Documentation IDPA

TARGET ORIENTED APPLICATION OF A SLAP ALGORITHM FOR ULTIMATE TIC TAC TOE

Stefan Jampen Damian Moser Guided By - Alejandro González and Barbara Becher Gewerbliche Berufsschule St.Gallen

Stefan Jampen & Damian Moser IDPA Contents

Contents

1 Introduction 3

2 Rules 3 2.1 Rules Tic-Tac-Toe . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3 2.2 Rules UTTT1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

3 Motivation 5

4 SLAP2 algorithm 5 4.1 Alpha Beta Pruning algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5 4.2 Heuristic . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8 4.3 Self Learning . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12 4.4 Runtime . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13 4.4.1 Alpha Beta Pruning algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13 4.4.2 Self Learning algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13 4.4.3 Device Restrictions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14

5 Implementation 14 5.1 Implementation in Dart . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14 5.1.1 Gamecontroller . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14 5.1.2 Game . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15 5.1.3 Website . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15 5.1.4 Heuristic. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16 5.1.5 Alpha Beta Pruning Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16 5.1.6 Selflearning Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17 5.1.7 Final Step . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17 5.2 Website and User Interface . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

6 Progress Evaluation 18 6.1 Expected Result . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18 6.2 Received results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18 6.3 Why we chose Dart . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21

7 Challenges 22 7.1 Worker . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22 7.2 Responsive design with CSS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22

8 Review 22

9 Acronyms 24

References 27

1Ultimate Tic-Tac-Toe 2Self Learning Alpha Beta Pruning

Side 2 of 27

Stefan Jampen & Damian Moser IDPA 1. Introduction

1 Introduction

Algorithms are everywhere. They are a part of our everyday lives without even realizing it. They can be found in general mathematics, or in practical applications such as computers or smartphones. But what exactly is an algorithm? According to the Oxford Dictionary an algorithm is generally speaken "a process or set of rules to be followed in calculations or other problem-solving operations, especially by a computer" (Oxford Dictio- naries, 2018). In our apprenticeship, we both have a lot to do with algorithms. Damian works as a software engineer, where he writes most of the time software. Stefan is an electronics engineer and his specialisation in the fourth year of education is also software. According to that, we both have a pretty good idea of what algorithms are and what they do. We found out, that each of us is fascinated by the concepts and pos- sibilities algorithmic structures provide. So we came up with the idea, to implement an algorithm by ourselves.

We set the goal, to program an algorithm, that can play the game Ultimate Tic-Tac-Toe (UTTT). UTTT is a more complex and strategic version of the ordinary Tic-Tac-Toe. Our approach to solve that problem is a so called Self Learning Alpha Beta Pruning algorithm, also known as SLAP Algorithm. It’s a combination of the already existing Alpha Beta Pruning algorithm (ABP) combined with an own made up extension which is, as the name suggests, self learning. We want to analyse the learning process of the algorithm in more detail and evaluate whether there is really an improvement. This will be the mathematical part of our work To experience our algorithm in action, we also wanted to create a website where everybody can play the game against our SLAP Algorithm. Another intention of our project is to bring the seemingly complex subject in an easy understandable form to the interested reader. We aim to resume our approach of the solution in a comprehensible way. We want to give the reader an insight and a deeper understanding of algorithms, how they work and how they are implemented. With the website, we hope to create an interesting and interactive extension to this paper. In order to meet the requirements to cover two subjects, we decided to write our paper and the website in English.

The first part of our IDPA will be to implement the game itself and the corresponding SLAP algorithm. That includes to find a way to value the states of the game in the most efficient way. This will be the task of the Self Learning algorithm (SL). Only if that part of the SLAP algorithm works fine, the ABP is able to work in our favour. We decided to realise all of that with a modern programming language called Dart from Google. Dart is a well-structured, object-oriented programming language that gives us the flexibility to write client and server-side code. All with one code base and one language. Once everything is set up, we will track the development of the SL algorithm. We are also planning to integrate a function into the website where the user can evolve his own SLAP algorithm and observe the progress themselves.

2 Rules

To understand how to play UTTT, it is required to know the rules of the ordinary Tic-Tac-Toe game. If you already know the rules of the ordinaty Tic-Tac-Toe, you can continue reading paragraph 2.2.

2.1 Rules Tic-Tac-Toe

The game Tic-Tac-Toe consist of a 3-by-3 board. Two players are required while one player represents X and the the other player represents O. One player can start and put his mark anywhere he’d like to. After that, the other player can take his turn and put his mark to a remaining spot. This procedure continues until someone has 3 of his own marks vertically, horizontally or diagonal aligned. It’s possible that no party wins and the match ends undecided.

Side 3 of 27

Figure 1: a random scenario of an ordinary Tic-Tac-Toe field

2.2 Rules UTTT

The board of UTTT is made up from 9 ordinary Tic-Tac-Toe boards, hence there are 81 possible fields. Each small Tic Tac Toe board will be called a local board and the big Tic-Tac-Toe board will be called the global board. A player can start anywhere he would like to on a local board. According to the location he played on a local board, his opponent will be sent to that relative position in the global board. The opponent can now mark a tile on that local board, keeping in mind that he will send the other player to the relative position on the global board. A victory on a local board is the equivalent to a marked tile on the global board. To win the game, one has to win 3 horizontally, vertically or diagonal aligned tiles of the global board. If a local board is won or draw, no more moves are allowed there. In case a player was sent to such a board, he is allowed to make his next turn on every other free local board. It’s possible that the game ends in a draw, because no more legal moves are allowed.

Side 4 of 27

Stefan Jampen & Damian Moser IDPA 2. Rules

Figure 2: a random scenario of an Ultimate Tic-Tac-Toe field

The yellow marked local fields are already won by the yellow party.

3 Motivation

Since the beginning of searching a topic for our IDPA, we knew that we both were interested in a practical project rather than a project work in written form. We both are interested in coding and have to work with software. Because of that, a software project was the most obvious option. Additionally, to create an own software project was for both of us a desirable idea. Damian came up with the idea, to create a kind of self learning algorithm which is able to play UTTT. We both were really interested in a practical implementation of such an algorithm. We both knew that we could learn a lot with a project like this and that the context of the IDPA provides the perfect opportunity.

We also knew, that we could integrate knowledge in our project, which we gained at our mathematics teaching during the last three years. To evaluate and describe certain behaviours of our algorithm, algebra and data analysis provide perfect tools. To cover two subjects in our IDPA, we decided to apply our English knowledge, which we also improved during the last years at the BMS. Because we write all our software in English, we came up with the idea to keep language consistency throughout our whole project, including the website and this documentation.

4 SLAP algorithm

4.1 Alpha Beta Pruning algorithm

Side 5 of 27

Stefan Jampen & Damian Moser IDPA 3. Motivation

Figure 3: A representation of an Alpha Beta Pruning tree. (Wikipedia, 2007)

The ABP algorithm always starts from one initial game state. This initial game state is represented by the top node. Each game state also corresponded to a value. Each game state has a certain number of moves that can be played. For example, when the game starts and the field is empty, there are 81 possible moves that can be played at a search depth of one. All the possible moves are now tried out one after the other. After each move we get a new game state, which also has possible moves again. Even after these moves, we have new game states again. This is how one score after the other is calculated. This can be represented as a tree structure. Each node (whether round or square) symbolizes a game state, represented by a number. The branches show the possible moves. Since evaluating all scores would be too computationally intensive, a search depth is defined. In the figure the search depth would be 4, because four turns were evaluated from the initial node. Now the scores of the lowest row (hands) are evaluated. The bigger the number, the better the game state for us. And the better the game state, the sooner we will win.

Now the tree is evaluated so that in the worst case, the score is as high as possible for us. That means as much as assuming that the opponent always makes the move that would be worst for us, because a bad game state for us is a good game state for the opponent. This is how the evaluation starts at the bottom of the tree. The fourth turn (branches between the last two levels) is played by the opponent (MIN stands for minimizing player). Therefore, the smallest number of leaves is always written in the upper node.

Figure 4: Nodes at the bottom left; the smaller value gets written to the upper node

The parent node is given the value five, because the minimizing player always chooses the smaller value. This happens with all nodes of this level. The third move is now played by us. We’ll take the highest possible score, of course. In this way, the higher-level nodes of the previously filled level are filled with the highest of

Side 6 of 27

Stefan Jampen & Damian Moser IDPA 4. SLAP algorithm

Stefan Jampen & Damian Moser IDPA 4. SLAP algorithm

the possible values. Now it’s the turn of the opponent who chooses the worst score for us. The procedure repeats itself up to the initial node. In this way we get the information, which move is best for us. The grey marked nodes and crossed out branches are optimizations of the ABP algorithm. So we do not have to evaluate these areas at all and can save computing time. The search tree gets evaluated after every new turn. In other words, the ABP algorithm is able to calculate n turns, which represents the search depth, in advance. Of course, the bigger the search depth n is, the more accurate our moves will turn out. But the deeper the search depth n is, the longer it takes to calculate all moves.

Side 7 of 27

Figure 5: All possible game moves in the 3rd level of the search depth, evaluated by the minimzing player

The average amount of options per move in the game UTTT is five. (Tsurel & Lifshitz, 2013) In our case, for simplification purposes, the ABP search tree can be pictured with five branches emerging from every new node. With a search depth of n, and without the optimizations of the ABP algorithm provides, the calculated amounts of all nodes can be calculated with the following equation:

a =

∑ni=1

δi

δi

a: amount of nodes, n: search depth, i: index, δ: average move per node

4.2 Heuristic

The ABP algorithm only decides which move is played. But without the heuristic, the ABP algorithm would be pointless. The task of the heuristic is to fill the nodes in the ABP search tree with meaningful numbers. The more accurate our heuristic will be, the better are the chances for our SLAP algorithm to win. The heuristic has only the purpose to evaluate the score of the board, or in other words, calculate a number for each possible game state3. The better the game state, the higher the score.

Now to how the evaluation of the game state works. We award points for certain conditions4 on the board. There are exactly five of these conditions occurring on a board, which are important for our heuristic. The check always takes place in a row, a column or a diagonal. The first two conditions are checked in the small fields.

//TODO akronyme momentaner standpunkt

• One field occupied, two fields empty. The condition occurs three times here. Once in the right column, once in the bottom row and once in the diagonal from top left to bottom right.

Stefan Jampen & Damian Moser IDPA 4. SLAP algorithm

Figure 6: Condition one

• Two fields occupied, one field empty. Here the condition occurs in the lowest row. Note how the first con- dition occurs three times again, in the left column vertically, diagonally left bottom to the right bottom and again vertically middle column.

3A game state is any possible configuration on the UTTT field. 4In our context, a condition is a heuristic parameter. In our heuristic, exactly five conditions exists

Side 8 of 27

Figure 7: Condition two

The following three conditions are checked in the large field.

• One small field won, two small fields not yet determined. Here the condition occurs once in the top row. In this figure, the first two conditions are no longer considered.

Stefan Jampen & Damian Moser IDPA 4. SLAP algorithm

Figure 8: Condition three

• Two small fields won, one small field not yet determined. Here is a visualization of this condition.

Figure 9: Condition four

• Three small squares won. This also corresponds to a victory. Here the condition occurs in the right col- umn.

Side 9 of 27

Stefan Jampen & Damian Moser IDPA 4. SLAP algorithm

Figure 10: Condition five

In the following figure, all five occurring states are marked with different colors: State one Yellow State two Green State three Blue State four Purple State five Light blue

Side 10 of 27

Figure 11: All Conditions

The states are determined separately for both players. We take the difference of the states of the two players. That is to get a more objective result of the five game conditions. If the difference is > 0, the first player has more occurring conditions than the second player. If the difference is equal to 0, both player have the exact same evaluation of the corresponding condition. If the difference is < 0, the second player has an advantage with the corresponding condition. After the subtraction we have five certain numbers, which describes each of the five conditions. Now, we have to convert them into a meaningful number. This is where the DNA comes into play. The DNA is an object consisting of 5 fields. Each field stands for a factor by which the number, that represents conditions, is multiplied. Not all conditions should contribute the same value to the final game state evaluation. With the DNA, we are able to control the relative proportions of the five conditions. Let’s do a sample calculation using the picture above. Our DNA has the following Numbers.

Factor condition one 1 Factor condition two 3 Factor condition three 10 Factor condition four 30 Factor state five 100

In other words, the 5th DNA Value contributes 100 times more value to the final game state evaluation than the 1st DNA factor. The 4th parameter is 10 times more valuable than the 2nd parameter. In the following table, the whole calculation of the value of the game state is shown.

Side 11 of 27

Stefan Jampen & Damian Moser IDPA 4. SLAP algorithm

Stefan Jampen & Damian Moser IDPA 4. SLAP algorithm

Condition Occurrence

Condition red

Occurrence Condition blue

Difference DNA factor Condition

Score 1 0 11 -11 1 -11 2 1 3 -2 3 -6 3 4 0 4 10 40 4 2 0 2 30 60 5 1 0 1 100 100 Game state score 183

The effective game state score in this case is 183, which is all the heuristic does. Based on this infor- mation, the Alpha Beta Pruning decides which move to play.

4.3 Self Learning

To say it in easy terms, the self learning part learns how to estimate the game state in the most meaningful way. It learns which heuristic parameters are important in relation to the other heuristic parameters. In our case, the DNA is responsible for that. The DNA dictates, which heuristic parameters are more important and it does that simply with a factor. The DNA is also the only thing which changes, while learning. Let’s take a look at how an algorithm is generated and how it gets better over time. To start a new era, we need to set a few things up. An era can be compared to several generations of humans. The era contains all generations. In a human generation, we have several individuals. In our case, we will call our individuals of the generation Organisms. We have to decide, how many Organisms per generation we want to create. But we have to keep in mind, the more Organisms we create, the longer the evolution will take. That’s because every Organisms will play against every other Organisms twice. They play two times, to make sure every Organisms has once the possibility to start. It’s crucial, that every Organism has once the first turn, because from a statistical perspective, the player with the first move wins in 56.17% of the cases. That is a difference of 12.34% to the second player, which is a statistically significant advantage. (Tsurel & Lifshitz, 2013) The process of playing against each other is called selection. The Organisms get rewarded with 3 points when they win, and get 1 point when they end in a draw. When they loose, they won’t get any points. The whole process of finding the best Organisms of a generation is actually very similar to a football cup. The amount of games that are played in that way with n participants is n2 − n. We also need to decide the search depth of the alpha beta pruning algorithm the organisms use, while playing against each other. The bigger the search depth, the longer it will take, because the computer has to calculate more scenarios. But the deeper we search, the more accurate results we will get. Now after we set up the amount of organisms and the search depth, we can start our new era. The first generation gets created (initialised) with random DNA. That means the proportion of the different parameters vary vividly. We already know, the Organisms with the best random parameter configuration will perform the best. But we don’t know yet exactly, which parameters are the most important ones. To find it out, which parameters are the best ones, we now let the organisms play against each other. In other words, we will start the selection. After all matches are finished, we can rank the organisms according to the point distribution system, which we used. The worse half of our first generation will die. And as it is in real evolution, mutation will take place. The DNA of the better half of the ranking will mutate two times and form a new generation. The best Organism of the generation will survive and continue to live in the next generation. As described at the beginning, the mutation only affects the DNA. The DNA contains the factor for each parameter and each factor will be mutated by a value between 0.8 and 1.2. This results in an almost totally new set of Organisms, which are descenders of the previous generation. The selection starts again for this generation. This cycle will continue as long as the creator of the era desires to.

Side 12 of 27

Figure 12: visualisation of an evolutionary algorithm

The era which is shown on our website contains 70 generations and each generation consists of 16 Organisms. It took a Computer with 3.2GHz and 16 threads 8 hours to calculate everything. 14 of 16 threads were utilized. With the following equation, we can calculate the time it took for the computer to play one game:

[t] = 8h · 60 minh · 60 smin

162 − 16 · 70 = 1.714s

The average duration of a game between two Organisms with above-mentioned computer is 1.714 seconds.

4.4 Runtime

Our two algorithms both depend on different parameters. These influence the runtime in different ways. If they are too small, we do not get an optimal result; if they are too large, the runtime can be too long.

4.4.1 Alpha Beta Pruning algorithm

The Alpha Beta Pruning algorithm has several factors that influence it. One factor is the average search depth, here δ. This factor is always similar for our game, but the more advanced the score, the smaller it gets. The second factor is the search depth, here n. The deeper you search, the more scores you have to evaluate. This factor influences the score exponentially. Therefore, the search depth of the era must also be carefully selected when creating. So we have a runtime of δn which in the O-notation, which is used in computer science, corresponds to a runtime of O(N m). This means that the parameter n influences the runtime exponentially.

4.4.2 Self Learning algorithm

The self-learning algorithm has two parameters. One parameter is the number of organisms per generation, g for short. Since all play against each other, with outward and return play, formula for the number of games is g2 − g. The second parameter is the search depth with which the organisms play against each other. Now we run the Alpha Beta Pruning Algorithm once for each move per game with our search depth n. Since we don’t know the average number of moves per game, we use the variable ω. And we already know the runtime of the Alpha Beta Pruning algorithm. So in the end we have a runtime of (g2 − g) ∗ ω ∗ deltan. The first part (g2 − g) ∗ ω, which is influenced by the parameter g, has a runtime of O(N2). The second part, which represents the Alpha Beta Pruning algorithm, is O(N k) as shown above. This means that the size of the generation affects the runtime quadratically, but the search depth exponentially. So the search depth should be played around more cautiously, as with the size of the generations.

Side 13 of 27

Stefan Jampen & Damian Moser IDPA 4. SLAP algorithm

Stefan Jampen & Damian Moser IDPA 5. Implementation

4.4.3 Device Restrictions

We have made our program as device friendly as possible. So you don’t need a powerful computer to play games, a smartphone is enough. To train the generations, on the other hand, as much computing power as possible is needed. So you will get better results with a multicore PC than with a smartphone. Since the software runs on as many threads as possible, this is the main criterion of the training. So it was also surprising that a game on my smartphone, the Galaxy S8, was played faster than on my business PC. As we already found out in subsection 4.3, on a computer with 16 threads we needed an average of [t]=1,714s. If we multiply that by the 14 (maximale number threads − 2) working threads, we get [t] = 24s. If we now calculate the time for the smartphone we get the following result. [t] = 1.5min·60 smin (62−6) = 3s. If we multiply this with the 6 working threads we get [t]=3s · 6 = 18s. This means that the smartphone actually works faster than the computer, but would still be slower because the computer has 8 threads more to work with. So you can’t generally say that a device is unsuitable for training organisms. However, the more threads and speed a device has, the faster the organisms can be trained.

5 Implementation

The implementation had two main parts. One part was the implementation of the algorithms and the corresponding one in Dart. The second part was the subsequent integration into the website and the clean design of this.

5.1 Implementation in Dart

The implementation in Dart had to be carefully planned. So we started to create the classes that store the data, like the score, for us. This step had to be carefully considered, we then built all the other classes on these classes. Afterwards, of course, we couldn’t immediately deal with the algorithms and the actual meaning of the project, but had some milestones, which we could only work through one after the other, because they depended on each other.

5.1.1 Gamecontroller

We already have the classes that store data for us, called modal for short. But to be able to play we need a lot more information. For example, which moves can be played when a certain game state is active. Therefore we created the GameController, which gives us exactly such information.

// Returns whether the game is finished or not based on the [state] bool isGameFinished(GameState state) {

// Implementation of IsGameFinished ... 5 }// Return a list of all possible playable moves based on the [state]

List<Move> getAllPossibleMoves(GameState state, [State s]) {

// Implementation of getAllPossibleMoves 10 ...

}// Reverts a played move on the [state] based on the [revertMove] GameState revertMove(GameState state, RevertMove revertMove) { 15 //Implementation of revertMove

... }

Side 14 of 27

Stefan Jampen & Damian Moser IDPA 5. Implementation

// Modifies the [state] based on the [move]. 20 RevertMove playMove(GameState state, Move move) {

//Implementation of playMove ... }

Listing 1: Structure of the GameController

Since these functions are needed for each individual move, we have made these functions as efficient as possible. So we have already evaluated every possible Tic Tac Toe field and stored it in a cache. If we now evaluate the Ultimate Tic Tac Toe field, we can access this information and minimize the computing time per call.

5.1.2 Game

Now that we have all the necessary information for a gamestate, we could start implementing the game. Therefore we created classes for the game itself and for the players. Since the whole program doesn’t yet run in the browser but in the console, we didn’t implement any input options yet. Instead we created the simplest algorithm there is. It chooses a random playable move and plays it. Then we could let the computer play the first games. The games didn’t really make sense, because only one random move was played at a time, but we reached this milestone.

class RandomMove extends Algorithm {

Random r;

/// Initialises a new Random Algorithm 5 RandomMove() {

r = Random(); }@override 10 Move getNextMove(GameState state) {

List<Move> moves = getAllPossibleMoves(state, State.flip(state.lastMove.state)); int randomIndex = r.nextInt(moves.length); return moves[randomIndex]; } 15 }

Listing 2: Implementation of the RandomMove

5.1.3 Website

Now that we have a working game, we want to play it ourselves. That’s why we started developing the website. First we had to make sure that the score was displayed correctly. Then we had to create the possibility that when we click into the grid, the correct move is played. After this implementation we were able to play against the computer, but only against the random algorithm. The further implementation of the website is described in more detail in the subsection 5.2. For this milestone we only needed a playable version so that we could start implementing the Alpha Beta pruning algorithm.

Side 15 of 27

Stefan Jampen & Damian Moser IDPA 5. Implementation

5.1.4 Heuristic

Before we can start with the Alpha Beta Pruning algorithm, we have to implement the heuristics. The heuristic is used by the Alpha Beta Pruning algorithm. For the heuristic we first create the DNA, the base of the evaluation. Based on this DNA, the states are then added together. In order to accelerate this evaluation, we have already loaded the evaluation for all possible Tic Tac Toe fields in the background, as with the game controller. With this information we can efficiently evaluate even a large field without much computational effort. Finally, this function is called several times with each move of the algorithm.

// Returns the evaluated score of the [state] from the // point of view of the [primaryState] double evaluateState(GameState state, State primaryState) {

// Evaluates the score for the [primaryState] and 5 // subtract score of the opponent

return evaluateForState(state, primaryState)

- evaluateForState(state, State.flip(primaryState)); }

10 // Returns the evaluated score of the [state] only for the [primaryState]

double evaluateForState(GameState state, State primaryState) {

double value = 0.0; // Checks if the game is finished i f (cache[state.value][primaryState].three > 0) { 15 // Because the game is finished, only the last parameter of the DNA is relevant

value += cache[state.value][primaryState].three \* dna.bigThree; } else {

// The game is not finished, therefore we analyse all local games state.tiles.map((b) => cache[b.value][primaryState]).forEach((info) { 20 // If the local board is finished, we don't evaluate the score, because

// the score will get added in the last step, where we evaluate // condition 3 and 4

i f (info.three == 0) {

// Adds the score of condition 1 and 2 for each local board 25 value += info.one \* dna.smallOne; value += info.two \* dna.smallTwo; } }); // Finally we evaluate condition 3 and 4 30 value += cache[state.value][primaryState].one \* dna.bigOne; value += cache[state.value][primaryState].two \* dna.bigTwo; }return value; }

Listing 3: Implementation of the evaluation

5.1.5 Alpha Beta Pruning Algorithm

Now we can implement the Alpha Beta Pruning algorithm. Since we already prepared everything, from the heuristics to the game controller, this work was done relatively fast. Since we also built our program cleanly with interfaces, we could simply replace the random algorithm with the Alpha Beta Pruning algorithm and play against it. However, the prerequisite for this was that the heuristic was filled with a meaningful DNA

Side 16 of 27

Stefan Jampen & Damian Moser IDPA 5. Implementation

object. We created this by hand, with values that we thought made sense. Afterwards we were able to play against our algorithm. What is striking is that it was really hard to win against the algorithm, although the parameters were not defined by the computer but by us. That was of course a good sign that our approach could be successful.

5.1.6 Selflearning Algorithm

Since our algorithm now works, we start with the self-learning part. At first we didn’t implement it on the website, but for the console. First we had to create the object that stores the whole process, the era. Then we had to implement the functions described in the subsection 4.3 and call them in the correct order. Then we just had to be patient to train a few generations to see if there was any improvement.

5.1.7 Final Step

Our main goal was achieved. The algorithm evolves itself and we can play against it. But we didn’t want to be satisfied with that yet. That’s why we added the self-learning algorithm to the website and added some nice features like color themes or compatibility on all devices. We also kept an eye on the performance and if possible, wrote the code as efficiently as possible.

5.2 Website and User Interface

In order that our project is accessible all over the world with different end devices ranging from Smartphones to Personal Computers running with different Operating Systems, we thought that it would be the most convenient solution to make a website. In general, the static part of the website is made with HTML (Hyper Text Markup Language) and CSS (Cascaded Style Sheets). HTML specifies the structure of a website. CSS is then required to style everything. The dynamic part was already discussed in the previous subsection. Because CSS can be extremly time consuming, we made use of a CSS Framework called Materialize from Google, which provides cross browser compatible and responsible components such as navbars, buttons, dropdown-lists, cards, modals and many more. To make our website more appealing and fluid, we used CSS animations. For this purpose, we used the website Animista, which provides ready-to-use CSS animations. To deploy our website online, we also needed to have a hosting service which offers a server and a domain. Because we already worked with tools of Google like the language Dart and the Materialize Framework, it wasn’t far-fetched, that we decided to make use of a hosting provider which also belongs to Google. It’s called Firebase. Firebase provides many options for developers such as Real Time Databases or Cloud Messag- ing, but we only needed to use the hosting opportunity, which supports hosting static files like CSS and HTML.

Our website consists of two main parts. The first part, we called it "Reduced View", is mainly designed for an easy game experience, where you can choose between three levels to play against the algorithm. The levels include "easy", "medium" and "hard". Behind the easy level plays the best organism of the first generation. The medium level is played by the best organism of the 35th generation and the hard level is the best organism of the whole era. The default search depth of the current game is three. Three is an experiential value, which is the best compromise between search duration and reliability of the algorithm. The first part of the website contains an explanation of the rules of the game as well. Via a button in the navigation bar, it’s possible to change to a more detailed part of the website, we called it "Advanced View", which we designed for people who are interested in what’s going on behind the scenes. It’s possible to see the whole era with all generations and organisms and it’s evident, how the algorithm has evolved. It’s also possible to generate a new era and calculate everything from scratch. As one plays against a desired Organism, it’s also possible to adjust the search depth of the current game dynamically. It’s observable, that it takes much longer for the algorithm to make a turn if the search depth is deeper.

Side 17 of 27

Stefan Jampen & Damian Moser IDPA 6. Progress Evaluation

Our Algorithm is separated from the part of the Web, that it is modular and reusable for any other applications like an app or something similar. In programming terms, things are separated with a thing called "Interface". Our Interface in dart is called the Player Interfacer. The Player Interface simply provides the information of our game state and is able to get input from the user interface on our website. In HTML and CSS, we prepared a 9x9 grid, which we had to fill dynamically with the information of our Player Interface. When the algorithm made his turn, we also implementet a function that shows the player, where he has to play his next turn. We also had ho make sure, that the calculations of the algorithm didn’t run on the same thread as the UI thread of the website. Otherwise, the UI couldn’t be updated at the moment when the user clicked, which would be a bad user experience.

6 Progress Evaluation

As already mentioned, first-generation organisms are produced from randomly generated DNA. Thus, on average, all parameters are still evaluated in approximately the same way. However, this changes over the generations.

6.1 Expected Result

According to instinct we can say that condition 5, a victory of the game, should be rated better than condition 1. We can also say that condition 2 should be rated better than condition 1, because with condition 2 we are closer to winning a local game. Also a won local game, condition 3, is better than a nearly won local game, condition 2. The point is that the conditions are already sorted the way we expect them to be weighted. Condition 1 should have the least weight, condition 5 the most.

6.2 Received results

The first generation consists only of randomly created organisms. Nevertheless, a first development can already be observed. The best organism already shows more or less the expected curve. The following states are better evaluated than the previous ones. This probably also helped him to victory. Also the last organism is the opposite. With the exception of the last parameter, the following states are rated worse than the previous ones.

First of Genration 1 Median of Generation 1 Last of Generation 1

1,500

A NDeulaV1,000 500

0

1 2 3 4 5 Condition Figure 13: The best, an average and the last DNA of Generation 1

Side 18 of 27

Stefan Jampen & Damian Moser IDPA 6. Progress Evaluation

While in the first generation only the first organism more or less reproduced the expected curve, in the 5th gen- eration there are already 11 of 16 organisms with such DNA. The remaining 5 organisms show the curve up to the fourth parameter, but the fifth parameter was mutated badly and the curve goes down again. So we have in only five generations already a first trend which plays itself in and already strongly differs from the generation 1.

First of Genration 5 Median of Generation 5 Last of Generation 5

2,000

1,500

A NDeulaV1,000

500

0

1 2 3 4 5 Condition Figure 14: The best, an average and the last DNA of Generation 5

In the tenth generation there is again a conspicuity. Slowly all organisms start to look the same, but places 3 and 4 stand out. In these organisms, the fourth and fifth parameters are almost identical. Probably because the fourth parameter mutated strongly upwards. Although these organisms do not look like the expected result, they are better than many that look like the expected result. The striking difference to the others is also that the relative difference of conditions 3 and 4 is greater than for the others. Precisely because the fourth parameter has probably mutated strongly upwards. According to this observation, the ratio of the parameters plays an important role.

Side 19 of 27

Stefan Jampen & Damian Moser IDPA 6. Progress Evaluation

First of Genration 10 Third of Generation 10 Fourth of Generation 10

2,000

A NDeulaV1,500 1,000

500

0

1 2 3 4 5 Condition Figure 15: Some conspicuous DNA of Generation 10

From the tenth generation onwards, these conditions are further developed. So the best of the tenth generation reminds rather of a line, whereas the best of the generation 20 reminds rather of a (half-)parabola. This development continues until generation 70, where we have completed the development.

Side 20 of 27

Stefan Jampen & Damian Moser IDPA 6. Progress Evaluation

First of Generation 10 F.o.G. 15 F.o.G. 20 F.o.G. 30 F.o.G. 50 F.o.G. 70

8,000

A NDeulaV6,000 4,000

2,000

0

1 2 3 4 5 Condition Figure 16: First

//TODO Nach State und Condition suchen und alles aufräumen

6.3 Why we chose Dart

At the beginning of every project there is always the question which programming language to use. Since we didn’t know exactly how we wanted to implement the program, we decided on a versatile language. Dart runs in the browser, respectively you can compile it in Javascript, and also runs on the server side and if necessary we could even build the code into an app. The only, just as versatile language and alternative would be Javascript. However, Dart was designed to compete with Javascript. The main goal was to eliminate the errors and inconveniences of Javascript. So Dart offers an excellent asynchronous language support, which is unfortunately missing in Javascript. Also the language is easier to read, because it is data type based and has many small features, which make the code more compact. Also, we both have no experience with dart yet, what we would like to have changed. And in this project we both saw our chance to change that. All these little things let us decide for Dart. The following is a small example of what is meant by the small features that make the code more compact.

// Returns obj2 if obj1 is null, otherwise returns obj1 function method1(obj1, obj2) {

i f (obj1 == null) {

return obj2; 5 } else {

return obj1; } }

10 // Executes the method bar, but only if foo is not null

function method2(foo) {

i f (foo != null) {

Side 21 of 27

Stefan Jampen & Damian Moser IDPA 7. Challenges

foo.bar(); } 15 }

Listing 4: Example in JavaScript

// Same as above but in Dart dynamic method1(dynamic obj1, dynamic obj2) {

return obj1 ?? obj2; }

5

// Same as above but in Dart void method2(Foo foo) {

foo?.bar(); }

Listing 5: Example in Dart

7 Challenges

7.1 Worker

Worker is a feature of web development. You can outsource computational tasks to them so that the main thread, who is responsible for updating the UI, is not under load for too long. Otherwise, the surface is no longer drawn cleanly and the use of the web application no longer feels fluid. Creating such a worker would not have been a problem with JavaScript. But Dart had his pitfalls. Since the programming language is still in development, this feature was not yet implemented cleanly. We still wanted to use it. In general, when we want to create a worker, we have to link a JavaScript file, which will get executed in an other thread. So our plan was to call a JavaScript file, which afterwards executes a Dart file, compiled in JavaScript. The problem was that there is not a official way, how we should call a Dart file out of a JavaScript file. We had to look deep in the documentation of the Dart compilers to find relevant informations to solve this problems. The most important thing is that it works now. Even tough it puzzled us some hours.

7.2 Responsive design with CSS

When designing a website, it’s important to consider, that different end devices are used to visit the website. The screen size and ratio of a normal desktop computer is extremely different from a smartphone. In order to render the website well on all end devices, responsive5 design is required. To meet that requirement, fortunately we had the Materialize CSS frame work. Despite that, it wasn’t always easy to achieve a seamless design on different end devices. While we developed the website, some components were often displaced, when we accessed the unfinished website with different end devices. In the end, even tough it was challenging, we had a well structured responsive design.

8 Review

In retrospect, it was a very interesting project to create. It was very motivating to us, to see the project grow while achieving small goals. Despite the fact that the algorithm works in theory, we had our concerns that

5Erklärung

Side 22 of 27

Stefan Jampen & Damian Moser IDPA 8. Review

it would seamlessly work in a practical implementation. It was a relieve, when we both saw, that the self learning part of the algorithm evolved in accordance to our expectations.

Side 23 of 27

Stefan Jampen & Damian Moser IDPA 9. Acronyms

9 Acronyms

ABP Alpha Beta Pruning condition In our context, a condition is a heuristic parameter. In our heuristic, exactly five conditions exist game state A game state is any possible configuration on the UTTT field. responsive Erklärung SL Self Learning SLAP Self Learning Alpha Beta Pruning UTTT Ultimate Tic-Tac-Toe

Side 24 of 27

Stefan Jampen & Damian Moser IDPA List of Figures

List of Figures

1 a random scenario of an ordinary Tic-Tac-Toe field . . . . . . . . . . . . . . . . . . . . . . . . 4 2 a random scenario of an Ultimate Tic-Tac-Toe field . . . . . . . . . . . . . . . . . . . . . . . . 5 3 A representation of an Alpha Beta Pruning tree. (Wikipedia, 2007) . . . . . . . . . . . . . . . 6 4 Nodes at the bottom left; the smaller value gets written to the upper node . . . . . . . . . . . 6 5 All possible game moves in the 3rd level of the search depth, evaluated by the minimzing player 8 6 Condition one . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8 7 Condition two . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9 8 Condition three . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9 9 Condition four . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9 10 Condition five . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10 11 All Conditions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11 12 visualisation of an evolutionary algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13 13 The best, an average and the last DNA of Generation 1 . . . . . . . . . . . . . . . . . . . . . 18 14 The best, an average and the last DNA of Generation 5 . . . . . . . . . . . . . . . . . . . . . 19 15 Some conspicuous DNA of Generation 10 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20 16 First . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21

Side 25 of 27

Stefan Jampen & Damian Moser IDPA List of Tables

List of Tables

Side 26 of 27

Stefan Jampen & Damian Moser IDPA References

References

Oxford Dictionaries. (2018, jan). https://en.oxforddictionaries.com/definition/algorithm. ([Online; accessed

23-January-2019]) Tsurel, D., & Lifshitz, E. (2013). Ai approaches to ultimate tic-tac-toe. Hebrew University of Jerusalem. Wikipedia. (2007, Mai 28). Alpha beta pruning tree. https://upload.wikimedia.org/wikipedia/commons/9/91/AB\_pruning.svg.

([Online; accessed 25-January-2019])

Side 27 of 27