REPORT GAME AI

Anna-Lena Popkes

University of Bonn

1. INTRODUCTION

In the last decade, computer games have become an enormous business and now represent one of the fastest growing sectors in several economies, e.g. in the U.S. [1, 2, 3]. This has to do not only with new developments in fields like computer graphics and networking, but also with the increasing application of pattern recognition and machine learning techniques in game development [4].

The lecture on Game AI (artificial intelligence) was about such computational intelligence methods and their application in computer games. In this context, we focused on two topics: the application of classical artificial intelligence techniques in games and the use of more modern machine learning approaches. During the semester we worked on three projects, including tasks related to various applications of AI techniques in computer games.

In the following, we will present our results for the different tasks. First, we will introduce the minmax algorithm, as it is being used in several of our solutions. Afterwards, we will talk about different strategies for simple turn-based games, using the examples of TIC TAC TOE and CONNECT FOUR. We will outline advantages and problems of the various approaches and will give a solution that leads to an (almost) infallible player. We will also talk about the topic of path planning, describing the functioning of the popular *$A*$ algorithm* as well as *Dijkstra's algorithm*. In the last part, we will consider the topics of *fuzzy logic* and *fuzzy control*, *self organizing maps* and *bayesian imitation learning*.

2. MINMAX ALGORITHM

In order to make optimal decisions in a game, a player needs some kind of strategy. A popular algorithm for finding such strategies in two-player turn-based games is the minmax algorithm.

In simple terms, finding an optimal strategy in a two-player turn-based game corresponds to finding a path from the root node of the game tree to a goal state that maximizes the players utility [5].[1] In the minmax algorithm the player who moves first is called MAX and its opponent MIN. An optimal strategy should specify MAX's initial move, moves in states resulting from possible responses by MIN, moves in states resulting from possible responses to possible responses by MIN and so on. This leads to a strategy that is *at least as good* as any other strategy when playing against an infallible opponent. Such a strategy can be found by determining the so called *minmax value* (mmv) of every node, representing the utility for MAX of being in the corresponding state (assuming that both players play optimally). The minmax value of a terminal state is its utility. Furthermore, when MAX moves he (naturally) wants to maximize his payoff and will therefore move to a state of *maximum* value. The opponent (MIN), however, wants to win as well and will therefore move to a state of *minimum* value [6]. The minmax algorithm computes the mmv for the current state. To do so, it makes use of the following recursive formula [5]:

$$mmv(n) = \begin{cases} u(n) & \text{if } n \text{ is a terminal node} \\ \max_{s \in Succ(n)} mmv(s) & \text{if } n \text{ is a } MAX \text{ node} \\ \min_{s \in Succ(n)} mmv(s) & \text{if } n \text{ is a } MIN \text{ node} \end{cases}$$

When employing the minmax algorithm, a complete depth-first exploration of the game tree will be performed [6]. After computation, player MAX can act according to the optimal strategy by always choosing the move with the highest minmax value. However, it is often infeasible to compute the mmv of every state throughout a game. For a maximum depth m and b legal moves at each point, the time complexity of the algorithm is $O(b^m)$. Therefore, it is often combined with pruning or used as a depth-restricted version [5].

An example of computed minmax values is illustrated in figure 1. At first view, it might seem better to store not only the mmv for every node but also the average score of the nodes at the next level. This could help determine "better alternatives" in case of ties [5]. However, when assuming that the opponent (MIN) always chooses the state of minimum value, storing another value becomes pointless.

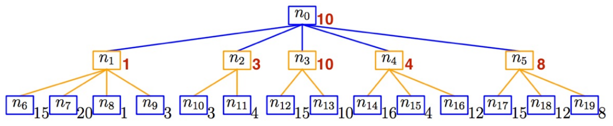


Fig. 1: Minmax computations with computed minmax values for nodes n_0 to n_5

[1]The utility assigns a numerical value to terminal states of a game, for example $+1$ for a win, -1 for a loss, and 0 for a draw [5].

3. TIC TAC TOE

A main part of the first project was dedicated to the game TIC TAC TOE. TIC TAC TOE is a simple turn-based game between two players (player X and O) on a 3×3 grid. The goal of each player is to place three of its symbols next to each other, either in a row, column or diagonal. When both players move at random, the player who starts (X in our case) will win more often as he is always one step ahead of the other player (see figure 2a). This can be further underlined when examining the TIC TAC TOE game tree, as done in project two.

Before building the tree, an upper bound on the total number of nodes can be computed by the following formula: $\sum_{i=0}^{9} \frac{9!}{(9-i)!}$ which results in a total of 986410 nodes. However, this upper bound incorrectly includes successor states of game states in which one player has already won. Therefore, when building the actual game tree, the number of nodes is much smaller (549946 nodes, average branching factor of 1.86).[2] Of the 255168 leaf nodes, X wins in 131184 of the cases, O only in 77904. So when considering the number of times the game ends in a win, the probability that X won the game is 62.74%, the probability that O won only 37.26%. We can further increase the success rate of X by means of various strategies. In the following, we will look at two of them: a simple probabilistic and a heuristic strategy.

3.1. Probabilistic strategy

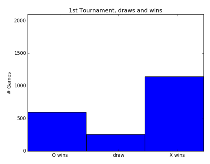
A very easy way to improve X's play is to collect a statistic about fortunate positions on the board. For this, we counted how often the individual cells contribute to a win in 2000 games between two random players. The resulting numbers can be normalized and used as a future probabilistic strategy. To be more precise: if it is X's turn, he evaluates all empty positions on the board and places his symbol on the position with the highest probability. Player O keeps picking positions at random.

When examining the resulting probabilities we found that the middle cell contributes to a win most often, followed by the corner cells. This is quite intuitive, as the middle cell provides the current player with the possibility to win on either of the diagonals, the middle column or the middle row. As illustrated in figure 2b, applying such a probabilistic strategy increases the winning rate of X. However, it does not completely eliminate the possibility of O winning.

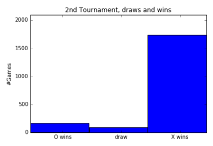
3.2. Heuristic strategy

When aiming at constructing an infallible player, a more sophisticated strategy needs to be developed. This can be done in several ways. In this work, we focused on an adjustment of the minmax algorithm introduced in section 2.

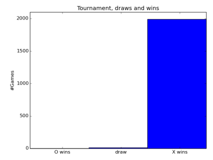
[2]This corresponds to a game tree without considering symmetries.



(a) Random players



(b) Probabilistic strategy



(c) Heuristic strategy

Fig. 2: Distribution of wins and draws over 2000 TIC TAC TOE games

When transferring the minmax algorithm to TIC TAC TOE we have to take into account that only player X uses a strategy. Its opponent O, however, moves at random. Therefore, we cannot assume that O always chooses the move with the minimal score. As a consequence, we adapted the minmax algorithm such that on O's turn we chose the *average* of the nodes scores. To reduce the number of nodes in the tree and hereby also its learning time, we made sure that the mirrored and rotated variants of a game state were all stored as one node. Furthermore, we did not expand the tree further if X was about to win with its next move (we called this "trivial checks"). The effect of the individual modifications is illustrated in figure 3. When using the algorithm as a heuristic strategy for player X, he comes close to infallibility. As can be seen in figure 2c only a few games end in a draw. In all other games, X wins.

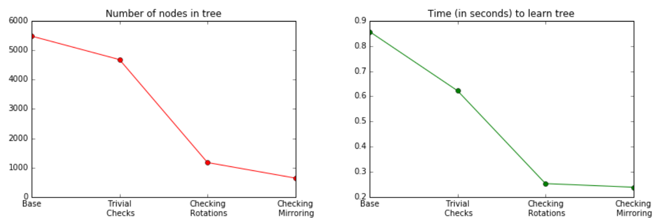


Fig. 3: Optimization results on learning the TIC TAC TOE tree

4. CONNECT FOUR

Like TIC TAC TOE, CONNECT FOUR is a turn-based game between two players. However, the field is larger (6×7 grid) and the players try to place *four* of their men next to each other, either horizontally, vertically or diagonally [7]. Each player has 21 men of a certain color (e.g. blue for player 1, red for player 2). Furthermore, a man placed in one of the columns will fall down to the lowest free cell of that column.

A column is blocked if it contains six men. Figure 4 shows a situation in which player blue has won.

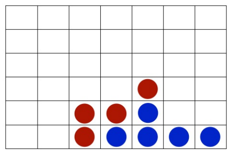
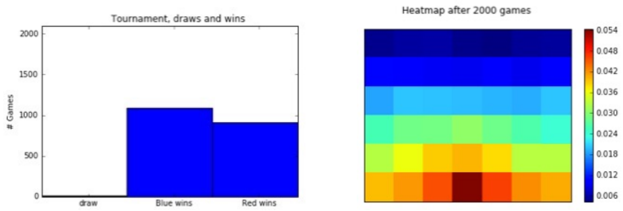


Fig. 4: Player blue has won

4.1. Probabilistic strategy

When running a tournament between two random players, the distribution of wins between them is quite balanced (see figure 5a). Player blue wins slightly more often because he is the one to start. To increase the winning rate of one on the players we could try to collect similar statistics as for TIC TAC TOE (see section 3.1). The result of counting how often the grid cells contributed to a win is illustrated in figure 5b.



(a) Random players (b) Probabilities

Fig. 5: Tournament and statistics of CONNECT FOUR

When taking a closer look at the figure we can see that the highest probabilities occur in the lowermost rows and decrease when going further up. This can be explained by the fact that most games are already won or lost before reaching the upper columns. When using the computed probabilities as a strategy for one of the players (e.g. blue), we soon ran into problems. In many situations, the strategy fails. One of them is illustrated in figure 6.

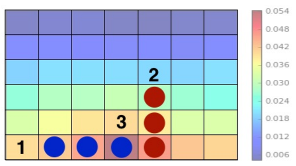
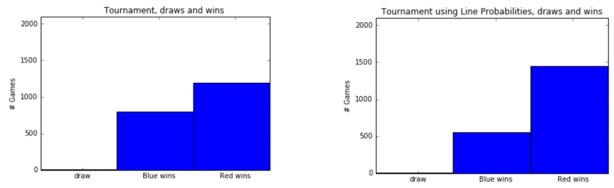


Fig. 6: Connect four situation in which a probabilistic strategy fails

When examining the figure we will assert that the best move for player blue would be to place its man at cell 1 (to win) or at cell 2 (to prevent red from winning). However, the heatmap will tell him to place his man at cell 3. Thus, blue will not only miss his opportunity to win but might lose at the next stage. This does not mean that a probabilistic strategy does not increase the winning probability at all. As can be seen in figure 7a, it does increase the winning rate of the corresponding player. In contrast to figure 5a, not player blue (who starts) is using the probabilistic strategy, but player red. However, there is still a lot of room for improvement. Even when counting how often the individual *lines* contribute to a win this does not change. The result of such a strategy can be seen in figure 7b. As evident in the figure the winning rate further increases but is still far from 100 %. We can conclude that a probabilistic strategy based on simple statistics does not work for CONNECT FOUR. A more advanced strategy is needed.



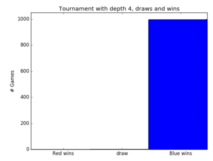
(a) Using cell probabilities (b) Using line probabilities

Fig. 7: Tournament with player red moving probabilisitically and player blue (who starts) at random

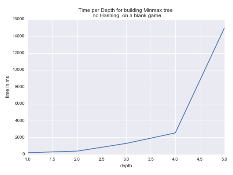
4.2. Heuristic strategy

In order to make more intelligent moves we made use of the minmax algorithm introduced in section 2. As computing the entire game tree for CONNECT FOUR is too complex, we used a depth-restricted version of the algorithm together with an evaluation function that evaluates the merits of non-terminal nodes in the tree. For a given game state, the heuristic function counts the number of two, three and four symbols in a row/column/diagonal of player MAX, weights them positively (with 1000 for four symbols being the highest weight) and sums up the result. From the sum, the number of two, three and four symbols in a row/column/diagonal of player MIN are subtracted, deliberately weighting four in a row by a factor of 10000 (indicating that player MIN wins). So every time player blue has to move, he constructs the game tree for the current state and picks the move with the highest mmv.

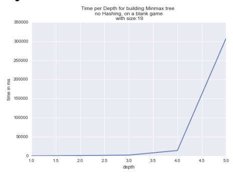
To further improve the running time of our algorithm we used alpha-beta pruning and hashing of the computed scores. As evident in figure 8a this strategy leads to an infallible player. When increasing the depth it takes much longer to learn the tree. A comparison between the learning time for different depths can be seen in figure 8b. When further increasing the size of the board to 19×19 (as used in the game GO), it will take much longer to learn the tree (see Fig. 8c). The distribution of wins and draws, however, stays the same.



(a) Minmax player



(b) Time comparison 6×7



(c) Time comparison 19×19

Fig. 8: Connect 4 using the minmax algorithm

5. PATH PLANNING

The task of finding and planning paths through graphs is a major topic in game AI. For example, when programming a non-player character (NPC) it has to move through its environment on safe and possibly short paths in order to reach goal locations [8]. Several different path planning algorithms exist. In accordance with project two we considered the two most poplaur ones: Dijkstra's algorithm and A*.

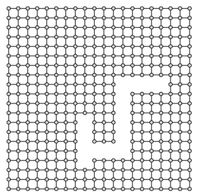
Before being able to find paths we first needed a proper representation of the environment. This is usually done by encoding the topology and geometry of a given map using a graph. In the graph vertices correspond to empty positions, and edges connect all empty positions that are vertically or horizontally adjacent. An example for this kind of representation is illustrated in figure 9a [5].

5.1. Dijkstra's algorithm

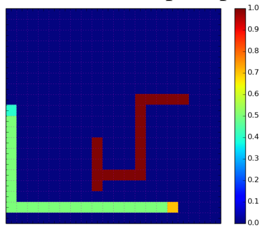
Dijkstra's algorithm is an informed tree search algorithm that finds shortest paths in a connected (directed or undirected) graph with weighted edges. To be more precise, it computes the shortest path from a given source vertex s to every other vertex in the graph [9].

In the beginning, the distance values of all vertices (except for s) are set to infinity, reflecting the fact that none of the nodes has been visited yet. During computation, the algorithm keeps track of all visited nodes by storing them in a set.

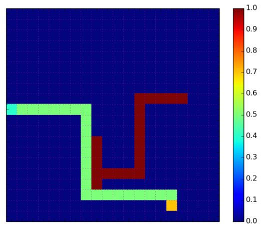
After initializing the distance values, Dijkstra's proceeds in a series of rounds. In each round, it computes the shortest path from the source vertex to *some* new vertex u, namely the one with the minimal distance value. The vertex is added to the set of visited nodes and all neighboring vertices v of u are checked to see whether a better path from s to v via u exists. If all vertices have been visited, the algorithm terminates. It is guaranteed to find a shortest path from the source vertex to some target vertex t, as long as all edges have non-negative costs [9]. To be able to retrieve the path from source s to target t, the general procedure can be extended such that the predecessor of every vertex is stored in addition to its distance value. An example of a path computed by Dijkstra's algorithm is illustrated in figure 9b with the source vertex colored in light blue and the target vertex in yellow.



(a) Graph representation of a game map



(b) Dijkstra's algorithm



(c) A* algorithm

Fig. 9: Path planning using A* and Dijkstra's algorithm

5.2. A* algorithm

Like Dijktra's algorithm, A* is an informed tree search algorithm. It computes the shortest path from a source vertex s to a target vertex t in a connected (directed or undirected) graph with weighted edges. In order to minimize the number of expanded nodes, A* makes use of an *evaluation function* f and always expands the node with the smallest f value. The value of f on a node n is given by $f(n) = g(n) + h(n)$ with $g(n)$ being the cost of an optimal path from s to n, and $h(n)$ being a *heuristic* that estimates the cost of an optimal path from n to the target node t. For path planning, a common choice for $h(n)$ is the Euclidean distance.

During computation, the algorithm maintains an *open set* or *fringe* for the repeated selection of the node with the smallest f value. In each iteration of the main loop, the node with the smallest $f(n)$ value is removed from the fringe and added to a *closed set*. Then, all its neighbors that are not in the closed set are added to the fringe and their g and f values are updated. Like Dijkstra's algorithm, A* can be extended to keep track of the predecessors of every node in order to retrieve the whole path from s to t. If no nodes are left in the fringe, the algorithm terminates. It can be shown that the A* algorithm is both optimal and complete, given that the used heuristic is *admissible* (like e.g. the Euclidean distance) [5]. A heuristic is called admissible if it does not overestimate the cost to reach a goal state [6]. An example of a computed path using A* is illustrated in figure 9c.

6. BREAKOUT

BREAKOUT is a simple computer game developed by Atari in 1975 and inspired by the Atari game PONG, in which the player has to use a ball to knock out bricks from a wall at the top of the screen [10]. BREAKOUT begins with eight rows of blocks, each two rows having a different color denoting the number of points the blocks give. The player coordinates the ball using the walls of the screen and a movable paddle at the bottom. Whenever the ball misses the paddle and hits the ground, the player loses one of his three lives. To make the game more complicated, the speed of the ball increases over time [11].

In order to move the paddle such that it misses the ball as little as possible, we developed different kinds of controllers in projects two and three.

6.1. Reinforcement Learning

The simplest way to control the paddle is to let it follow the x-position of the ball. In order to move the paddle accordingly, we determined the x-position of both the paddle and the ball, computed the way the paddle had to move to be under the ball and set its new position accordingly. To develop a more sophisticated controller, we made use of *reinforcement learning*.

Besides supervised and unsupervised learning, reinforcement learning is a third major machine learning paradigm [12]. A main aspect of the approach is the use of *feedback* in form of positive and negative rewards. The overall goal for a reinforcement-learning agent is to use the obtained rewards in order to learn an optimal policy for the environment without having any prior knowledge about it or the reward function.[3] In simple terms: an agent is placed into an unknown environment and must learn to behave in it successfully, given only positive and negative rewards.

For our controller we used *Q-learning*, a technique that does not require a model of the environment (called a *model-free* method). In Q-learning the agent learns an *action-utility function* that gives the expected reward of taking a certain action a at at certain state s, denoted as $Q(s, a)$ [6]. The update equation is given by:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(R(s) + \gamma \max_{a'} Q(s',a') - Q(s,a))$$

with $R(s)$ being the reward for state s, α being the learning rate and $0 < \gamma < 1$ being the discount parameter, assigning more importance to immediate rewards than to later ones. Our reward function is given by:

$$R(s) = \begin{cases} -10000 & \text{if game lost} \\ \text{current\_score} - \text{distance\_to\_ball} & \text{otherwise} \end{cases}$$

[3]An optimal policy maximizes the expected total reward of the agent.

With the first part discouraging the agent from losing and the second part encouraging it to keep as close to the ball as possible. So in the training phase, our agent learned the action-utility function Q. After training, for every possible state s, it determined the action a with the highest Q value and executed it.

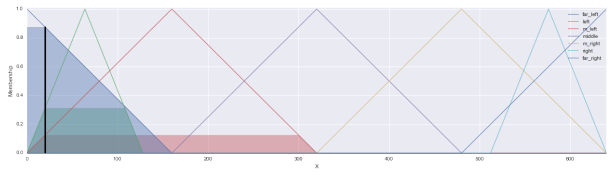
Although the functioning of the first controller is very simple, it works surprisingly well and can cope even with an increasing ball speed. Also the more advanced controller, using Q-learning and the described reward function, works very well. Both controllers start missing the ball only when the ball speed exceeds the paddle speed. When adjusting the controllers such that also the speed of the paddle increases, the episodes last longer. Furthermore, when not increasing the speed of the ball, both controllers win the game.

6.2. Fuzzy controller

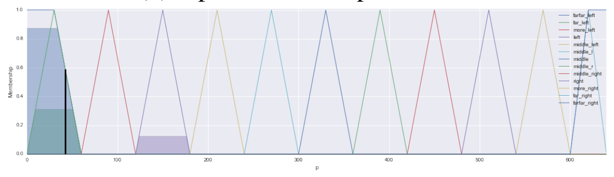
In project three we developed another controller for breakout relying on fuzzy logic/control. The basis for fuzzy logic and control is fuzzy set theory.

Fuzzy set theory is an extension of classical set theory: instead of belonging or not belonging to a set, objects have different *degrees of membership*. This is realized by generalizing the classical indicator functions to so called *membership functions* with values between 0 and 1, representing the degree to which an element X is part of some fuzzy set S. Similar to fuzzy set theory, fuzzy logic is an extension of classical logic, allowing truth values of logical expressions to be any real number between 0 and 1. Fuzzy logic allows us to reason with logical expressions that describe membership in fuzzy sets [5, 6].

In fuzzy control, which we used for our controller, the mapping between real-valued inputs (the ball position in our case) and output parameters (where to move the paddle) is represented by fuzzy rules. To create the controller we first subdivided our crisp measurements of the ball position into seven different fuzzy sets indicating the different x-positions of the ball. With respect to this, we defined seven input membership functions, each covering several of the fuzzy sets (illustrated in Fig. 10a). After formulating the membership functions we defined seven different rules describing where to move the paddle dependent on the x-position of the ball e.g. if the x-position of the ball is in the set far left, move the paddle far far left. During the game, the current x-position of the ball will activate several membership functions and hence several rules. An example is given in figure 10. In part a) the input position of the ball is $x = 20$, resulting in the activation of three membership functions and hence three different rules. In the subsequent "defuzzification step", the different activated rules are mapped to a single number, resulting in a smooth interpolation of the different actions. This smoothness is reflected in the characteristic behavior of the fuzzy controller to sometimes move faster and sometimes more slowly. The output, so the new paddle position, is also divided into fuzzy sets (13 in total), as illustrated in Fig. 10b). The figure also shows to what degree the different output membership functions are activated by the three rules that an input of $x = 20$ triggers. The defuzzification step maps the different activated rules to an output of $x = 40$, denoting the new position of the paddle.



(b) Input membership functions



(b) Example of possible output computation

Fig. 10: Problem illustration for dataset 1 with arrows denoting performed actions

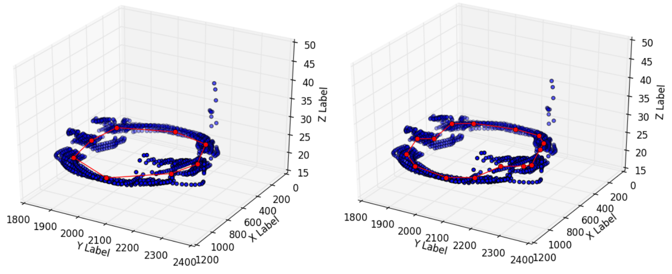
Although our controller is very simple it copes well with the increasing speed of the ball. To further improve the algorithm we could extend our input to two variables. For example, we could also take into account the position of the paddle or the speed of the ball. This should result in an even better controller.

7. SELF ORGANIZING MAPS

A self organizing map (SOM) is a type of artificial neural network that is based on *competitive* learning and trained in an *unsupervised* way [13]. It consists of neurons, connections between them and weight vectors. Typically, the neurons are arranged as a grid and selectively tuned in order to produce a low-dimensional representation of some high-dimensional input data [5]. A self organizing map is hence characterized "by the formation of a topographic map of the input patterns, in which the spatial locations (i.e. coordinates) of the neurons in the lattice are indicative of intrinsic statistical features contained in the input patterns"[13].

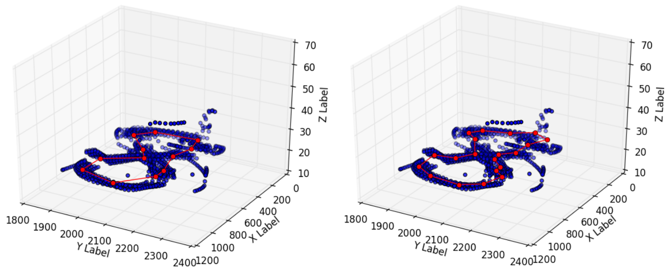
In the competitive learning process, a random input vector x is sampled from the input space and the so called *winner neuron* v_j that best matches x is determined using the following formula: $i \leftarrow \text{argmin}_j ||w_j - x||^2$ where w_j corresponds to the weight vector of neuron j. Afterwards, the weight vectors of *all* neurons are updated according to: $w_j \leftarrow w_j + \eta(t) \cdot e^{\frac{D_{ij}}{2\sigma(t)} \cdot (x - w_j)}$ with D_{ij} being the topological distance between neurons v_j and v_i. The learning rate η and the topological adaptation rate σ are functions of t (time) and chosen such that they decrease over time.

In project three we used the above algorithm to fit self organizing maps to the movements of a human game player. Instead of arranging the neurons in a lattice, we chose the SOM topology to be a circular path graph. The results for two different, given datasets are illustrated in figures 11 and 12.



(a) 8 vertices (b) 15 vertices

Fig. 11: Self organizing map for dataset 1



(a) 12 vertices (b) 20 vertices

Fig. 12: Self organizing map for dataset 2

In the future, to further improve our SOM and make it applicable to a wide variety of input patterns we should change the initialization of the neurons. For the given datasets we started of with a circular structure, not with random points. However, this already formulates an expectation about properties of the data that might not exist in reality.

8. BAYESIAN IMITATION LEARNING

In the last task of project three we used the trajectory data and results from our self organizing maps in order to realize a simple form of bayesian imitation learning. The goal of bayesian imitation learning in the game domain is to reproduce complex movements of human players in a convincing way. This could help substantially in the development of artificial human-like gamebots [14].

The l weights of the SOM fitted in section 7 can be interpreted as representing prototypical game states assumed by the player. In this sense, we have clustered the location data $\boldsymbol{x}_t$ into l states $\boldsymbol{s}_i$. The player's activity $\boldsymbol{a}_t$ at timestep t can be computed as $\boldsymbol{a}_t = \boldsymbol{x}_{t+1} - \boldsymbol{x}_t$. To imitate the trajectory we first computed all activity vectors and clustered them into

k prototypical activities $\boldsymbol{r}_j$ using the k-means algorithm. We further assigned every location vector $\boldsymbol{x}_t$ to the closest state $\boldsymbol{s}_i$ and every action vector $\boldsymbol{a}_t$ to its closest prototype $\boldsymbol{r}_j$. Next, we computed the joint probabilities $p(\boldsymbol{s}_i, \boldsymbol{r}_j)$ that state $\boldsymbol{s}_i$ and action $\boldsymbol{r}_j$ occur together. This translates to constructing a matrix that stores for every SOM state $\boldsymbol{s}_i$ the probability for each activity $\boldsymbol{r}_j$ to occur.

The joint probabilities can be used to generate a new trajectory. For this, we started with a random point $\boldsymbol{x}_t$ from the given data at time step $t = 0$, determined its closest state vector $\boldsymbol{s}_i$ and chose an action according to the formula $\boldsymbol{a}_t = \operatorname{argmax}_{\boldsymbol{r}_j} p(\boldsymbol{r}_j|\boldsymbol{s}_i)$. This action was used to generate a new point $\boldsymbol{x}_{t+1} = \boldsymbol{x}_t + \boldsymbol{a}_t$. Iterating this process several times results a new trajectory, illustrated in figure 14a. As evident in the figure, the new trajectory is not similar to the original one but moves both inwards and downwards like a spiral. When considering the procedure used to generate new points we can see why. For every newly created point $\boldsymbol{x}_t$, the closest state vector $\boldsymbol{s}_i$ and corresponding action $\boldsymbol{a}_t$ are computed. However, the action will be performed at the location of $\boldsymbol{x}_t$, not at $\boldsymbol{s}_i$. Hence, the trajectory will behave in the described way. An illustration of the problem is given in figure 13. The unwanted effect can be prevented by considering not only the action $\boldsymbol{a}_t$ but also the vector from $\boldsymbol{x}_t$ to $\boldsymbol{s}_i$ and computing a new, modified action by combining the two. This approach works very well, producing trajectories like the one in figure 14b.

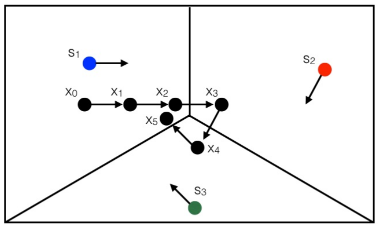
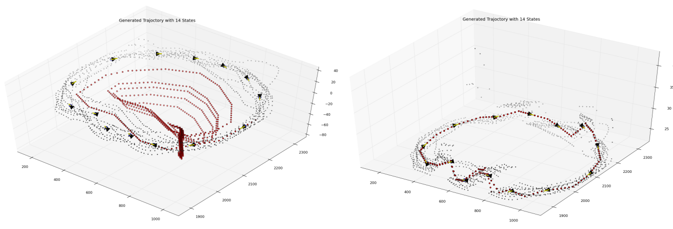


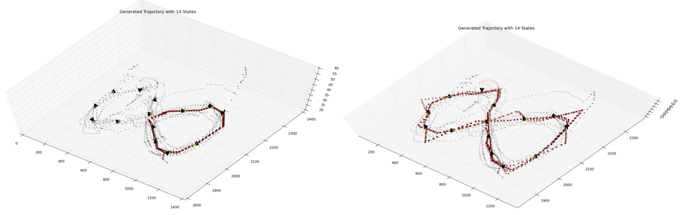
Fig. 13: Problem illustration for dataset 1 with arrows denoting performed actions



(a) Computed trajectory (b) Corrected trajectory

Fig. 14: Computed trajectories for dataset 1

When applying the path generation procedure to the second dataset (see Fig. 12) we ran into a different problem. Although the original trajectory has the shape of a horizontal eight, the computed trajectory stays only on one side of the eight (see Fig. 15a). Why this happens becomes evident when examining the intersection of the two loops. All location vectors near the intersection will be assigned to the state $\boldsymbol{s}_i$ in the middle of the eight (illustrated as a black arrow). Because we will always choose the same action $\boldsymbol{a}_t$ for $\boldsymbol{s}_i$, the trajectory will never cross the intersection and will hence stay on the same side. A possible solution to this could be to take into account how likely the action $\boldsymbol{a}_t$ is to occur after action $\boldsymbol{a}_{t-1}$. After modifying the approach accordingly, we got a trajectory like the one illustrated in figure 15b. However, the success of the modification depended on the initial l SOM-weights used as prototypical game states.



(a) Computed trajectory (b) Corrected trajectory

Fig. 15: Computed trajectories for dataset 2

Another simple solution to the problem would be to enlarge the input to the system. Instead of taking into account the current position only, we could also include the previous position. Alternatively, we could choose not the assigned action, but a random one, based on the probability that each action appears at a certain state. As tested by another group, this also solves the described problem.

9. CONCLUSION

In this report I have presented the different solutions we developed in order to successfully solve all projects. Project one focused on simple strategies for turn-based games, including a probabilistic and heuristic strategy for TIC TAC TOE and a probabilistic strategy for CONNECT FOUR. To create an (almost) infallible player for TIC TAC TOE we made use of the popular minmax algorithm. In CONNECT FOUR, a simple probabilistic approach does not suffice, but will fail to perform sensible moves in several situations.

In project two we further focused on the two mentioned games, building the complete game tree for TIC TAC TOE and finding a more sophisticated strategy for CONNECT FOUR. To develop a strategy towards more intelligent moves we again made use of the minmax algorithm. In combination with an appropriate evaluation function this leads to an infallible player. The project also introduced the game BREAKOUT and asked for the implementation of a controller. We made use of reinforcement learning and succeeded in developing a successful controller, coping well with an increasing ball speed. In a final task, we dealt with the topic of path planning and implemented the well known algorithms A* and Dijkstra's.

In project three we first tested the performance of our CONNECT FOUR strategy on a larger board (19×19) and found that the time requirements increase but the winning rate stays the same. Further, in addition to the reinforcement controller, we implemented another controller for breakout using fuzzy logic and fuzzy control. Although our fuzzy controller is structured in a simple way it works well and can cope even with the increasing ball speed. In this project, we furthermore considered self-organizing maps, a special artificial neural network and used it to model the trajectory of a human game player. As a last task, we used the results from the self organizing maps to realize a simple form of bayesian imitation learning. While trying to reproduce the given trajectories we encountered several problems. One of them, found for the second dataset, shows that the Markov assumption, stating that "the future is independent of the past given the present" [15] does not carry over reliably to real world problems, but rather represents a mathematical convenience. In real world problems it is often not enough to consider the current position and the goal location, but information about previous positions plays a major role.

To sum up ...

10. REFERENCES

[1] Entertainment software association (esa), "Games: Improving the economy," http://www.theesa.com/wp-content/uploads/2014/11/Games_Economy-11-4-14.pdf, November 2014, Accessed: 2016-08-07.

[2] Entertainment software association (esa), "Video games in the 21st century: the 2014 report," http://www.theesa.com/wp-content/uploads/2014/11/VideoGames21stCentury_2014.pdf, November 2014, Accessed: 2016-08-07.

[3] Entertainment software association (esa), "2015 annual report," http://www.theesa.com/wp-content/uploads/2016/04/ESA-Annual-Report-2015-1.pdf, April 2016, Accessed: 2016-08-07.

[4] C. Thurau and C. Bauckhage, "Synthesizing movements for computer game characters," August 2004.

[5] Prof. Dr. Christian Bauckhage, "Game AI," University Lecture, Bonn-Aachen International Center for Information Technology, 2016.

[6] S. J. Russel and P. Norvig, *Artificial Intelligence - A Modern Approach*, Upper Saddle River, NJ : Prentice Hall, 3rd edition, 2010.

[7] Louis Victor Allis, *A knowledge-based approach of connect-four*, Vrije Universiteit, Subfaculteit Wiskunde en Informatica, 1988.

[8] Alexander Nareyek, "AI in computer games," *Queue*, vol. 1, no. 10, pp. 58–65, Feburary 2004.

[9] Steven S. Skiena, *The Algorithm Design Manual*, Springer Publishing Company, Incorporated, 2nd edition, 2008.

[10] Steven L. Kent, *The Ultimate History of Video Games: From Pong to Pokemon-the Story Behind the Craze That Touched Our Lives and Changed the World*, Prima Communications, Inc., Rocklin, CA, USA, 2001, pp. 71.

[11] "Atari breakout," http://ataribreakout.net, Accessed: 2016-07-10.

[12] Richard S. Sutton and Andrew G. Barto, *Introduction to Reinforcement Learning*, MIT Press, Cambridge, MA, USA, 1st edition, 1998.

[13] Simon Haykin, *Neural networks and learning machines*, Pearson Prentice Hall, 3rd edition, 2009.

[14] Christian Thurau, Tobias Paczian, and Christian Bauckhage, "Is bayesian imitation learning the route to believable gamebots?," in *Proceedings GAME-ON North America*, 2005, pp. 3–9.

[15] Kevin P. Murphy, *Machine Learning: A Probabilistic Perspective*, MIT Press, 1st edition, 2012.