66%	0.035s	98%	4 4 4
0.397s	43%	0.560s	44%	0.076s	99%	0.108s	99%	0.089s	96%	4 4 4 4
5.888s	14%	6.330s	14%	0.113s	74%	0.165s	79%	0.147s	68%	4 4 4 4 4
0.042s	100%	0.048s	100%	0.024s	100%	0.023s	100%	0.024s	94%	5 5
0.105s	62%	0.121s	46%	0.039s	66%	0.048s	63%	0.050s	94%	5 5 5
1.197s	23%	1.499s	24%	0.118s	99%	0.171s	99%	0.155s	84%	5 5 5 5
32.590s	6%	33.137s	6%	0.225s	65%	0.317s	71%	0.301s	84%	5 5 5 5 5

8.1.4 The Untrimmed Sarsa

Here are statistics on how the program running the sarsa algorithm with the untrimmed gametree. Something interesting is how the time used by the setup increases faster than that of the training.

Table 8: The statistics for applying sarsa to the untrimmed game tree.

Setup	**Training**	**Total**	**Winrate**	**Starting State**
0.000s	0.014s	0.014s	100%	1 1
0.000s	0.022s	0.022s	100%	1 1 1
0.000s	0.030s	0.030s	100%	1 1 1 1
0.001s	0.040s	0.041s	100%	1 1 1 1 1
0.000s	0.015s	0.016s	100%	2 2
0.001s	0.026s	0.026s	97%	2 2 2
0.003s	0.054s	0.057s	99%	2 2 2 2
0.026s	0.104s	0.130s	72%	2 2 2 2 2
0.000s	0.019s	0.019s	100%	3 3
0.002s	0.034s	0.036s	87%	3 3 3
0.028s	0.120s	0.149s	96%	3 3 3 3
0.455s	0.351s	0.820s	40 %	3 3 3 3 3
0.000s	0.021s	0.023s	100%	4 4
0.007s	0.049s	0.056s	67%	4 4 4
0.167s	0.226s	0.397s	43%	4 4 4 4
4.529s	1.200s	5.888s	14%	4 4 4 4 4
0.001s	0.025s	0.042s	100%	5 5
0.021s	0.084s	0.105s	62%	5 5 5
0.738s	0.438s	1.197s	23%	5 5 5 5
27.906s	3.008s	32.590s	6%	5 5 5 5 5

8.1.5 The Untrimmed Q-learning

Here are some statistics on how the program running the sarsa algorithm with the untrimmed gametree. The training time-use seems significantly higher than that for untrimmed Sarsa, while the setup time is about the same, supporting the theory of why Q-learning is slower.

Table 9: The statistics for applying Q-learning to the untrimmed game tree.

Setup	**Training**	**Total**	**Winrate**	**Starting State**
0.000s	0.014s	0.014s	100%	1 1
0.000s	0.039s	0.039s	100%	1 1 1
0.000s	0.045s	0.045s	100%	1 1 1 1
0.001s	0.062s	0.063s	100%	1 1 1 1 1
0.000s	0.017s	0.017s	100%	2 2
0.001s	0.036s	0.036s	98%	2 2 2
0.004s	0.087s	0.091s	99%	2 2 2 2
0.028s	0.173s	0.201s	67%	2 2 2 2 2
0.000s	0.021s	0.022s	100%	3 3
0.002s	0.048s	0.050s	85%	3 3 3
0.031s	0.218s	0.251s	96%	3 3 3 3
0.473s	0.611s	1.099s	42%	3 3 3 3 3
0.000s	0.025s	0.027s	100%	4 4
0.007s	0.065s	0.072s	57%	4 4 4
0.175s	0.378s	0.560s	44%	4 4 4 4
4.384s	1.805s	6.330s	14%	4 4 4 4 4
0.001s	0.032s	0.048s	100%	5 5
0.022s	0.098s	0.121s	46%	5 5 5
0.763s	0.714s	1.499s	24%	5 5 5 5
26.996s	4.696s	33.137s	6%	5 5 5 5 5

8.1.6 The Trimmed Sarsa

These are some statistics for the program using the trimmed gametree and that applies the sarsa algorithm on it. This program is clearly more useful than the untrimmed programs, being able to play larger heaps both better and faster. The time use still increases faster for setup than training though, interestingly enough. This is not strange however, since the code is quite similar, and the main difference lies in the graph used.

Table 10: The statistics for applying sarsa to the trimmed game tree.

Setup	**Training**	**Total**	**Winrate**	**Starting State**
0.000s	0.023s	0.023s	98%	2 2 2
0.000s	0.036s	0.036s	100%	2 2 2 2
0.000s	0.046s	0.046s	92%	2 2 2 2 2
0.001s	0.062s	0.063s	100%	2 2 2 2 2 2
0.001s	0.026s	0.027s	90%	3 3 3
0.001s	0.049s	0.05s	100%	3 3 3 3
0.002s	0.063s	0.065s	84%	3 3 3 3 3
0.005s	0.100s	0.106s	99%	3 3 3 3 3 3
0.001s	0.031s	0.032s	84%	4 4 4
0.004s	0.072s	0.076s	99%	4 4 4 4
0.011s	0.102s	0.113s	74%	4 4 4 4 4
0.029s	0.198s	0.228s	86%	4 4 4 4 4 4
0.003s	0.037s	0.039s	66%	5 5 5
0.011s	0.107s	0.118s	99%	5 5 5 5
0.037s	0.187s	0.225s	65%	5 5 5 5 5
0.124s	0.344s	0.472s	43%	5 5 5 5 5 5
0.006s	0.047s	0.052s	70%	6 6 6
0.027s	0.148s	0.175s	76%	6 6 6 6
0.126s	0.302s	0.433s	38%	6 6 6 6 6
0.458s	0.581s	1.051s	22%	6 6 6 6 6 6
0.009s	0.065s	0.074s	72%	7 7 7
0.067s	0.206s	0.275s	49%	7 7 7 7
0.361s	0.484s	0.856s	23%	7 7 7 7 7
1.518s	0.941s	2.499s	12%	7 7 7 7 7 7
0.016s	0.080s	0.096s	56%	8 8 8
0.128s	0.236s	0.367s	31%	8 8 8 8
0.862s	0.608s	1.492s	13%	8 8 8 8 8
4.284s	1.655s	6.056s	6%	8 8 8 8 8 8

8.1.7 The Trimmed Q-learning

These are some statistics for the program using the trimmed gametree and that applies the Q-learning algorithm on it. Trimmed Q-learning is also somewhat slower in its training, just like untrimmed Q-learning. They share the same code for training, so this was expected.

Table 11: The statistics for applying Q-learning to the trimmed game tree.

Setup	**Training**	**Total**	**Winrate**	**Starting State**
0.000s	0.029s	0.029s	97%	2 2 2
0.000s	0.051s	0.052s	100%	2 2 2 2
0.001s	0.069s	0.070s	89%	2 2 2 2 2
0.001s	0.097s	0.098s	100%	2 2 2 2 2 2
0.001s	0.034s	0.034s	93%	3 3 3
0.001s	0.074s	0.075s	100%	3 3 3 3
0.003s	0.097s	0.100s	84%	3 3 3 3 3
0.006s	0.165s	0.170s	99%	3 3 3 3 3 3
0.001s	0.038s	0.040s	66%	4 4 4
0.004s	0.105s	0.108s	99%	4 4 4 4
0.010s	0.155s	0.165s	79%	4 4 4 4 4
0.027s	0.309s	0.336s	99%	4 4 4 4 4 4
0.002s	0.045s	0.048s	63%	5 5 5
0.010s	0.161s	0.171s	99%	5 5 5 5
0.037s	0.279s	0.317s	71%	5 5 5 5 5
0.117s	0.527s	0.648s	52%	5 5 5 5 5 5
0.005s	0.056s	0.060s	53%	6 6 6
0.027s	0.234s	0.261s	88%	6 6 6 6
0.129s	0.494s	0.627s	47%	6 6 6 6 6
0.477s	0.971s	1.461s	24%	6 6 6 6 6 6
0.009s	0.071s	0.080s	51%	7 7 7
0.061s	0.309s	0.372s	56%	7 7 7 7
0.336s	0.698s	1.043s	26%	7 7 7 7 7
1.527s	1.632s	3.200s	13%	7 7 7 7 7 7
0.017s	0.089s	0.106s	42%	8 8 8
0.139s	0.416s	0.558s	37%	8 8 8 8
0.868s	1.076s	1.967s	15%	8 8 8 8 8
4.503s	2.843s	7.477s	7%	8 8 8 8 8 8

8.1.8 The One Dimensional Program

These are some statistics for the program using the One-dimensional array, or the State-value function, to contain the expected rewards. This program seem to generally do better than the other trimmed programs, but it seems not be able to reach 100% as easily as the other programs for some reason.

Table 12: The statistics for using the program using the one-dimensional array, and sarsa.

Setup	**Training**	**Total**	**Winrate**	**Starting State**
0.000s	0.016s	0.016s	100%	2 2 2
0.000s	0.026s	0.026s	97%	2 2 2 2
0.000s	0.035s	0.036s	100%	2 2 2 2 2
0.001s	0.051s	0.052s	97%	2 2 2 2 2 2
0.000s	0.025s	0.025s	100%	3 3 3
0.001s	0.050s	0.051s	89%	3 3 3 3
0.001s	0.074s	0.075s	90%	3 3 3 3 3
0.002s	0.124s	0.126s	96%	3 3 3 3 3 3
0.000s	0.034s	0.035s	98%	4 4 4
0.001s	0.087s	0.089s	96%	4 4 4 4
0.004s	0.143s	0.147s	68%	4 4 4 4 4
0.009s	0.304s	0.313s	98%	4 4 4 4 4 4
0.001s	0.049s	0.050s	94%	5 5 5
0.004s	0.151s	0.155s	84%	5 5 5 5
0.011s	0.291s	0.301s	84%	5 5 5 5 5
0.029s	0.722s	0.751s	81%	5 5 5 5 5 5
0.002s	0.068s	0.070s	90%	6 6 6
0.008s	0.250s	0.258s	88%	6 6 6 6
0.028s	0.610s	0.638s	64%	6 6 6 6 6
0.087s	1.385s	1.474s	46%	6 6 6 6 6 6
0.004s	0.091s	0.094s	89%	7 7 7
0.016s	0.413s	0.429s	89%	7 7 7 7
0.070s	1.088s	1.160s	58%	7 7 7 7 7
0.257s	2.529s	2.794s	32%	7 7 7 7 7 7
0.006s	0.128s	0.134s	58%	8 8 8
0.031s	0.635s	0.667s	64%	8 8 8 8
0.148s	1.656s	1.808s	44%	8 8 8 8 8
0.655s	4.147s	4.826s	23%	8 8 8 8 8 8

8.1.9 Phenomena: Total Time Use Is More than the Sum of the Training and Setup Time.

A phenomena found in the statistics for all of the different programs is the fact that at times the total time used is more than the training and setup times together. This seems strange, because it seems that they should be the same, and they mostly are. The Table 13 shows some examples of this occuring, from the two programs using the untrimmed game tree. It does happen with the other algorithms too, however.

Table 13: The lines which show a total time use that is more than the sum of the setup and train times.

Setup	Training	Total	Winrate	Starting State
RS				
0.455s	0.351s	0.820s	40 %	3 3 3 3 3
0.000s	0.021s	0.023s	100%	4 4
0.167s	0.226s	0.397s	43%	4 4 4 4
4.529s	1.200s	5.888s	14%	4 4 4 4 4
0.001s	0.025s	0.042s	100%	5 5
0.738s	0.438s	1.197	23%	5 5 5 5
27.906s	3.008s	32.590s	6%	5 5 5 5 5
RQ				
0.473s	0.611s	1.099s	42%	3 3 3 3 3
4.384s	1.805s	6.330s	14%	4 4 4 4 4
0.001s	0.032s	0.048s	100%	5 5
0.763s	0.714s	1.499s	24%	5 5 5 5
26.996s	4.696s	33.137s	6%	5 5 5 5 5

Hypothesis I believe this happens because the times train and setup do not measure in total the part of code that total does, it runs a little more code than the other 2 in total. See Listing 22 where the the total time is taken in the file *Run.py*. The setup and training time is in Listing 23 and the initialization, run only within the total time is in Listing 24. Based upon all that it seems reasonable that the computer might just have used a little extra time to initialize when running, skewing the numbers a bit.

```
57 start = time.time()
58 rs = RS.ReinforcementSarsa(state)
59 times = rs.setup()
60 end = time.time()
```

Listing 22: Total time is taken here. From file *Run.py*

```
23 def setup(self):
24   timeStart = time.time()
25   for i in range(self.sum):
```

```
26


46 timeEnd = time.time()
47 setupTime = timeEnd-timeStart
48 if(type == 0):
49   epsilon = 0
50   mu = 0.5
51   gamma = 0.5
52   amountEpisodes = 1000
53   timeStart = time.time()
54   self.train(amountEpisodes, epsilon, mu, gamma)
55   timeEnd = time.time()
56   trainTime = timeEnd-timeStart

58   times = []
59   times.append(setupTime)
60   times.append(trainTime)
61
62   return times
63 return -1
```

Listing 23: Setup and training time is taken here. From file *ReinforcementSarsa.py*, but is the same for all the similar programs.

```
8 def __init__(self, startBoard):
9   self.sum = startBoard[0]+1
10  self.board = startBoard
11  for i in range(1, len(self.board)):
12    self.sum *= self.board[i]+1
13  self.listOfStates = [0] * self.sum
14  self.listOfStates[0] = self.board[:]
15  self.T = []
16  self.Q = []
```

Listing 24: The initialization of the file, which is part of the what is run in the total time only. From file *ReinforcementSarsa.py*, but is the same for all the similar programs.

8.1.10 Phenomena: Irregular Decrease in Success-rate

A phenomena to be observed here, is the fact that the success rate does not decrease in a regular way, when given larger states. It seems to actually be so that states with an even number of heaps has a higher success rate, than states with an odd number of heaps. That is, the algorithm trains better when it starts with a state of an even amount of heaps. The difference between these states is clear, and that is that a state with an even amount of heaps of the same size is a losing state, it has a Nim-sum of 0, while a state with an odd number heaps of the same size is a winning state, its Nim-sum is not 0. Table 14.

Table 14: The strange phenomena, success rate is higher where there are an even number of heaps. From running TS.

Setup	**Training**	**Total**	**Winrate**	**Starting State**
0.000s	0.019s	0.019s	100%	4 4
0.001s	0.031s	0.032s	84%	4 4 4
0.004s	0.072s	0.076s	99%	4 4 4 4
0.011s	0.102s	0.113s	74%	4 4 4 4 4
0.029s	0.198s	0.228s	86%	4 4 4 4 4 4

Hypothesis I believe that because the even number of heaps is a losing position, the algorithm would be more exploretative, because its first move would mostly change each time, as the policy in use is ϵ-greedy. This means it tries out more positions, which might be beneficial. On the odd number of heaps it might however be too exploitative, and not explore all possible positions, as perhaps it should. If ϵ was increased, making it more explorative, the phenomena should in this case disappear, because its advantage would be nullified. With ϵ set to 1, where in the training every move is made by picking a random move, the expected result would be that the winrate should decrease properly. This would be because any extra exploration for some of the starting states would be nullified, as the policy would just be random moves anyway. The results were:

Results Table 15 shows that the winrate is now regular again, so I conclude that the effect was cased by some states causing more exploration, which is beneficial for the winrate. This makes sense, as the algorithms for reinforcement learning really find the best way to get from the start to the end, while in the case of solving Nim training for all possible states is desirable. Based upon this phenomena a better solution might be found. If the goal is more exploration with the goal being training from all states, I suggest that rather than just starting in the starting state, the training should start games from all possible positions, which might show results similar to those states with an even number of heaps, for all states.

Table 15: Statistics for trimmed sarsa, with ϵ set to 1.

Setup	Training	Total	Winrate	Starting State
0.000s	0.027s	0.027s	39%	5 5
0.003s	0.057s	0.060s	30%	5 5 5
0.011s	0.132s	0.143s	18%	5 5 5 5
0.039s	0.287s	0.327s	11%	5 5 5 5 5
0.001s	0.033s	0.034s	42%	6 6
0.005s	0.078s	0.083s	23%	6 6 6
0.028s	0.207s	0.235s	12%	6 6 6 6
0.129s	0.567s	0.698s	6%	6 6 6 6 6

8.1.11 Training From All States

The program is the similar to the trimmed programs, and uses the sarsa algorithm. The only difference is in the method for the training, of course, since only the training is different in theory. The new method is shown in Listing 25.

```
144 def train(self, amountEpisodes, epsilon, mu, gamma):
145   curStartState = 0
146   for neverUsed in range(amountEpisodes):
147     curState = curStartState
148     playing = 1
149     while(playing):
150       curAction = self.epsilonGreedyChoice(curState, epsilon)
151       curReward = self.Q[curState][curAction]
152       nextState = curAction
153       if(nextState == len(self.listOfStates)-1):
154         break
155       nextAction = self.epsilonGreedyChoice(nextState, epsilon)
156       self.Q[curState][curAction] -= mu * (curReward + gamma*self.Q
    [nextState][nextAction] - self.Q[curState][curAction])
157       curState = nextState
158       if (nextAction == self.sum-1):
159         playing = 0
160     curStartState += 1
161     if(curStartState >= len(self.listOfStates)-2):
162       curStartState = 0
```

Listing 25: The method *train* in the program that trains from all the states; *SpaceSaveSarsaSpread.py*

Table 16 contains the results for this program compared to the ordinary trimmed program using sarsa. As can be seen the new program does better in most cases, with the exception of some cases where there are an equal amount of heaps. This is expected, as the new program trains less per episode, as it starts from lower states where the game can be beaten with fewer moves. As for the timeuse, the training should be quicker for the new program, and it is.

Table 16: Comparing the Algorithm that trains from all states with Trimmed Sarsa.

Program	Training	Total	Winrate	Starting State
New Program	0.041s	0.047s	98%	6 6 6
TS	0.052s	0.058s	73%	
New Program	0.90s	0.122s	70%	6 6 6 6
TS	0.170s	0.202s	74%	
New Program	0.204s	0.350s	41%	6 6 6 6 6
TS	0.341s	0.482s	38%	
New Program	0.052s	0.063s	88%	7 7 7
TS	0.071s	0.081s	73%	
New Program	0.123s	0.196s	49%	7 7 7 7
TS	0.218s	0.294s	50%	
New Program	0.293s	0.686s	26%	7 7 7 7 7
TS	0.542s	0.973s	22%	
New Program	0.071s	0.094s	70%	8 8 8
TS	0.109s	0.131s	60%	
New Program	0.183s	0.350s	33%	8 8 8 8
TS	0.303s	0.467s	31%	
New Program	0.491s	1.537s	16%	8 8 8 8 8
TS	0.876s	1.957s	14%	
New Program	0.066s	0.093s	57%	9 9 9
TS	0.105s	0.132s	55%	
New Program	0.191s	0.450s	24%	9 9 9 9
TS	0.315s	0.579s	21%	
New Program	0.594s	2.527s	10%	9 9 9 9 9
TS	0.987s	3.047s	8%	

8.1.12 Difference in Time-use, Comparing the Agorithms

It is quite clear that the algorithms using the trimmed game trees uses less time, in fact, as can be seen in Table 17. They are also more successful, that is they have a higher winrate against the perfect player. This is to be expected after all, since a lesser game-tree means that the matrices are faster to set up, searching for the best move is also faster, as there are less to pick from, and there is a smaller Q to train, so finding the proper policy is easier.

Table 17: The timeuse and winrate for the different algorithms when given the gamestate: 5 5 5 5 5.

Program	Setup	Training	Total	Winrate	Starting State
RS	27.906s	3.008s	32.590s	6%	5 5 5 5 5
RQ	26.996s	4.696s	33.137s	6%	5 5 5 5 5
TS	0.037s	0.187s	0.225s	65%	5 5 5 5 5
TQ	0.037s	0.279s	0.317s	71%	5 5 5 5 5
ODS	0.011s	0.291s	0.301s	84%	5 5 5 5 5

It is clear that the growth of the setup time is greater than the growth of the training time for the programs that use the untrimmed game-tree. The same is true for the programs that use the trimmed game-tree. Table 18 Shows 2 examples of this. This is not surprising, given that the parts of the programs that use the most time, is going to be those that have to iterate over the matrices somehow, because the matrices are clearly the largest datastructures in the programs. Since the setup have to go through the whole matrix twice, it should always use the most time as the training will only go through one column the matrix an amount of time, which is at the highest the amount of counters in the startstate times the amount of episodes. For these statistics it always trains with 1000 episodes. As long as a matrix is used this will be the case. Thus we look towards the last program, the one that does away with the matrix altogether, and uses an array instead.

Table 18: How the timeuse develops for greater start-states.

Program	Setup	Training	Total	Winrate	Starting State
RS	0.001s	0.040s	0.041s	100%	1 1 1 1 1
RS	0.026s	0.104s	0.130s	72%	2 2 2 2 2
RS	0.455s	0.351s	0.820s	40 %	3 3 3 3 3
RS	4.529s	1.200s	5.888s	14%	4 4 4 4 4
RS	27.906s	3.008s	32.590s	6%	5 5 5 5 5
TS	0.124s	0.344s	0.472s	43%	5 5 5 5 5 5
TS	0.458s	0.581s	1.051s	22%	6 6 6 6 6 6
TS	1.518s	0.941s	2.499s	12%	7 7 7 7 7 7
TS	4.284s	1.655s	6.056s	6%	8 8 8 8 8 8
TS	11.957s	2.808s	15.306s	4%	9 9 9 9 9 9
TS	32.736s	4.666s	39.09s	2%	10 10 10 10 10 10
ODS	0.009s	0.304s	0.313s	98%	4 4 4 4 4 4
ODS	0.029s	0.722s	0.751s	81%	5 5 5 5 5 5
ODS	0.087s	1.385s	1.474s	46%	6 6 6 6 6 6
ODS	0.257s	2.529s	2.794s	32%	7 7 7 7 7 7
ODS	0.655s	4.147s	4.826s	23%	8 8 8 8 8 8

The last selection of statistics in Table 18 refers to the program that uses a one dimensional array to train. It is incidentally clear how much faster the trimmed programs are, so even though only certain states can be trained from, it is still better. If the input state was changed within the program to be the state of all heaps of the size of the biggest input heap, it would perhaps train much more of the game than necessary, but it would still in most cases be far quicker than the untrimmed programs.

8.1.13 Stochastic Termination

The program *Episodes.py* runs a program until it has been trained to reach a predefined winrate. This table of results are presented to argue against the use of stochastic termination. Table 19 are the result of running *Episodes.py* twice with program TS, with a desired winrate of 90%.

Table 19: The results of stochastic termination of TS

Time used	**Episodes used**	**startstate**
0.001s	10	2 2
0.000s	10	
0.046s	2010	2 2 2
0.004s	80	
0.001s	30	2 2 2 2
0.001s	20	
0.001s	20	3 3
0.001s	20	
1.149s	44210	3 3 3
0.069s	3510	
0.007s	90	3 3 3 3
0.012s	130	

This shows how unreliable stochastic termination might be, and how time-consuming training the program actually is, when one desires a high winrate. the training required for 3 3 3, for example, seems hard to predict, and from this it seems obvious the problem will just intensify. The reason that 3 3 3 3 is so much easier would be the phenomena described earlier.

8.1.14 Playing the Game

Thus far all the time-use has been based upon the time-use of the setup and the training. There is one part of the program that has not been explored yet, which is the time-use of the move-selection, that is, the time used when the program finds the best move it can take(based upon its statistics).

Table 20: Time-use of moves

Program	Move	Time-use	Current State
RS	5 counters from heap 4	0.001s	5 5 5 5 5
Deterministic	1 counter from heap 1	0.000s	5 5 5 0 5
RS	3 counters from heap 2	0.002s	4 5 5 0 5
TS	4 counters from heap 1	0.000s	8 8 8 8 8 8 8
Deterministic	4 counters from heap 1	0.000s	4 8 8 8 8 8 8
TS	5 counters from heap 2	0.000s	0 8 8 8 8 8 8
ODS	6 counters from heap 1	0.001s	8 8 8 8 8 8 8
Deterministic	2 counters from heap 1	0.000s	2 8 8 8 8 8 8
ODS	4 counters from heap 2	0.002s	0 8 8 8 8 8 8

Table 20 shows the timeuse for making moves of some selected programs. The states were chosen for being rather large, so that the limits of the programs might be tested. It only shows small sections of complete games, as the rest of the games had as small time-uses. The conclusion here is that making moves is very quick and simple in comparison to the time used for setup and training. That is to be expected as in the training the program might find moves several times, and as such should be slower. The timeuse of the deterministic algorithm is very small, which I predicted when I considered the theoretical time-use of the algorithms.

8.2 Supervised Learning

Separating training from setup doesnt make as much sense for this program, so time-use is just shown in total for the whole program. Table 21 shows that it generally does worse than the untrimmed programs, both in success and time-use. It also does not show the phenomena of higher winrates for states with an equal amount of heaps.

Table 21: Statistics of a run of the program for playing nim with Supervised learning.

Time-use	**Winrate**	**Starting state**
0.017s	68%	2 2
0.082s	45%	2 2 2
0.447s	41%	2 2 2 2
1.371s	30%	2 2 2 2 2
0.025s	61%	3 3
0.260s	44%	3 3 3
1.443s	33%	3 3 3 3
9.087s	16%	3 3 3 3 3
0.051s	57%	4 4
0.574s	29%	4 4 4
2.444s	14%	4 4 4 4
26.476s	6%	4 4 4 4 4
0.140s	62%	5 5
0.871s	24%	5 5 5
6.608s	9%	5 5 5 5
48.948s	3%	5 5 5 5 5

9 Conclusion

Solving Nim with machine learning is evidently possible. I am confident in saying that it will never be better than the deterministic solution. This seems quite obvious, as any machine learning solution is by definition stochastic, the supervised solution is undecidable, and the reinforcement solution is based upon the game-tree, which is too big, even when trimmed. The results show the same, so it seems a fair conclusion. That was not surprising however, because the deterministic solution is very good, and this problem is not within the area which machine learning is considered best. What to take from this is rather the troubles faced by the machine when trying to solve Nim, that is the problem of achieving general success when playing. That is that the amount of training is based upon the size of the game-tree, and the tree gets large, fast. In cases where the training graph remains fairly small, and does not increase too much, playing well from all positions from the graph should be feasable, especially of methods which are more explorative are used.

The Importance What is explored here might be interesting when approaching other problems with similar traits, and similar solutions might apply. For example, trimming the graph is important(But that might as well apply in all cases of reinforcement learning), training from all possible starts might help. I suspect that training for all states is more interesting when preparing for a human opponent, as their unpredictability makes being versatile more important.

9.1 Where Now?

There are some extra solutions and statistics that might be interesting, for an expanded understanding of the problem.

- Trying to train from all positions could be applied to ODS, which already has some advantages, to see if that union will create an even more successful program.
- Making a program that just checks how well the reinforcement programs do if the only goal is to play as well as possible from the startstate might be interesting, as that is closer to what reinforcement learning is considered good at.

10 The Code in Full

Here are the programs used in this paper, in full. They are all written in the language Python.

```
class PerfectPlayer:
  #This method applies the deterministic algorithm for making
    perfect moves in Nim, and returns the index of the move and the
     amoiunt of counters removed.
  def makeMove(self, state):
    nimSum = self.findNimSum(state)
    for i in range(len(state)):
      remove = nimSum^state[i]
      if(remove < state[i]):
        result = [state[i] - remove, i]
        return result
    for i in range(len(state)):
      if(state[i] != 0):
        result = [1, i]
        return result
    return -1

  #Since finding nimsums is relevant in the program Run.py this
    program finds the nimsum of the input state.
  def findNimSum(self, state):
    nimSum = 0
    for i in range(len(state)):
      nimSum = nimSum^state[i]
    return nimSum
```

Listing 26: This is the code in the file *PerfectPlayer.py*.

```
import time
import sys
import ReinforcementSarsa as RS
import ReinforcementQlearning as RQ
import PerfectPlayer as PP
import SpaceSaveSarsa as SS
import SpaceSaveQlearning as SQ
import OneDSarsa as ODS
import SpaceSaveSarsaSpread as SSSS

if(len(sys.argv) < 11):
  print("Error: too few arguments. Got " + str(len(sys.argv)-1) + "
    , Expected 10")
elif (len(sys.argv) > 11):
  print("Error: too many arguments. Got, " + str(len(sys.argv)-1) +
     ", Expected 10")
else:
  #Parsing the input.
  heapSizeLower = int(sys.argv[1])
  heapSizeUpper = int(sys.argv[2])
  heapAmountLower = int(sys.argv[3])
  heapAmountUpper = int(sys.argv[4])

  player = PP.PerfectPlayer()
```

```
ssstatistics = open("ssstatistics.txt", "w")

#writing the header of the statistics file, to show which numbers correspond to which program.
header = ""
if(int(sys.argv[5]) == 1):
  header += "\t\t\t\t\t\tRS\t"
if(int(sys.argv[6]) == 1):
  header += "\t\t\t\t\t\tRQ\t"
if(int(sys.argv[7]) == 1):
  header += "\t\t\t\t\t\tTS\t"
if(int(sys.argv[8]) == 1):
  header += "\t\t\t\t\t\tTQ\t"
if(int(sys.argv[9]) == 1):
  header += "\t\t\t\t\t\tODS\t"
if(int(sys.argv[10]) == 1):
  header += "\t\t\t\t\t\tNew Program"
header += "\n"
ssstatistics.write(header)
#The loop that defines the start-states from which the programs will be run.
for heapSize in range(heapSizeLower, heapSizeUpper+1):
  for heapAmount in range(heapAmountLower, heapAmountUpper+1):
    state = [heapSize] * heapAmount
    print(state)
    line = ""
    """This repeats for each program. For each start-state they are run 10 times, taking their time use, after which they are set to play against the perfect player
    for all starts it can play from, from which it can win when starting. That is, the states with a nim-sum != 0. The averages of the times and winrate is then
    written into the statistics file."""
    if(int(sys.argv[5]) == 1):
      timeAvg = 0
      percent = 0
      setupTime = 0
      trainTime = 0
      for i in range(10):
        start = time.time()
        rs = RS.ReinforcementSarsa(state)
        times = rs.setup()
        end = time.time()
        timeAvg += end-start
        setupTime += times[0]
        trainTime += times[1]
        listOfStates = rs.getListOfStates()
        wins = 0
        losses = 0
        for i in range(len(listOfStates)):
          if(player.findNimSum(listOfStates[i]) != 0):
            playing = 1
            board = listOfStates[i][:]
            winBoard = [0] * len(listOfStates[i])
            while(playing):
              move = rs.makeMove(board)
              board[move[1]] = board[move[1]]-move[0]
```

```
                if(board == winBoard):
                  wins += 1
                  break
                move = player.makeMove(board)
                board[move[1]] = board[move[1]]-move[0]
                if(board == winBoard):
                  losses += 1
                  break
          percent += (wins/(wins+losses)*100)
        timeAvg /= 10
        percent /= 10
        setupTime /= 10
        trainTime /= 10
        line += "setup: " + str(round(setupTime, 3)) + "s\t train:
    " + str(round(trainTime, 3)) + "s\t total: " + str(round(
    timeAvg, 3)) + "s\t" + str(int(percent)) + "%\t"
      if(int(sys.argv[6]) == 1):
        timeAvg = 0
        percent = 0
        setupTime = 0
        trainTime = 0
        for i in range(10):
          start = time.time()
          rq = RQ.ReinforcementQlearning(state)
          times = rq.setup()
          end = time.time()
          timeAvg += end-start
          setupTime += times[0]
          trainTime += times[1]
          listOfStates = rq.getListOfStates()
          wins = 0
          losses = 0
          for i in range(len(listOfStates)):
            if(player.findNimSum(listOfStates[i]) != 0):
              playing = 1
              board = listOfStates[i][:]
              winBoard = [0] * len(listOfStates[i])
              while(playing):
                move = rq.makeMove(board)
                board[move[1]] = board[move[1]]-move[0]
                if(board == winBoard):
                  wins += 1
                  break
                move = player.makeMove(board)
                board[move[1]] = board[move[1]]-move[0]
                if(board == winBoard):
                  losses += 1
                  break
          percent += (wins/(wins+losses)*100)
        timeAvg /= 10
        percent /= 10
        setupTime /= 10
        trainTime /= 10
        line += "setup: " + str(round(setupTime, 3)) + "s\t train:
    " + str(round(trainTime, 3)) + "s\t total: " + str(round(
    timeAvg, 3)) + "s\t" + str(int(percent)) + "%\t"
      if(int(sys.argv[7]) == 1):
```

```
128         timeAvg = 0
129         percent = 0
130         setupTime = 0
131         trainTime = 0
132         for i in range(10):
133           start = time.time()
134           sss = SS.SpaceSaveSarsa(state)
135           times = sss.setup()
136           end = time.time()
137           timeAvg += end-start
138           setupTime += times[0]
139           trainTime += times[1]
140           listOfStates = sss.getListOfStates()
141           wins = 0
142           losses = 0
143           for i in range(len(listOfStates)):
144             if(player.findNimSum(listOfStates[i]) != 0):
145               playing = 1
146               board = listOfStates[i][:]
147               winBoard = [0] * len(listOfStates[i])
148               while(playing):
149                 move = sss.makeMove(board)
150                 board[move[1]] = board[move[1]]-move[0]
151                 if(board == winBoard):
152                   wins += 1
153                   break
154                 move = player.makeMove(board)
155                 board[move[1]] = board[move[1]]-move[0]
156                 if(board == winBoard):
157                   losses += 1
158                   break
159           percent += (wins/(wins+losses)*100)
160         timeAvg /= 10
161         percent /= 10
162         setupTime /= 10
163         trainTime /= 10
164         line += "setup: " + str(round(setupTime, 3)) + "s\t train:
    " + str(round(trainTime, 3)) + "s\t total: " + str(round(
    timeAvg, 3)) + "s\t" + str(int(percent)) + "%\t"
165       if(int(sys.argv[8]) == 1):
166         timeAvg = 0
167         percent = 0
168         setupTime = 0
169         trainTime = 0
170         for i in range(10):
171           start = time.time()
172           ssq = SQ.SpaceSaveQlearning(state)
173           times = ssq.setup()
174           end = time.time()
175           timeAvg += end-start
176           setupTime += times[0]
177           trainTime += times[1]
178           listOfStates = ssq.getListOfStates()
179           wins = 0
180           losses = 0
181           for i in range(len(listOfStates)):
182             if(player.findNimSum(listOfStates[i]) != 0):
```

```
183               playing = 1
184               board = listOfStates[i][:]
185               winBoard = [0] * len(listOfStates[i])
186               while(playing):
187                 move = ssq.makeMove(board)
188                 board[move[1]] = board[move[1]]-move[0]
189                 if(board == winBoard):
190                   wins += 1
191                   break
192                 move = player.makeMove(board)
193                 board[move[1]] = board[move[1]]-move[0]
194                 if(board == winBoard):
195                   losses += 1
196                   break
197           percent += (wins/(wins+losses)*100)
198         timeAvg /= 10
199         percent /= 10
200         setupTime /= 10
201         trainTime /= 10
202         line += "setup: " + str(round(setupTime, 3)) + "s\t train:
    " + str(round(trainTime, 3)) + "s\t total: " + str(round(
    timeAvg, 3)) + "s\t" + str(int(percent)) + "%\t"
203       if(int(sys.argv[9]) == 1):
204         timeAvg = 0
205         percent = 0
206         setupTime = 0
207         trainTime = 0
208         for i in range(10):
209           start = time.time()
210           ods = ODS.OneDSarsa(state)
211           times = ods.setup()
212           end = time.time()
213           timeAvg += end-start
214           setupTime += times[0]
215           trainTime += times[1]
216           listOfStates = ods.getListOfStates()
217           wins = 0
218           losses = 0
219           for i in range(len(listOfStates)):
220             if(player.findNimSum(listOfStates[i]) != 0):
221               playing = 1
222               board = listOfStates[i][:]
223               winBoard = [0] * len(listOfStates[i])
224               while(playing):
225                 move = ods.makeMove(board)
226                 board[move[1]] = board[move[1]]-move[0]
227                 if(board == winBoard):
228                   wins += 1
229                   break
230                 move = player.makeMove(board)
231                 board[move[1]] = board[move[1]]-move[0]
232                 if(board == winBoard):
233                   losses += 1
234                   break
235           percent += (wins/(wins+losses)*100)
236         timeAvg /= 10
237         percent /= 10
```

```
        setupTime /= 10
        trainTime /= 10
        line += "setup: " + str(round(setupTime, 3)) + "s\t train:
    " + str(round(trainTime, 3)) + "s\t total: " + str(round(
    timeAvg, 3)) + "s\t" + str(int(percent)) + "%\t"
      if(int(sys.argv[10]) == 1):
        timeAvg = 0
        percent = 0
        setupTime = 0
        trainTime = 0
        for i in range(10):
          start = time.time()
          ssss = SSSS.SpaceSaveSarsaSpread(state)
          times = ssss.setup()
          end = time.time()
          timeAvg += end-start
          setupTime += times[0]
          trainTime += times[1]
          listOfStates = ssss.getListOfStates()
          wins = 0
          losses = 0
          for i in range(len(listOfStates)):
            if(player.findNimSum(listOfStates[i]) != 0):
              playing = 1
              board = listOfStates[i][:]
              winBoard = [0] * len(listOfStates[i])
              while(playing):
                move = ssss.makeMove(board)
                board[move[1]] = board[move[1]]-move[0]
                if(board == winBoard):
                  wins += 1
                  break
                move = player.makeMove(board)
                board[move[1]] = board[move[1]]-move[0]
                if(board == winBoard):
                  losses += 1
                  break
          percent += (wins/(wins+losses)*100)
        timeAvg /= 10
        percent /= 10
        setupTime /= 10
        trainTime /= 10
        line += "setup: " + str(round(setupTime, 3)) + "s\t train:
    " + str(round(trainTime, 3)) + "s\t total: " + str(round(
    timeAvg, 3)) + "s\t" + str(int(percent)) + "%\t"
      line += str(state) + "\n"
      ssstatistics.write(line)
```

Listing 27: This is the code in the file *Run.py*.

```
import time
import sys

import ReinforcementSarsa as RS
import ReinforcementQlearning as RQ
import PerfectPlayer as PP
import SpaceSaveSarsa as SS
import SpaceSaveQlearning as SQ
```

```
import OneDSarsa as ODS
import SpaceSaveSarsaSpread as SSSS

if(len(sys.argv) < 5):
  print("Error: too few arguments. Got " + str(len(sys.argv)-1) + "
    , Expected 4")
elif (len(sys.argv) > 5):
  print("Error: too many arguments. Got, " + str(len(sys.argv)-1) +
     ", Expected 4")
else:
  #Parsing the input
  startState = list(map(int, sys.argv[3].split(',')))
  trainingType = int(sys.argv[2])
  if(trainingType == 0):
    ml = RS.ReinforcementSarsa(startState)
  elif(trainingType == 1):
    ml = RQ.ReinforcementQlearning(startState)
  elif(trainingType == 2):
    ml = SS.SpaceSaveSarsa(startState)
  elif(trainingType == 3):
    ml = SQ.SpaceSaveQlearning(startState)
  elif(trainingType == 4):
    ml = ODS.OneDSarsa(startState)
  else:
    ml = SSSS.SpaceSaveSarsaSpread(startState)
  ml.setup()

  #The code for the reinforcement program playing against the
    deterministic program, and getting statistics from it.
  if(int(sys.argv[1]) == 0):
    playstatistics = open("playstatistics.txt", "w")
    playstatistics.write(str(trainingType) + "\n")
    player = PP.PerfectPlayer()
    playing = 1
    board = startState[:]
    winBoard = [0] * len(startState)
    #If the reinforcement algo gets the first move it gets to make
    a move before the loop start.
    if(int(sys.argv[4]) == 1):
      start = time.time()
      move = player.makeMove(board)
      end = time.time()
      moveTime = end-start
      board[move[1]] = board[move[1]]-move[0]
      playstatistics.write("Move by Pre-algo, took " + str(move[0])
     + " counters from heap " + str(move[1]+1) + ". New state: " +
    str(board) + " time used: " + str(round(moveTime, 3)) + "s.\n")
      if(board == winBoard):
        playstatistics.write("Pre-algo is the winner.")
        playing = 0
    #Otherwise the game starts here.
    while(playing):
      start = time.time()
      move = ml.makeMove(board)
      end = time.time()
      moveTime = end-start
      board[move[1]] = board[move[1]]-move[0]
```

```
60        playstatistics.write("Move by ML, took " + str(move[0]) + "
      counters from heap " + str(move[1]+1) + ". New state: " + str(
      board) + " time used: " + str(round(moveTime, 3)) + "s.\n")
61        if(board == winBoard):
62          playstatistics.write("ML is the winner.")
63          break
64        start = time.time()
65        move = player.makeMove(board)
66        end = time.time()
67        moveTime = end-start
68        board[move[1]] = board[move[1]]-move[0]
69        playstatistics.write("Move by Pre-algo, took " + str(move[0])
       + " counters from heap " + str(move[1]+1) + ". New state: " +
      str(board) + " time used: " + str(round(moveTime, 3)) + "s.\n")
70        if(board == winBoard):
71          playstatistics.write("Pre-algo is the winner.")
72          break
73    #If the player wants to play against the Reinforcement algo.
74    else:
75      player = PP.PerfectPlayer()
76      playing = 1
77      board = startState[:]
78      winBoard = [0] * len(startState)
79      #If the player wants to start first, they make a move first.
80      if(int(sys.argv[4]) == 1):
81        move = [-1, -1]
82        print("Current board is: " + str(board) + "\n")
83        move[1] = int(input("Which heap do you want to remove from? "
      ))-1
84        while((move[1] < 0 or move[1] >= len(board)) or board[move
      [1]] == 0):
85          move[1] = int(input("Illegal heap, choose again. "))-1
86        move[0] = int(input("How many counters do you wish to remove?
       "))
87        while(move[0] < 1 or move[0] > board[move[1]]):
88          move[0] = int(input("Illegal amount, choose again. "))
89        board[move[1]] = board[move[1]]-move[0]
90        if(board == winBoard):
91          print("You are the winner.")
92          playing = 0
93      #Otherwise the game starts here, with the reinforcement program
       making the first move.
94      while(playing):
95        move = ml.makeMove(board)
96        board[move[1]] = board[move[1]]-move[0]
97        print("Move by ML, took " + str(move[0]) + " counters from
      heap " + str(move[1]+1) + ".")
98        if(board == winBoard):
99          print("ML is the winner.")
100         break
101       move = [-1, -1]
102       print("Current board is: " + str(board) + "\n")
103       move[1] = int(input("Which heap do you want to remove from? "
      ))-1
104       while(move[1] < 0 or move[1] >= len(board) or board[move[1]]
      == 0):
105         move[1] = int(input("Illegal heap, choose again. "))-1
```

```
106       move[0] = int(input("How many counters do you wish to remove?
      "))
107       while(move[0] < 1 or move[0] > board[move[1]]):
108         move[0] = int(input("Illegal amount, choose again. "))
109       board[move[1]] = board[move[1]]-move[0]
110       if(board == winBoard):
111         print("You are the winner.")
112         break
```

Listing 28: This is the code in the file *Play.py*.

```
1  import numpy as np
2  import random
3  import time

5  class ReinforcementSarsa:
6
7    #The initialization takes the starting board, and finds the
       amount of states in its untrimmed game-tree, which is called
       self.sum. It also initializes T and Q.
8    def __init__(self, startBoard):
9      self.sum = startBoard[0]+1
10     self.board = startBoard
11     for i in range(1, len(self.board)):
12       self.sum *= self.board[i]+1
13     self.listOfStates = [0] * self.sum
14     self.listOfStates[0] = self.board[:]
15     self.T = []
16     self.Q = []

18   """This method is for setting up the matrices T and Q, as well as
        creating the list of states. For each state it makes all
       possible moves it can,
19   and for each of those, it adds them to their spot in listOfStates
       , if the state wasnt already there, using the method newIndex
       to find the states' index.
20   All those moves are also added to T, which searchingis the main
       purpose. Then Q is made, by setting all positions in T with 1,
       to have small random values in Q.
21   The other positions in Q gets the value of negative infinity, -np
       .inf, so that they are always smaller than the expected reward
       of a move.
22   Lastly the method sets up the inputs for the training, and runs
       it as well. Lastly the method takes the time used both by the
       setup and the training."""
23   def setup(self, type):
24     timeStart = time.time()
25     for i in range(self.sum):
26       self.T.append([0] * self.sum)
27       for x in range(len(self.listOfStates[i])):
28         for y in range(1, self.listOfStates[i][x] + 1):
29           tmpValue = self.newIndex(x, y)
30           self.T[i][i+tmpValue] = 1
31           if(self.listOfStates[i+tmpValue] == 0):
32             tmpList = self.listOfStates[i][:]
33             tmpList[x] -= y
34             self.listOfStates[i+tmpValue] = tmpList[:]
35
```

```
      self.Q.append([])
      for j in range(self.sum-1):
        if(self.T[i][j] == 1):
          self.Q[i].append(random.random())
        else:
          self.Q[i].append(-np.inf)
      if (self.T[i][self.sum-1] == 1):
        self.Q[i].append(100.0)
      else:
        self.Q[i].append(-np.inf)
    timeEnd = time.time()
    setupTime = timeEnd-timeStart
    if(type == 0):
      epsilon = 0
      mu = 0.5
      gamma = 0.5
      amountEpisodes = 1000
      timeStart = time.time()
      self.train(amountEpisodes, epsilon, mu, gamma)
      timeEnd = time.time()
      trainTime = timeEnd-timeStart

      times = []
      times.append(setupTime)
      times.append(trainTime)

      return times
    return -1

  # This method exists for finding, given a move, how far down the
    list of states the new state is.
  def newIndex(self, heapIndex, amount):
    newMove = 1
    for i in range(len(self.board)-1, heapIndex, -1):
      newMove *= self.board[i]+1
    newMove *= amount
    return newMove

  """This is the method that finds the best state to make a move to
    , from the input state. It does this by searching through list
    to find the current state,
  and then using np.argmax to find the following state with the
    highest expected reward. Then it finds the heap where the move
    was made,
  to find exactly what to return, that is the amount of counters
    removed and the index of the heap removed from."""
  def makeMove(self, state):
    curState = state[:]
    nextState = []
    move = []
    if (len(state)<len(self.listOfStates[0])):
      dif = len(self.listOfStates[0])-len(state)
      for i in range(dif):
        curState.append(0)
    for i in range(len(self.listOfStates)):
      if (curState == self.listOfStates[i]):
        nextState = self.listOfStates[np.argmax(self.Q[i])]
```

```
    for i in range(len(curState)):
      if (curState[i] != nextState[i]):
        move.append(curState[i] - nextState[i])
        move.append(i)
        return move

  """This method is for training the matrix Q, by using the Sarsa
    algorithm. The main difference is that it removes the
    calculated reward, instead of adding it.
  It trains by playing against itself, until it wins, and then
    starts again. each game is an episode."""
  def train(self, amountEpisodes, epsilon, mu, gamma):
    for neverUsed in range(amountEpisodes):
      curState = 0
      curAction = self.epsilonGreedyChoice(curState, epsilon)
      playing = 1
      while(playing):
        curReward = self.Q[curState][curAction]
        nextState = curAction
        nextAction = self.epsilonGreedyChoice(nextState, epsilon)
        self.Q[curState][curAction] -= mu * (curReward + gamma*self
    .Q[nextState][nextAction] - self.Q[curState][curAction])
        curState = nextState
        curAction = nextAction
        if (curAction == self.sum-1):
          playing = 0

  #This is an implementation of the epsilon greedy policy, by the
    use of numpy.argmax when finding the best move, or just random
    choice of the possible moves otherwise.
  def epsilonGreedyChoice(self, state, epsilon):
    if(random.random()<epsilon):
      tmpList = []
      for i in range(state, self.sum):
        if(self.T[state][i] == 1):
          tmpList.append(i)
      return tmpList[random.randrange(0, len(tmpList))]
    else:
      return np.argmax(self.Q[state])

  #This method returns the list of states, which is used to test
    the program, by having it play from many different states.
  def getListOfStates(self):
    return self.listOfStates
```

Listing 29: This is the code in the file *ReinforcementSarsa.py*.

```
import numpy as np
import random
import time

class ReinforcementQlearning:

  #The initialization takes the starting board, and finds the
    amount of states in its untrimmed game-tree, which is called
    self.sum. It also initializes T and Q.
  def __init__(self, startBoard):
    self.sum = startBoard[0]+1
```

```
10      self.board = startBoard
11      for i in range(1, len(self.board)):
12        self.sum *= self.board[i]+1
13      self.listOfStates = [0] * self.sum
14      self.listOfStates[0] = self.board[:]
15      self.T = []
16      self.Q = []
17
18    """This method is for setting up the matrices T and Q, as well as
         creating the list of states. For each state it makes all
        possible moves it can,
19    and for each of those, it adds them to their spot in listOfStates
        , if the state wasnt already there, using the method newIndex
        to find the states' index.
20    All those moves are also added to T, which searchingis the main
        purpose. Then Q is made, by setting all positions in T with 1,
        to have small random values in Q.
21    The other positions in Q gets the value of negative infinity, -np
        .inf, so that they are always smaller than the expected reward
        of a move.
22    Lastly the method sets up the inputs for the training, and runs
        it as well. Lastly the method takes the time used both by the
        setup and the training."""
23    def setup(self, type):
24      timeStart = time.time()
25      for i in range(self.sum):
26        self.T.append([0] * self.sum)
27        for x in range(len(self.listOfStates[i])):
28          for y in range(1, self.listOfStates[i][x] + 1):
29            tmpValue = self.newIndex(x, y)
30            self.T[i][i+tmpValue] = 1
31            if(self.listOfStates[i+tmpValue] == 0):
32              tmpList = self.listOfStates[i][:]
33              tmpList[x] -= y
34              self.listOfStates[i+tmpValue] = tmpList[:]
35
36        self.Q.append([])
37        for j in range(self.sum-1):
38          if(self.T[i][j] == 1):
39            self.Q[i].append(random.random())
40          else:
41            self.Q[i].append(-np.inf)
42        if (self.T[i][self.sum-1] == 1):
43          self.Q[i].append(100.0)
44        else:
45          self.Q[i].append(-np.inf)
46      timeEnd = time.time()
47      setupTime = timeEnd-timeStart
48      if(type == 0):
49        epsilon = 0
50        mu = 0.5
51        gamma = 0.5
52        amountEpisodes = 1000
53        timeStart = time.time()
54        self.train(amountEpisodes, epsilon, mu, gamma)
55        timeEnd = time.time()
56        trainTime = timeEnd-timeStart
```

```
57
58          times = []
59          times.append(setupTime)
60          times.append(trainTime)
61
62          return times
63        return -1
64
65      # This method exists for finding, given a move, how far down the
          list of states the new state is.
66      def newIndex(self, heapIndex, amount):
67        newMove = 1
68        for i in range(len(self.board)-1, heapIndex, -1):
69          newMove *= self.board[i]+1
70        newMove *= amount
71        return newMove
72
73      """This is the method that finds the best state to make a move to
          , from the input state. It does this by searching through list
          to find the current state,
74      and then using np.argmax to find the following state with the
          highest expected reward. Then it finds the heap where the move
          was made,
75      to find exactly what to return, that is the amount of counters
          removed and the index of the heap removed from."""
76      def makeMove(self, state):
77        curState = state[:]
78        nextState = []
79        move = []
80        if (len(state)<len(self.listOfStates[0])):
81          dif = len(self.listOfStates[0])-len(state)
82          for i in range(dif):
83            curState.append(0)
84        for i in range(len(self.listOfStates)):
85          if (curState == self.listOfStates[i]):
86            nextState = self.listOfStates[np.argmax(self.Q[i])]
87        for i in range(len(curState)):
88          if (curState[i] != nextState[i]):
89            move.append(curState[i] - nextState[i])
90            move.append(i)
91            return move
92
93      """This method is for training the matrix Q, by using the Q-
          learning algorithm. The main difference is that it removes the
          calculated reward, instead of adding it.
94      It trains by playing against itself, until it wins, and then
          starts again. each game is an episode."""
95      def train(self, amountEpisodes, epsilon, mu, gamma):
96        for neverUsed in range(amountEpisodes):
97          curState = 0
98          playing = 1
99          while(playing):
100           curAction = self.epsilonGreedyChoice(curState, epsilon)
101           curReward = self.Q[curState][curAction]
102           nextState = curAction
103           nextAction = np.argmax(self.Q[nextState])
104           self.Q[curState][curAction] -= mu * (curReward + gamma*self
```

```
    .Q[nextState][nextAction] - self.Q[curState][curAction])
105         curState = nextState
106         if (nextAction == self.sum-1):
107           playing = 0

109   #This is an implementation of the epsilon greedy policy, by the
        use of numpy.argmax when finding the best move, or just random
        choice of the possible moves otherwise.
110   def epsilonGreedyChoice(self, state, epsilon):
111     if(random.random()<epsilon):
112       tmpList = []
113       for i in range(state, self.sum):
114         if(self.T[state][i] == 1):
115           tmpList.append(i)
116       return tmpList[random.randrange(0, len(tmpList))]
117     else:
118       return np.argmax(self.Q[state])

120   #This method returns the list of states, which is used to test
        the program, by having it play from many different states.
121   def getListOfStates(self):
122     return self.listOfStates
```

Listing 30: This is the code in the file *ReinforcementQlearning.py*.

```
1  import numpy as np
2  import random
3  import math
4  import time

6  class SpaceSaveSarsa:

8    #The initialization takes the starting board, and finds the
       amount of states in its untrimmed game-tree, which is called
       self.sum. It also initializes T and Q.
9    def __init__(self, startBoard):
10     self.board = startBoard
11     self.sum = self.getSum()
12     self.board.sort(reverse = True)
13     self.listOfStates = []
14     self.listOfStates.append(self.board[:])
15     self.T = []
16     self.Q = []

18   #This method is for finding the amount of states reachable from
       the start-state, for the trimmed game-tree.
19   def getSum(self):
20     return int(math.factorial(len(self.board)+ self.board[0])/(math
       .factorial(len(self.board))*math.factorial(self.board[0])))

22   #This method is for taking a state and a move in the state, and
       returning the resulting state in the correct order.
23   def newState(self, heapIndex, amount, state):
24     curState = state[:]
25     curAmount = amount
26     for i in range(heapIndex, len(state)-1):
27       dif = curState[i] - curState[i+1]
28       if (dif >= curAmount):
```

```
29          curState[i] -= curAmount
30          curAmount = 0
31          break
32        else:
33          curState[i] -= dif
34          curAmount -= dif
35      if (curAmount > 0):
36        curState[len(curState)-1] -= curAmount
37      return curState

39    """This method is for setting up the matrices T and Q, as well as
         creating the list of states. For each state it makes all
        possible moves it can,
40    and for each of those, it adds them to their spot in listOfStates
        . The function newState is used to get the resulting state from
         a given move.
41    the states are added at the end of the list of states in this
        case, unless they are already in the list.
42    All those moves are also added to T, which is the main purpose.
        Then Q is made, by setting all positions in T with 1, to have
        small random values in Q.
43    The other positions in Q gets the value of negative infinity, -np
        .inf, so that they are always smaller than the expected reward
        of a move.
44    Lastly the method sets up the inputs for the training, and runs
        it as well. Lastly the method takes the time used both by the
        setup and the training."""
45    def setup(self, type = 0):
46      timeStart = time.time()
47      emptyBoard = [0] * len(self.listOfStates[0])
48      for i in range(self.sum):
49        self.T.append([0] * self.sum)
50        for x in range(len(self.listOfStates[i])-1, -1, -1):
51          atEnd = 0
52          curIndex = i+1
53          for y in range(1, self.listOfStates[i][x]+1):
54            newState = self.newState(x, y, self.listOfStates[i])
55            if(atEnd):
56              self.T[i][curIndex] = 1
57              self.listOfStates.append(newState)
58              curIndex += 1
59            else:
60              atEnd = 1
61              for j in range(curIndex, len(self.listOfStates)):
62                if (newState == self.listOfStates[j]):
63                  self.T[i][j] = 1
64                  curIndex = j+1
65                  atEnd = 0
66                  break
67              if (atEnd):
68                self.T[i][len(self.listOfStates)] = 1
69                self.listOfStates.append(newState)
70                curIndex = len(self.listOfStates)
71        self.Q.append([])
72        for j in range(self.sum-1):
73          if(self.T[i][j] == 1):
74            self.Q[i].append(random.random())
```

```
        else:
          self.Q[i].append(-np.inf)
      if (self.T[i][self.sum-1] == 1):
        self.Q[i].append(100.0)
      else:
        self.Q[i].append(-np.inf)
    timeEnd = time.time()
    setupTime = timeEnd-timeStart
    if(type == 0):
      epsilon = 0.1
      mu = 0.5
      gamma = 0.5
      amountEpisodes = 1000
      timeStart = time.time()
      self.train(amountEpisodes, epsilon, mu, gamma)
      timeEnd = time.time()
      trainTime = timeEnd-timeStart

      times = []
      times.append(setupTime)
      times.append(trainTime)

      return times
    return -1

  """ This is the method for making a move, it finds the state(
    which is sorted first) in the list of states, by calculating
    its index. Then the best move is found by use
  of the function np.argmax. Lastly it identifies the index of the
    heap in which the move is made, and it finds a heap of the same
     size
  in the input state."""
  def makeMove(self, state):
    curState = state[:]
    curState.sort(reverse = True)
    nextState = []
    move = []
    if (len(state)<len(self.listOfStates[0])):
      dif = len(self.listOfStates[0])-len(state)
      for i in range(dif):
        curState.append(0)
    goal = 0
    progress = self.board[0]
    for i in range(len(self.board)):
      heaps = len(self.board)- (i+1)
      for j in range(progress, -1, -1):
        if(j == curState[i]):
          break
        else:
          if(heaps == 0):
            goal += j-curState[i]
            break
          result = 1
          for x in range(1, heaps + 1):
            result *= (j + x)
          goal += (result)//math.factorial(heaps)
          progress -= 1
```

```
    nextState = self.listOfStates[np.argmax(self.Q[goal])][:]
    numToDec = 0
    for i in range(len(curState)-1, -1, -1):
      if (curState[i] != nextState[i]):
        numToDec += curState[i]-nextState[i]
        newIndex = curState[i]
    move.append(numToDec)
    for i in range(len(state)):
      if(newIndex == state[i]):
        newIndex = i
        break
    move.append(newIndex)
    return move

  """This method is for training the matrix Q, by using the Sarsa
    algorithm. The main difference is that it removes the
    calculated reward, instead of adding it.
  It trains by playing against itself, until it wins, and then
    starts again. each game is an episode."""
  def train(self, amountEpisodes, epsilon, mu, gamma):
    for neverUsed in range(amountEpisodes):
      curState = 0
      curAction = self.epsilonGreedyChoice(curState, epsilon)
      playing = 1
      while(playing):
        curReward = self.Q[curState][curAction]
        nextState = curAction
        nextAction = self.epsilonGreedyChoice(nextState, epsilon)
        self.Q[curState][curAction] -= mu * (curReward + gamma*self
    .Q[nextState][nextAction] - self.Q[curState][curAction])
        curState = nextState
        curAction = nextAction
        if (curAction == self.sum-1):
          playing = 0

  #This is an implementation of the epsilon greedy policy, by the
    use of numpy.argmax when finding the best move, or just random
    choice of the possible moves otherwise.
  def epsilonGreedyChoice(self, state, epsilon):
    if(random.random()<epsilon):
      tmpList = []
      for i in range(state, self.sum):
        if(self.T[state][i] == 1):
          tmpList.append(i)
      return tmpList[random.randrange(0, len(tmpList))]
    else:
      return np.argmax(self.Q[state])

  #This method returns the list of states, which is used to test
    the program, by having it play from many different states.
  def getListOfStates(self):
    return self.listOfStates
```

Listing 31: This is the code in the file *SpaceSaveSarsa.py*.

```
import numpy as np
import random
import math
```

```
import time

class SpaceSaveQlearning:

  #The initialization takes the starting board, and finds the
    amount of states in its untrimmed game-tree, which is called
    self.sum. It also initializes T and Q.
  def __init__(self, startBoard):
    self.board = startBoard
    self.sum = self.getSum()
    self.board.sort(reverse = True)
    self.listOfStates = []
    self.listOfStates.append(self.board[:])
    self.T = []
    self.Q = []

  # This method is for finding the amount of states reachable from
    the start-state, for the trimmed game-tree.
  def getSum(self):
    return int(math.factorial(len(self.board)+ self.board[0])/(math
    .factorial(len(self.board))*math.factorial(self.board[0])))

  #This method is for taking a state and a move in the state, and
    returning the resulting state in the correct order.
  def newState(self, heapIndex, amount, state):
    curState = state[:]
    curAmount = amount
    for i in range(heapIndex, len(state)-1):
      dif = curState[i] - curState[i+1]
      if (dif >= curAmount):
        curState[i] -= curAmount
        curAmount = 0
        break
      else:
        curState[i] -= dif
        curAmount -= dif
    if (curAmount > 0):
      curState[len(curState)-1] -= curAmount
    return curState

  """This method is for setting up the matrices T and Q, as well as
     creating the list of states. For each state it makes all
    possible moves it can,
  and for each of those, it adds them to their spot in listOfStates
    . The function newState is used to get the resulting state from
     a given move.
  the states are added at the end of the list of states in this
    case, unless they are already in the list.
  All those moves are also added to T, which is the main purpose.
    Then Q is made, by setting all positions in T with 1, to have
    small random values in Q.
  The other positions in Q gets the value of negative infinity, -np
    .inf, so that they are always smaller than the expected reward
    of a move.
  Lastly the method sets up the inputs for the training, and runs
    it as well. Lastly the method takes the time used both by the
    setup and the training."""
```

```
45  def setup(self, type):
46    timeStart = time.time()
47    emptyBoard = [0] * len(self.listOfStates[0])
48    for i in range(self.sum):
49      self.T.append([0] * self.sum)
50      for x in range(len(self.listOfStates[i])-1, -1, -1):
51        atEnd = 0
52        curIndex = i+1
53        for y in range(1, self.listOfStates[i][x]+1):
54          newState = self.newState(x, y, self.listOfStates[i])
55          if(atEnd):
56            self.T[i][curIndex] = 1
57            self.listOfStates.append(newState)
58            curIndex += 1
59          else:
60            atEnd = 1
61            for j in range(curIndex, len(self.listOfStates)):
62              if (newState == self.listOfStates[j]):
63                self.T[i][j] = 1
64                curIndex = j+1
65                atEnd = 0
66                break
67            if (atEnd):
68              self.T[i][len(self.listOfStates)] = 1
69              self.listOfStates.append(newState)
70              curIndex = len(self.listOfStates)
71      self.Q.append([])
72      for j in range(self.sum-1):
73        if(self.T[i][j] == 1):
74          self.Q[i].append(random.random())
75        else:
76          self.Q[i].append(-np.inf)
77      if (self.T[i][self.sum-1] == 1):
78        self.Q[i].append(100.0)
79      else:
80        self.Q[i].append(-np.inf)
81    timeEnd = time.time()
82    setupTime = timeEnd-timeStart
83    if(type == 0):
84      epsilon = 0
85      mu = 0.5
86      gamma = 0.5
87      amountEpisodes = 1000
88      timeStart = time.time()
89      self.train(amountEpisodes, epsilon, mu, gamma)
90      timeEnd = time.time()
91      trainTime = timeEnd-timeStart
92
93      times = []
94      times.append(setupTime)
95      times.append(trainTime)
96
97      return times
98    return -1

100 """ This is the method for making a move, it finds the state(
      which is sorted first) in the list of states, by calculating
```

```
    its index. Then the best move is found by use
101 of the function np.argmax. Lastly it identifies the index of the
    heap in which the move is made, and it finds a heap of the same
     size
102 in the input state."""
103   def makeMove(self, state):
104     curState = state[:]
105     curState.sort(reverse = True)
106     nextState = []
107     move = []
108     if (len(state)<len(self.listOfStates[0])):
109       dif = len(self.listOfStates[0])-len(state)
110       for i in range(dif):
111         curState.append(0)
112     goal = 0
113     progress = self.board[0]
114     for i in range(len(self.board)):
115       heaps = len(self.board)- (i+1)
116       for j in range(progress, -1, -1):
117         if(j == curState[i]):
118           break
119         else:
120           if(heaps == 0):
121             goal += j-curState[i]
122             break
123           result = 1
124           for x in range(1, heaps + 1):
125             result *= (j + x)
126           goal += (result)//math.factorial(heaps)
127           progress -= 1
128     nextState = self.listOfStates[np.argmax(self.Q[goal])][:]
129     numToDec = 0
130     for i in range(len(curState)-1, -1, -1):
131       if (curState[i] != nextState[i]):
132         numToDec += curState[i]-nextState[i]
133         newIndex = curState[i]
134     move.append(numToDec)
135     for i in range(len(state)):
136       if(newIndex == state[i]):
137         newIndex = i
138         break
139     move.append(newIndex)
140     return move
141
142   """This method is for training the matrix Q, by using the Q-
      learning algorithm. The main difference is that it removes the
      calculated reward, instead of adding it.
143   It trains by playing against itself, until it wins, and then
      starts again. each game is an episode."""
144   def train(self, amountEpisodes, epsilon, mu, gamma):
145     for neverUsed in range(amountEpisodes):
146       curState = 0
147       playing = 1
148       while(playing):
149         curAction = self.epsilonGreedyChoice(curState, epsilon)
150         curReward = self.Q[curState][curAction]
151         nextState = curAction
```

```
        nextAction = np.argmax(self.Q[nextState])
        self.Q[curState][curAction] -= mu * (curReward + gamma*self
    .Q[nextState][nextAction] - self.Q[curState][curAction])
        curState = nextState
        if (nextAction == self.sum-1):
          playing = 0

  #This is an implementation of the epsilon greedy policy, by the
    use of numpy.argmax when finding the best move, or just random
    choice of the possible moves otherwise.
  def epsilonGreedyChoice(self, state, epsilon):
    if(random.random()<epsilon):
      tmpList = []
      for i in range(state, self.sum):
        if(self.T[state][i] == 1):
          tmpList.append(i)
      return tmpList[random.randrange(0, len(tmpList))]
    else:
      return np.argmax(self.Q[state])

  #This method returns the list of states, which is used to test
    the program, by having it play from many different states.
  def getListOfStates(self):
    return self.listOfStates
```

Listing 32: This is the code in the file *SpaceSaveQlearning.py*.

```
import numpy as np
import random
import math
import time

class OneDSarsa:

  #The initialization takes the starting board, and finds the
    amount of states in its untrimmed game-tree, which is called
    self.sum. It also initializes T and Q.
  def __init__(self, startBoard):
    self.board = startBoard
    self.sum = self.getSum()
    self.board.sort(reverse = True)
    self.listOfStates = []
    self.listOfStates.append(self.board[:])
    self.T = []
    self.Q = []

  #This method is for finding the amount of states reachable from
    the start-state, for the trimmed game-tree.
  def getSum(self):
    return int(math.factorial(len(self.board)+ self.board[0])/(math
    .factorial(len(self.board))*math.factorial(self.board[0])))

  #This method is for taking a state and a move in the state, and
    returning the resulting state in the correct order.
  def newState(self, heapIndex, amount, state):
    curState = state[:]
    curAmount = amount
    for i in range(heapIndex, len(state)-1):
```

```
27        dif = curState[i] - curState[i+1]
28        if (dif >= curAmount):
29          curState[i] -= curAmount
30          curAmount = 0
31          break
32        else:
33          curState[i] -= dif
34          curAmount -= dif
35      if (curAmount > 0):
36        curState[len(curState)-1] -= curAmount
37      return curState
38
39    """This method is for setting up the matrices T and Q, as well as
         creating the list of states. For each state it makes all
        possible moves it can,
40    and for each of those, it adds them to their spot in listOfStates
        . The function newState is used to get the resulting state from
         a given move.
41    the states are added at the end of the list of states in this
        case, unless they are already in the list.
42    All those moves are also added to T, which is the main purpose.
        Then Q is made, and since each state's index in it is reachable
         all fields
43    are filled with small random numbers.
44    Lastly the method sets up the inputs for the training, and runs
        it as well. Lastly the method takes the time used both by the
        setup and the training."""
45    def setup(self, type = 0):
46      timeStart = time.time()
47      emptyBoard = [0] * len(self.listOfStates[0])
48      for i in range(self.sum):
49        self.T.append([0] * self.sum)
50        for x in range(len(self.listOfStates[i])-1, -1, -1):
51          atEnd = 0
52          curIndex = i+1
53          for y in range(1, self.listOfStates[i][x]+1):
54            newState = self.newState(x, y, self.listOfStates[i])
55            if(atEnd):
56              self.T[i][curIndex] = 1
57              self.listOfStates.append(newState)
58              curIndex += 1
59            else:
60              atEnd = 1
61              for j in range(curIndex, len(self.listOfStates)):
62                if (newState == self.listOfStates[j]):
63                  self.T[i][j] = 1
64                  curIndex = j+1
65                  atEnd = 0
66                  break
67              if (atEnd):
68                self.T[i][len(self.listOfStates)] = 1
69                self.listOfStates.append(newState)
70                curIndex = len(self.listOfStates)
71        if(i == (self.sum-1)):
72          self.Q.append(100.0)
73        else:
74          self.Q.append(random.random())
```

```
    timeEnd = time.time()
    setupTime = timeEnd-timeStart
    if(type == 0):
      epsilon = 0
      mu = 0.5
      gamma = 0.5
      amountEpisodes = 1000
      timeStart = time.time()
      self.train(amountEpisodes, epsilon, mu, gamma)
      timeEnd = time.time()
      trainTime = timeEnd-timeStart

      times = []
      times.append(setupTime)
      times.append(trainTime)

      return times
    return -1

  """ This is the method for making a move, it finds the state(
    which is sorted first) in the list of states, by calculating
    its index. Then the best move is found,
  By checking which of the accessible positions are the best.
    Lastly it identifies the index of the heap in which the move is
     made, and it finds a heap of the same size
  in the input state."""
  def makeMove(self, state):
    curState = state[:]
    curState.sort(reverse = True)
    nextState = []
    move = []
    if (len(state)<len(self.listOfStates[0])):
      dif = len(self.listOfStates[0])-len(state)
      for i in range(dif):
        curState.append(0)
    goal = 0
    progress = self.board[0]
    for i in range(len(self.board)):
      heaps = len(self.board)- (i+1)
      for j in range(progress, -1, -1):
        if(j == curState[i]):
          break
        else:
          if(heaps == 0):
            goal += j-curState[i]
            break
          result = 1
          for x in range(1, heaps + 1):
            result *= (j + x)
          goal += (result)//math.factorial(heaps)
          progress -= 1
    highVal = -np.inf
    highValIndex = -1
    for j in range(goal+1, len(self.Q)):
      if(self.T[goal][j] == 1 and np.greater(self.Q[j], highVal)):
        highVal = self.Q[j]
        highValIndex = j
```

```
    nextState = self.listOfStates[highValIndex][:]
    numToDec = 0
    for i in range(len(curState)-1, -1, -1):
      if (curState[i] != nextState[i]):
        numToDec += curState[i]-nextState[i]
        newIndex = curState[i]
    move.append(numToDec)
    for i in range(len(state)):
      if(newIndex == state[i]):
        newIndex = i
        break
    move.append(newIndex)
    return move

  """This method is for training the matrix Q, by using the Sarsa
    algorithm. The main difference is that it removes the
    calculated reward, instead of adding it.
  It trains by playing against itself, until it wins, and then
    starts again. each game is an episode."""
  def train(self, amountEpisodes, epsilon, mu, gamma):
    for neverUsed in range(amountEpisodes):
      curState = 0
      curAction = self.epsilonGreedyChoice(curState, epsilon)
      playing = 1
      while(playing):
        curReward = self.Q[curAction]
        nextState = curAction
        nextAction = self.epsilonGreedyChoice(nextState, epsilon)
        self.Q[curAction] -= mu * (curReward + gamma*self.Q[
    nextAction] - self.Q[curAction])
        curState = nextState
        curAction = nextAction
        if (curAction == self.sum-1):
          playing = 0

  #This is an implementation of the epsilon greedy policy, by the
    use of numpy.argmax when finding the best move, or just random
    choice of the possible moves otherwise.
  """This is an implementation of the epsilon greedy policy,
    however it finds the best move without using np.argmax, as Q
    doesnt correspond to T in the same way here,
  as it is only a 1-D array."""
  def epsilonGreedyChoice(self, state, epsilon):
    if(random.random()<epsilon):
      tmpList = []
      for i in range(state, self.sum):
        if(self.T[state][i] == 1):
          tmpList.append(i)
      return tmpList[random.randrange(0, len(tmpList))]
    else:
      highVal = -np.inf
      highValIndex = -1
      for i in range(state+1, len(self.Q)):
        if(self.T[state][i] == 1 and np.greater(self.Q[i], highVal)
    ):
          highVal = self.Q[i]
          highValIndex = i
```

```
176       return highValIndex

178   #This method returns the list of states, which is used to test
        the program, by having it play from many different states.
179   def getListOfStates(self):
180     return self.listOfStates
```

Listing 33: This is the code in the file *OneDSarsa.py*.

```
1  import MakeData as MD
2  import PerfectPlayer as PP
3  import numpy as np
4  import mlp
5  import sys
6  import time

8  if(len(sys.argv) < 5):
9    print("Error: too few arguments. Got " + str(len(sys.argv)-1) + "
       , Expected 4")
10 elif (len(sys.argv) > 5):
11   print("Error: too many arguments. Got, " + str(len(sys.argv)-1) +
       ", Expected 4")
12 else:
13   #Parsing the input.
14   heapSizeLower = int(sys.argv[1])
15   heapSizeUpper = int(sys.argv[2])
16   heapAmountLower = int(sys.argv[3])
17   heapAmountUpper = int(sys.argv[4])
18   f = open("mlpstatistics.txt", "w")
19   filename = "CorrectMoves.txt"
20   player = PP.PerfectPlayer()
21   #The loop that defines the start-states from which the programs
       will be run.
22   for heapSize in range(heapSizeLower, heapSizeUpper+1):
23     for heapAmount in range(heapAmountLower, heapAmountUpper+1):
24       """The Supervised learning program is run 10 times for each
       start-state, and plays against the perfect player each time,
25       to get a winrate for statistics. The time used is also
       written to file as statistics. The data used is the same for a
       given state,
26       but it is sorted differently for each time."""
27       state = [heapSize] * heapAmount
28       print(state)
29       md = MD.MakeData()
30       md.setup(state)
31       timeAvg = 0
32       percent = 0
33       for i in range(10):
34         #sorting and separating the data into the needed categories
       .
35         data = np.loadtxt(filename)

37         stateLen = len(data[0])

39         boards = data[::2]
40         moves = data[1::2]

42         order = list(range(np.shape(boards)[0]))
```

```
        np.random.shuffle(order)
        boards = boards[order,:]
        moves = moves[order,:]

        train = boards[::2]
        trainTargets = moves[::2]

        valid = boards[1::4]
        validTargets = moves[1::4]

        test = boards[3::4]
        testTargets = moves[3::4]

        hidden = len(state)+10

        start = time.time()
        net = mlp.mlp(hidden, stateLen)
        net.earlystopping(train, trainTargets, valid, validTargets)
        end = time.time()
        timeAvg += end-start
        listOfStates = md.getStates()
        wins = 0
        losses = 0
        for i in range(len(listOfStates)):
          if(player.findNimSum(listOfStates[i]) != 0):
            playing = 1
            board = listOfStates[i][:]
            winBoard = [0] * len(listOfStates[i])
            while(playing):
              move = net.makeMove(board)
              board[move[1]] = board[move[1]]-move[0]
              if(board == winBoard):
                wins += 1
                break
              move = player.makeMove(board)
              board[move[1]] = board[move[1]]-move[0]
              if(board == winBoard):
                losses += 1
                break
        percent += (wins/(wins+losses)*100)
      timeAvg /= 10
      percent /= 10
      line = "Time used:" + str(round(timeAvg, 3)) + "s Winrate/
    sucessrate: " + str(int(percent)) + "\\% "
      line += str(state) + "\n"
      f.write(line)
```

Listing 34: This is the code in the file *RunMlp.py*.

```
import PerfectPlayer as PP

class MakeData:
  #This method just makes the list of states accessible from the
    start-state, and then runs the writeToFile method.
  def setup(self, startBoard):
    sum = startBoard[0]+1
    board = startBoard
    for i in range(1, len(board)):
```

```
      sum *= board[i]+1
    self.listOfStates = [0] * sum
    self.listOfStates[0] = board[:]
    for i in range(sum):
      for x in range(len(self.listOfStates[i])):
        for y in range(1, self.listOfStates[i][x] + 1):
          tmpValue = self.newIndex(x, y, board)
          if(self.listOfStates[i+tmpValue] == 0):
            tmpList = self.listOfStates[i][:]
            tmpList[x] -= y
            self.listOfStates[i+tmpValue] = tmpList[:]
    self.writeToFile(self.listOfStates)

  # This method exists for finding, given a move, how far down the
    list of states the new state is.
  def newIndex(self, heapIndex, amount, board):
    newMove = 1
    for i in range(len(board)-1, heapIndex, -1):
      newMove *= board[i]+1
    newMove *= amount
    return newMove

  #This file writes each state in the list to the file, and a
    correct move to make when in that state.
  def writeToFile(self, states):
    player = PP.PerfectPlayer()
    f = open("CorrectMoves.txt", "w")
    for i in range(len(states)):
      if (player.findNimSum(states[i]) != 0):
        tmp = [0]*len(states[0])
        result = player.makeMove(states[i])
        tmp[result[1]] = result[0]
        for j in range(len(states[i])):
          f.write(str(states[i][j]) + " ")
        f.write("\n")
        for j in range(len(states[i])):
          f.write(str(tmp[j]) + " ")
        f.write("\n")

  #This method returns the list of states, which is used to test
    programs, by having them play from many different states.
  def getStates(self):
    return self.listOfStates
```

Listing 35: This is the code in the file *MakeData.py*.

```
import numpy as np
import random
from scipy.special import expit

class mlp:
  def __init__(self, nhidden, stateLen):
    self.beta = 1
    self.eta = 0.1
    self.momentum = 0.0
    self.matrix1 = []
    self.matrix2 = []
    self.inputAmount = stateLen
```

```
    self.outputAmount = stateLen
    self.hiddenAmount = nhidden
    self.outputA = [0.0] * (self.outputAmount)
    self.outputList = [0.0] * (self.outputAmount)
    self.hiddenU = [0.0] * (nhidden)
    self.hiddenList = [0.0] * (nhidden)
    for i in range(self.inputAmount + 1):
      self.matrix1.append([])
      for j in range(nhidden):
        self.matrix1[i].append(random.random())
    for i in range(nhidden + 1):
      self.matrix2.append([])
      for j in range(self.outputAmount):
        self.matrix2[i].append(random.random())

  def sigmoidDerived(self, x):
    return expit(x)*(1-expit(x))

  def identityFunction(self, x):
    return x

  def identityDerived(self, x):
    return 1

  #this function calculates the sum of squares error of all the
    valid inputs and targets, and sums them, and returns that for
    the earlystopping to use for figuring out when to stop.
  def totalError(self, inputs, targets):
    accumError = 0.0
    for i in range(len(inputs)):
      accum = 0.0
      self.forward(inputs[i])
      for j in range(self.outputAmount):
        accum += (targets[i][j] - self.outputList[j])**2
      accumError += accum/2
    return accumError

  #earlystopping, stops when error increases, or decreases too
    slowly
  def earlystopping(self, inputs, targets, valid, validtargets):
    stopTraining = 0
    lowestError = self.totalError(valid, validtargets)
    while not stopTraining:
      self.train(inputs, targets, 10)
      newError = self.totalError(valid, validtargets)
      if newError > lowestError or lowestError-newError < 0.5:
        stopTraining = 1
      else:
        lowestError = newError

  #train trains the neural net for the inputted amount of
    iterations. One pass through the dataset is an iteration.
  def train(self, inputs, targets, iterations=100):
    for notUsed in range(iterations):
      for ite in range(len(inputs)):
        self.forward(inputs[ite])
        outDelta = []
```

```
66          for i in range(self.outputAmount):
67            outDelta.append((self.outputList[i] - targets[ite][i]) *
      self.identityDerived(self.outputA[i]))
68          hiddenDelta = []
69          for i in range(self.hiddenAmount):
70            accumulator = 0.0
71            for j in range(self.outputAmount):
72              accumulator += outDelta[j] * self.matrix2[i+1][j]
73            hiddenDelta.append(self.sigmoidDerived(self.hiddenU[i]) *
       accumulator)
74          for j in range(self.outputAmount):
75            self.matrix2[0][j] -= self.eta * outDelta[j] * 1
76            for i in range(self.hiddenAmount):
77              self.matrix2[i+1][j] -= self.eta * outDelta[j] * self.
      hiddenList[i]
78          for j in range(self.hiddenAmount):
79            self.matrix1[0][j] -= self.eta * hiddenDelta[j] * 1
80            for i in range(self.inputAmount):
81              self.matrix1[i+1][j] -= self.eta * hiddenDelta[j] *
      inputs[ite][i]

83    #forward passes the inputs through the neural net to get the
        outputs.
84    def forward(self, inputs):
85      for i in range(len(self.hiddenList)):
86        self.hiddenU[i] = 1*self.matrix1[0][i]
87        for j in range(self.inputAmount):
88          self.hiddenU[i] += inputs[j] * self.matrix1[j+1][i]
89        self.hiddenList[i] = expit(self.hiddenU[i])
90      for i in range(len(self.outputList)):
91        self.outputA[i] = 1*self.matrix2[0][i]
92        for j in range(len(self.hiddenList)):
93          self.outputA[i] += self.hiddenList[j]*self.matrix2[j+1][i]
94        self.outputList[i] = self.identityFunction(self.outputA[i])
95
96    """To find the best move, the input is run forward through the
        neural net. The highest output is considered the chosen move.
97    If the chosen output is illegal it is trimmed, to make it legal,
        either making it bigger or smaller. if the index is of a hap of
         0,
98    a removal of 1 in some non-empty heap is chosen."""
99    def makeMove(self, state):
100     self.forward(state)
101     if(state[0] < -1):
102       1/0
103     move = [0,0]
104     move[1] = np.argmax(self.outputList)
105     move[0] = int(self.outputList[move[1]])
106     if(move[0] < 1):
107       move[0] = 1
108     elif(move[0] > state[move[1]]):
109       move[0] = state[move[1]]
110     if(state[move[1]] == 0):
111       for i in range(len(state)):
112         if(state[i] > 0):
113           move[0] = 1
114           move[1] = i
```

```
115     return move
```

Listing 36: This is the code in the file *mlp.py*.

```
1  import numpy as np
2  import random
3  import math
4  import time

6  class SpaceSaveSarsaSpread:

8    #The initialization takes the starting board, and finds the
       amount of states in its untrimmed game-tree, which is called
       self.sum. It also initializes T and Q.
9    def __init__(self, startBoard):
10     self.board = startBoard
11     self.sum = self.getSum()
12     self.board.sort(reverse = True)
13     self.listOfStates = []
14     self.listOfStates.append(self.board[:])
15     self.T = []
16     self.Q = []

18   #This method is for finding the amount of states reachable from
       the start-state, for the trimmed game-tree.
19   def getSum(self):
20     return int(math.factorial(len(self.board)+ self.board[0])/(math
       .factorial(len(self.board))*math.factorial(self.board[0])))

22   #This method is for taking a state and a move in the state, and
       returning the resulting state in the correct order.
23   def newState(self, heapIndex, amount, state):
24     curState = state[:]
25     curAmount = amount
26     for i in range(heapIndex, len(state)-1):
27       dif = curState[i] - curState[i+1]
28       if (dif >= curAmount):
29         curState[i] -= curAmount
30         curAmount = 0
31         break
32       else:
33         curState[i] -= dif
34         curAmount -= dif
35     if (curAmount > 0):
36       curState[len(curState)-1] -= curAmount
37     return curState

39   """This method is for setting up the matrices T and Q, as well as
        creating the list of states. For each state it makes all
       possible moves it can,
40   and for each of those, it adds them to their spot in listOfStates
       . The function newState is used to get the resulting state from
        a given move.
41   the states are added at the end of the list of states in this
       case, unless they are already in the list.
42   All those moves are also added to T, which is the main purpose.
       Then Q is made, by setting all positions in T with 1, to have
       small random values in Q.
```

```
43    The other positions in Q gets the value of negative infinity, -np
        .inf, so that they are always smaller than the expected reward
        of a move.
44    Lastly the method sets up the inputs for the training, and runs
        it as well. Lastly the method takes the time used both by the
        setup and the training."""
45    def setup(self, type = 0):
46      timeStart = time.time()
47      emptyBoard = [0] * len(self.listOfStates[0])
48      for i in range(self.sum):
49        self.T.append([0] * self.sum)
50        for x in range(len(self.listOfStates[i])-1, -1, -1):
51          atEnd = 0
52          curIndex = i+1
53          for y in range(1, self.listOfStates[i][x]+1):
54            newState = self.newState(x, y, self.listOfStates[i])
55            if(atEnd):
56              self.T[i][curIndex] = 1
57              self.listOfStates.append(newState)
58              curIndex += 1
59            else:
60              atEnd = 1
61              for j in range(curIndex, len(self.listOfStates)):
62                if (newState == self.listOfStates[j]):
63                  self.T[i][j] = 1
64                  curIndex = j+1
65                  atEnd = 0
66                  break
67              if (atEnd):
68                self.T[i][len(self.listOfStates)] = 1
69                self.listOfStates.append(newState)
70                curIndex = len(self.listOfStates)
71        self.Q.append([])
72        for j in range(self.sum-1):
73          if(self.T[i][j] == 1):
74            self.Q[i].append(random.random())
75          else:
76            self.Q[i].append(-np.inf)
77        if (self.T[i][self.sum-1] == 1):
78          self.Q[i].append(100.0)
79        else:
80          self.Q[i].append(-np.inf)
81      timeEnd = time.time()
82      setupTime = timeEnd-timeStart
83      if(type == 0):
84        epsilon = 0
85        mu = 0.5
86        gamma = 0.5
87        amountEpisodes = 1000
88        timeStart = time.time()
89        self.train(amountEpisodes, epsilon, mu, gamma)
90        timeEnd = time.time()
91        trainTime = timeEnd-timeStart
92
93        times = []
94        times.append(setupTime)
95        times.append(trainTime)
```

```
96
97          return times
98        return -1
99
100     """ This is the method for making a move, it finds the state(
          which is sorted first) in the list of states, by calculating
          its index. Then the best move is found by use
101     of the function np.argmax. Lastly it identifies the index of the
          heap in which the move is made, and it finds a heap of the same
           size
102     in the input state."""
103     def makeMove(self, state):
104       curState = state[:]
105       curState.sort(reverse = True)
106       nextState = []
107       move = []
108       if (len(state)<len(self.listOfStates[0])):
109         dif = len(self.listOfStates[0])-len(state)
110         for i in range(dif):
111           curState.append(0)
112       goal = 0
113       progress = self.board[0]
114       for i in range(len(self.board)):
115         heaps = len(self.board)- (i+1)
116         for j in range(progress, -1, -1):
117           if(j == curState[i]):
118             break
119           else:
120             if(heaps == 0):
121               goal += j-curState[i]
122               break
123             result = 1
124             for x in range(1, heaps + 1):
125               result *= (j + x)
126             goal += (result)//math.factorial(heaps)
127             progress -= 1
128       nextState = self.listOfStates[np.argmax(self.Q[goal])][:]
129       numToDec = 0
130       for i in range(len(curState)-1, -1, -1):
131         if (curState[i] != nextState[i]):
132           numToDec += curState[i]-nextState[i]
133           newIndex = curState[i]
134       move.append(numToDec)
135       for i in range(len(state)):
136         if(newIndex == state[i]):
137           newIndex = i
138           break
139       move.append(newIndex)
140       return move
141
142     """This method is for training the matrix Q, by using the Sarsa
          algorithm, as well as training not just from the start states,
143     but from all states(except the win-state)"""
144     def train(self, amountEpisodes, epsilon, mu, gamma):
145       curStartState = 0
146       for neverUsed in range(amountEpisodes):
147         curState = curStartState
```

```
148       curAction = self.epsilonGreedyChoice(curState, epsilon)
149       playing = 1
150       while(playing):
151         curReward = self.Q[curState][curAction]
152         nextState = curAction
153         if(nextState == len(self.listOfStates)-1):
154           break
155         nextAction = self.epsilonGreedyChoice(nextState, epsilon)
156         self.Q[curState][curAction] -= mu * (curReward + gamma*self
    .Q[nextState][nextAction] - self.Q[curState][curAction])
157         curState = nextState
158         curAction = nextAction
159         if (nextAction == self.sum-1):
160           playing = 0
161       curStartState += 1
162       if(curStartState >= len(self.listOfStates)-2):
163         curStartState = 0

165   #This is an implementation of the epsilon greedy policy, by the
      use of numpy.argmax when finding the best move, or just random
      choice of the possible moves otherwise.
166   def epsilonGreedyChoice(self, state, epsilon):
167     if(random.random()<epsilon):
168       tmpList = []
169       for i in range(state, self.sum):
170         if(self.T[state][i] == 1):
171           tmpList.append(i)
172       return tmpList[random.randrange(0, len(tmpList))]
173     else:
174       return np.argmax(self.Q[state])

176   #This method returns the list of states, which is used to test
      the program, by having it play from many different states.
177   def getListOfStates(self):
178     return self.listOfStates
```

Listing 37: This is the code in the file *SpaceSaveSarsaSpread.py*.

References

Bouton, Charles L. "Nim, A Game with a Complete Mathematical Theory". In: *Annals of Mathematics* 3.1/4 (1901), pp. 35–39. ISSN: 0003486X. URL: http://www.jstor.org/stable/1967631.

Conway, John H., Elwyn R. Berlekamp, and Richard K. Guy. *Winning Ways for Your Mathematical Plays*. Second. Vol. 1. CRC Press, 2001.

Marsland, Stephen. *Machine learning: an algorithmic perspective*. Second. CRC Press/Taylor & Francis Group, 2015.

The Ferranti NIMROD Digital Computer. URL: https://www.goodeveca.net/nimrod.

List of Figures

List of Tables

Listings