Machine Learning: Project 2022-2023

Multi-Agent Learning in Canonical Games and Dots-and-Boxes (OpenSpiel)

Jeff Christoffels
Department of Computer Science
KU Leuven, Belgium
jeff.christoffels@student.kuleuven.be

Steven Huygens
Department of Computer Science
KU Leuven, Belgium
steven.huygens@student.kuleuven.be

Abstract

This paper researches the application of multi-agent reinforcement learning in various game settings. Specifically, it examines the benchmark matrix games dispersion game, battle of the sexes, prisoner's dilemma and biased rock-paper-scissors, as well as Dots-and-Boxes. Several algorithms, including Q-Learning, minimax, and Monte Carlo Tree Search (MCTS) and a proposed extension of the minimax algorithm using a sliding window are tested on these different game types to assess their performance. The experiments are conducted using the OpenSpiel implementation of the games.

Keywords: Game Theory, Machine Learning, Minimax, Multi-agent Learning, OpenSpiel, Reinforcement Learning

1 Introduction

In our multi-agent world, the success of agents, particularly artificial intelligent (AI) agents, hinges on their ability to recognize and respond to the agency of others. To thrive in this dynamic environment, agents must be prepared to engage in various activities such as competition, collaboration, communication, coordination, and negotiation. By understanding and adapting to the agency of other agents, AI agents can navigate complex social interactions and achieve their objectives effectively.

Dots-and-Boxes [3] is a classic two-player paper-and-pencil game that, despite its simple rules, presents a remarkable level of complexity. The game's immense state space makes it an intriguing challenge for the development of AI systems. The objective of the research presented in this paper is to create an agent capable of learning to play Dots-and-Boxes using the code infrastructure offered by OpenSpiel [1]. The aim is to develop an agent that can adapt to any board size, ensuring its versatility across different game configurations.

This paper covers the assignment given in the class *Machine Learning: Project (H0T25A)* at the KU Leuven, which consists of four tasks. The first task concerns a literature review on the topics of the remaining three tasks. The result is a critical analysis of the previous work and how this project relates to it, which is covered in Section 2. The second task is about independent learning and dynamics of learning of four matrix games: dispersion game, battle of the sexes, the prisoners dilemma and biased rock-paper-scissors. Section 3 covers this task and is about agents learning to play these matrix games using variations of Q-learning. In task three, we are tasked with using minimax to solve a tiny version of Dots-and-Boxes. This implementation, together with two optimization techniques, are written out in Section 4. The fourth and final task challenges us to find a more advanced strategy that works for any size of Dots-and-Boxes. We propose a sliding window extension for the minimax algorithm, which is explained in Section 5.

2 Previous work

Bloembergen et al. [4] established a connection between Evolutionary Game Theory (EGT) and Reinforcement Learning (RL). They highlight the one-to-one mapping between games, players, strategies, and payoffs in classic game theory, and environments, agents, policies, and rewards in multi-agent systems. The authors demonstrate this mapping by showing that the replicator dynamics of EGT can be obtained from the continuous time limit of the Cross Learning RL algorithm. The paper also explores the application of Q-Learning, as well as its variants such as Lenient Frequency Adjusted Q-learning (LFAQ) and the Boltzmann exploration mechanism, in two-player normal-form games. They examine the influence of tuning the κ and τ parameters in these approaches, which proves to be significant for matrix games. Additionally, the concept of evolutionary stable strategies (ESS) is introduced in EGT to enhance the Nash Equilibrium concept. These concepts are explored in Section 3.

Joseph Barker and Richard Korf presented an approach to solve the game of Dots-and-Boxes optimally [2]. The authors employ a combination of pattern databases and a dynamic programming technique called retrograde analysis to achieve this goal. Pattern databases are used to estimate the optimal value of a game state by considering subproblems or patterns within the game. By precomputing and storing the values of these patterns, the authors can efficiently evaluate the desirability of a given game state, this is comparable to the algorithm we proposed in Section 5. Retrograde analysis is employed to work backward from the final game state to determine the optimal strategy for each game state encountered during the process. By iteratively assigning values to states based on the optimal moves leading to their successors, the authors compute the optimal values for all game states. The authors also introduce several optimizations to enhance the efficiency of their algorithm. They exploit symmetry and

transposition properties of the game to reduce the search space and accelerate the computation, which is also explored in Section 4. Additionally, they utilize a technique called incremental pattern databases to update the values of patterns during the retrograde analysis process, avoiding the need for complete recomputation. The solver successfully solves Dots-and-Boxes games of varying sizes, including a 4×5 size which was previously unsolved.

Zhuang et al. introduced novel strategies for enhancing the Monte-Carlo Tree Search (MCTS) algorithm in the game Dots-and-Boxes [19]. In Section 5, we compare our proposed algorithm to an MCTS-based algorithm. The authors propose a unique board representation technique that leverages game-specific knowledge to accelerate key operations on the board. This representation is efficiently implemented using disjoint set and stack data structures. Additionally, artificial neural networks (ANN) and various other optimizations are integrated into the MCTS framework to create an advanced AI agent named QDab. Experimental results demonstrate that QDab outperforms agents based on minimax search and Nimstring-Analysis approaches. This improved performance is attributed to the combination of the proposed board representation and the additional optimizations applied in the MCTS algorithm. The research contributes to advancing the state-of-the-art in Dots-and-Boxes AI by introducing innovative strategies and demonstrating their effectiveness through empirical evaluation.

3 Learning & dynamics

Normal-form games are often referred to as "one-shot games" because each player only makes one decision before the game concludes. These games can be represented in an n-dimensional table, where each dimension corresponds to the actions of an individual player. Each cell in the table corresponds to a specific action profile, which is the intersection of one action from each player. The cell contains a vector of utilities, also known as payoffs or rewards, for each player [11]. Two-player normal-form games can be modeled by a bi-matrix, which is why they are also referred to as matrix games. This section discusses four benchmark matrix games: dispersion game, battle of the sexes, the prisoner's dilemma and biased rock-paper-scissors. The payoff tables for these matrix games are shown in Table 1.

	A	B
A	-1, -1	1, 1
B	1, 1	-1, -1

(a) Dispersion game

	O	M
O	3, 2	0, 0
M	0, 0	2, 3

(b) Battle of the sexes

	C	D
C	-1, -1	-4, 0
D	0, -4	-3, -3

(c) Prisoner's dilemma

	R	P	S
R	0	-0.25	0.5
P	0.25	0	-0.05
S	-0.5	0.05	0

(d) Biased rock-paper-scissors

Table 1. Payoff tables of the matrix games

3.1 Reinforcement learning background

3.1.1 Q-learning. Q-learning is a prevalent learning algorithm in multi-agent reinforcement learning that can find an optimal policy in any finite Markov decision process (MDP). This model-free algorithm maximizes the expected value of the total reward. To achieve this, a utility Q is associated with each state-action pair. The stateless update rule for Q-learning when an agent takes action a and receives reward R is

$$Q(a) = Q(a) + \alpha(R + \gamma \max_a(Q(a') - Q(a)).$$

The learning rate is denoted by α, and the discount factor γ is used to include future rewards in the utility estimate [13].

3.1.2 ϵ-greedy Q-learning. A popular extension of the Q-learning algorithm that balances exploration and exploitation during the learning process is called ϵ-greedy Q-learning. An agent using ϵ-greedy Q-learning chooses a random action with probability ϵ and the action with the highest Q-value with probability $1 - \epsilon$. In other words, there is a probability ϵ that the agent explores new actions which may lead to higher rewards in the future. ϵ can be reduced as the agent gains experience to encourage exploitation of the learned-policy [18].

3.1.3 Lenient Boltzmann Q-learning. A second variant of Q-learning is lenient Boltzmann Q-learning. This variant uses a Boltzmann distribution to balance exploration and exploitation during the learning process. In contrast to ϵ-greedy Q-learning, which chooses actions randomly with a constant probability ϵ, lenient Boltzmann Q-learning selects actions based on a probability distribution that takes into account the current Q-values. the probability p_i of selecting the action i is given by

$$p_i = \frac{e^{Q_i/T}}{\sum_k e^{Q_k/T}}.$$

The exploration/exploitation trade-off is determined by the temperature parameter $T > 0$: as T approaches 0, the agent will always act greedily and select the strategy with the highest Q-value (pure exploitation), while as T approaches infinity, the agent's strategy becomes completely random (pure exploration) [8].

3.1.4 Nash equilibrium. A set of strategies and their corresponding payoffs, which guarantee that no player can improve their payoff by altering their strategy while the other players keep their strategies unchanged is known as a Nash equilibrium (NE) [17]. In a mixed-strategy NE, players are allowed to select a probability distribution over their set of

actions rather than being limited to selecting a single deterministic action. A mixed strategy NE is considered a pure strategy NE when it assigns a probability of 1 to a single action [12].

3.1.5 Pareto optimal. A combination of actions is Pareto optimal if no player can improve their payoff without decreasing the payoff of another player. In other words, if there doesn't exist another solution that guarantees that all players do at least as well while at least one players does better [17].

3.1.6 Evolutionary game theory. Classical game theory assumes that agents always act in a fully rational manner, but this assumption may not accurately capture the dynamics of real-world interactions. In contrast, evolutionary game theory incorporates concepts from biology, such as natural selection and mutation, to replace this assumption. In this framework, agents undergo a learning process to optimize their behavior and maximize their payoff, akin to the process of evolution in nature [4]. In evolutionary game theory, populations consist of a collection of individuals or agents that engage in interactions and competition, guided by specific strategies. Each individual within the population adopts a strategy that dictates their behavior within the game. Over time, the population undergoes evolution as individuals interact and learn from their experiences, and adjust their strategies in response to the outcomes of their interactions and the overall dynamics of the game. This process of evolution shapes the collective behavior and strategy distribution of the population as a whole [10].

3.1.7 Replicator dynamics. Replicator dynamics capture the temporal evolution of a strategy population and are mathematically represented as a set of differential equations. The fundamental replicator dynamic is expressed in the form of

$$\frac{dx_i}{dt} = [(Ax)_i - x \cdot Ax]x_i. \tag{1}$$

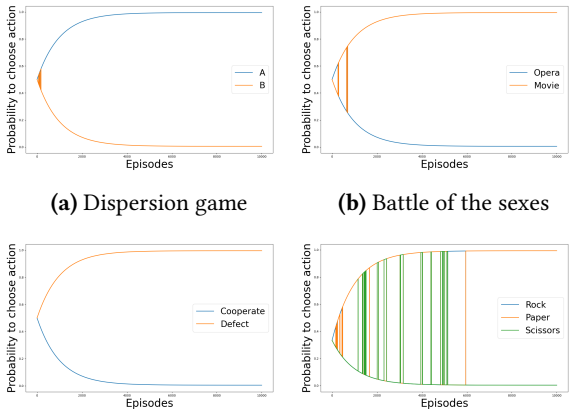
In this equation, x represents a probability vector that describes the population's state. The payoff received by replicator i in state x is denoted as (Ax). The average payoff of the population is represented by $x \cdot Ax$. [14]

3.2 Independent learning in benchmark matrix games

We implemented two variations of Q-learning, namely ϵ-greedy Q-learning and lenient Boltzmann Q-learning as explained in Section 3.1. For each matrix game listed in Table 1, our own implementation of a tabular Q-learner using the *OpenSpiel* framework was implemented [1].

The performance of the Q-learning algorithm can be significantly affected by the choice of hyperparameters. Since the optimal values of hyperparameters can depend on the problem being solved and the specific Q-learning implementation, it is important to experiment with different approaches for selecting hyperparameters and carefully adjust their values to achieve the best performance [16]. For the sake of comparison, all results discussed in the remainder of this section use a learning rate α of 0.05, a discount rate γ of 0.1, an initial exploration rate ϵ of 1 (for ϵ-greedy Q-learning) and an initial temperature T of 80 (for lenient Boltzmann Q-learning).

Figure 1 and Figure 2 show the empirical learning trajectories of the row player for the four matrix games for ϵ-greedy Q-learning and lenient Boltzmann Q-learning respectively. Two agents running the same algorithm played against each other for 10000 episodes. As can be seen on Figure 1, the exploration rate ϵ decreases in function of the episodes. In episode k, the exploration rate equals $\epsilon_i = \epsilon_{min} + (\epsilon_{max} - \epsilon_{min})e^{0.001 \cdot k}$, with $\epsilon_{min} = 0.01$. On Figure 2, the algorithm converges around episode 8000 in all four cases, this is because the temperature T starts at 80 and decreases 0.01 each episode to a minimum value of 0.01. In other words, around episode 8000, the algorithm will focus almost solely on exploitation.



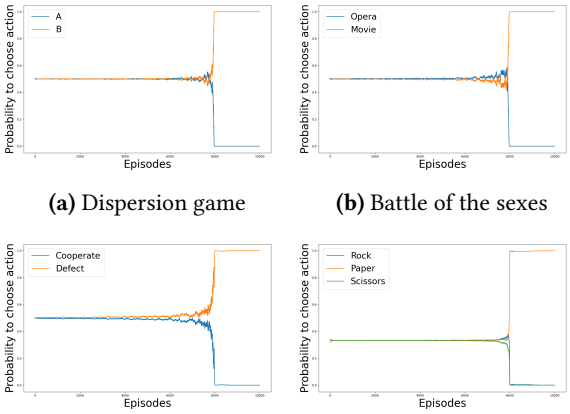
(a) Dispersion game **(b)** Battle of the sexes

(c) Prisoner's dilemma **(d)** Biased rock-paper-scissors

Figure 1. Empirical learning trajectories of a row player using ϵ-greedy Q-learning for matrix games

(a) Dispersion game. In the dispersion game, both players lose a point when they play the same action. Conversely, they gain a point when this is not the case. This matrix game has two pure NEs (A, B) and (B, A), both Pareto optimal, and one mixed NE, where both players play action A with a probability of $(0.5, 0.5)$.

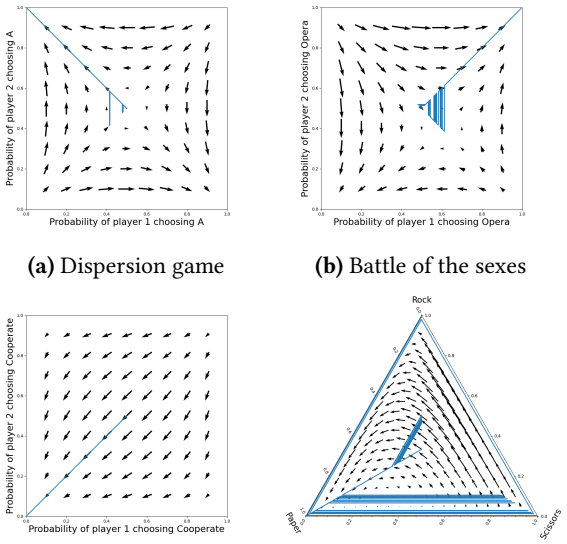
(b) Battle of the sexes: the battle of the sexes portrays a conflict of interest. Two players with distinct preferences need to reach a coordinated decision to maximize their joint satisfaction. Due to differing preferences, the players may initially need to make conflicting choices. One player prefers seeing a movie (M) while the other player prefers going to the opera (O). The battle of the sexes has two pure NEs (M, O) and (O, M), which are both Pareto optimal, and one



(a) Dispersion game **(b)** Battle of the sexes

(c) Prisoner's dilemma **(d)** Biased rock-paper-scissors

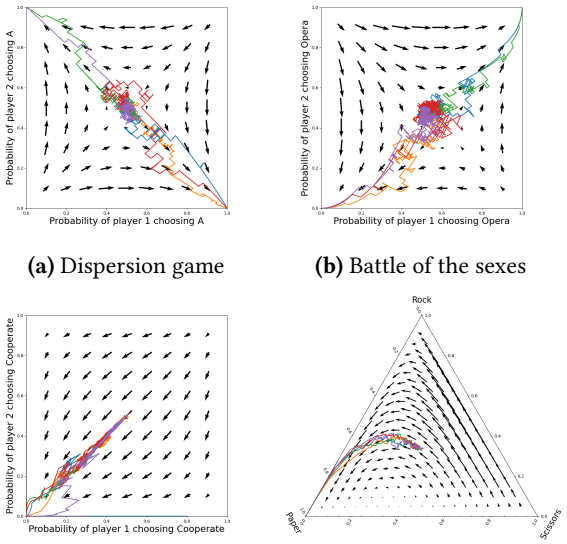
Figure 2. Empirical learning trajectories of a row player using lenient Boltzmann Q-learning for matrix games



(a) Dispersion game **(b)** Battle of the sexes

(c) Prisoner's dilemma **(d)** Biased rock-paper-scissors

Figure 3. Field plots and learning trajectories of ϵ-greedy Q-learning for matrix games



(a) Dispersion game **(b)** Battle of the sexes

(c) Prisoner's dilemma **(d)** Biased rock-paper-scissors

Figure 4. Field plots and learning trajectories of lenient Boltzmann Q-learning for matrix games

mixed NE where the row player and columna player pick opera with a probability of $(0.6, 0.4)$ respectively.

(c) Prisoner's dilemma: in the prisoner's dilemma, there are two individuals who have been arrested and are held in different prison cells. The prosecution doesn't have enough evidence to convict them on a serious charge but can still convict them on a lesser offense. The prisoners face a choice between cooperating with each other or betraying each other, which results in four potential outcomes. This game has one pure NE, namely (D, D), which is not Pareto optimal.

(d) Biased rock-paper-scissors: the game biased rock-paper, scissors is a modified version of the well-known rock-paper-scissors game. It incorporates biases in the payoffs assigned to each action. This version has one pure NE at (P, P). This Nash equilibrium is Pareto optimal, as biased rock-paper-scissors is a zero-sum game.

3.3 Dynamics of learning in benchmark matrix games

In Figure 1 and Figure 4, the directional field plots of the matrix games of Table 1 are shown. These field plots are obtained by implementing the replicator dynamics as explained in Section 3.1.7. Overlaid over these field plots are the learning trajectories of an agent using our implementation of ϵ-greedy Q-learning (Figure 3) and of five different agents using lenient Boltzmann Q-learning (Figure 4). Each agent starts with a Q-table filled with random values $\in [-4, 4]$.

As ϵ-greedy Q-learning is a greedy algorithm, there isn't really a trajectory to speak of. This is apparent when comparing Figure 3 to 4). An agent using ϵ-greedy Q-learning selects the action with the highest Q-value with a probability of $1 - \epsilon$, even if there is only a small difference between Q-values. Lenient Boltzmann Q-learning enables ongoing exploration of the action space by assigning probabilities to each action proportional to their Q-values. This allows for a more continuous exploration of the available actions. This approach ensures that even sub-optimal actions have a probability of being selected, which can be advantageous in complex environments where the optimal action may not be immediately apparent.

(a) Dispersion game: the agent using ϵ-greedy Q-learning reaches the NE (B, A) as seen in Figure 3a and the agents using lenient Boltzmann Q-learning reach both NE (A, B) and (B, A) depending on the initial Q-values and their opponents as seen in Figure 4a. The mixed NE at $(0.5, 0.5)$ is not asymptotically stable.

(b) Battle of the sexes: in the learning trajectory shown in Figure 3b, the agent reaches the NE at (O, O). Figure 4b shows the learning trajectory of five different agents that reach both NEs at (O, O) and (M, M). The asymptotically unstable mixed NE is located at $(0.6, 0.4)$.

(c) Prisoner's dilemma: both the agent using ϵ-greedy Q-learning (Figure 3c and the agents using lenient Boltzmann Q-learning (Figure 4c) reach the NE at (D, D).

(d) Biased rock-paper-scissors: Figure 3d (which is already hard to read with a single learning trajectory) and Figure 4d show that the ϵ-greedy Q-learning agent and the lenient Boltzmann Q-learning agents all reach the NE at (P, P).

4 Minimax for small Dots-and-Boxes

Dots-and-Boxes [3] is a combinatorial game that requires nothing more than paper and pencil to play. The game is played on a rectangular grid of dots, forming a grid of $r \times c$ squares, also known as boxes. Two players take turns drawing lines, either horizontally or vertically, between adjacent dots. When a player completes a box by closing its boundaries with their line, they claim that box as their own and earn an additional turn. The objective of the game is to capture as many boxes as possible. At the end of the game, the player with the highest number of captured boxes emerges as the winner.

Dots-and-Boxes has $|S| = 2^{r(c+1)+(r+1)c}$ states and $log_2|S|$ of unique games which can be played, with r equal to the number of rows and c equal to the number of columns. At present, there is no known universally optimal strategy for playing Dots-and-Boxes. The game remains challenging and unsolved in terms of finding a foolproof winning approach. Currently, the largest game that has been successfully solved is a 4×5 grid configuration [2].

This section covers the problem of solving small versions of Dots-and-Boxes using the minimax algorithm. By exhaustively exploring the entire tree of possible moves, the minimax algorithm evaluates and assigns a value to each game state, indicating its desirability. However, the game of Dots-and-Boxes exhibits a significantly wide game tree, even for small grid sizes. Consequently, the minimax algorithm can already be slow in processing and analyzing the numerous possibilities that arise during gameplay [6].

4.1 The minimax algorithm

The minimax algorithm [15] is a recursive decision-making algorithm commonly employed in two-player games. It assigns a value to each game state, representing the desirability of that state. The algorithm evaluates these values by analyzing the possible moves and their corresponding outcomes.

The algorithm operates under the assumption that both players aim to maximize their own outcomes and play optimally. It proceeds by alternating between two players: the maximizing player and the minimizing player. The maximizing player assesses the values of their possible moves using an evaluation function. The minimizing player, representing the opponent, subsequently selects the move that minimizes the maximum value for the maximizing player.

Throughout the recursive process, the algorithm systematically traverses the game tree, examining all possible moves and the corresponding resulting states. This exploration persists until a terminal state is reached. In terminal states, the evaluation function provides a value that gauges the desirability of the respective state.

By recursively evaluating and comparing the values, the minimax algorithm identifies the best move for the maximizing player, assuming optimal play by both players. The goal is to maximize the minimum value that can be achieved, considering the opponent's best possible moves.

4.2 Optimizations

4.2.1 transposition table. During the minimax algorithm a lot of similar states get revisited and expanded. For example in Dots-and-Boxes suppose the first player places a horizontal line at the top left of the board and the second player places a vertical line at the top left of the board. This would give a state where the vertical and horizontal lines at the top left of the board are drawn. Now suppose the actions of the players are switched between each other. The first player draws the vertical line at the top left and the second player draws the horizontal line at the top left of the board. These two actions would give the same state as before, where the horizontal and vertical lines at the top left of the board are drawn. This example illustrates how different action paths could lead to the same state. This insight can be used to speed up the minimax algorithm using a transposition table. A transposition table is a data structure which takes a game state as a key and maps this to the corresponding evaluation of that state. During minimax after the evaluation of a state is gotten, the evaluation can be saved in the transposition table with the state as its key. The next time a state is encountered, we can simply check if it is present in the transposition table. And if it is, we do not have to re-search the game tree below this state.

4.2.2 Symmetries. Another insight which can optimize the minimax is that some states are symmetric. The Symmetric states share the same value because they share the

same symmetric actions. For example, a state of a square grid in a Dots-and-Boxes game has eight states which are equivalent. The state can be rotated 90, 180 and 270 degrees. This gives four symmetric states. Every one of these states are symmetric to the state flipped over the x-axis. This gives eight symmetric states in total. A state of a rectangular grid in Dots-and-Boxes only has four symmetries. A state can be rotated by 180 degrees, it can be flipped over the x-axis and it can be flipped over the y-axis.

There are two strategies to exploit the symmetries to optimize the minimax algorithm. When a state is encountered in the first strategy, the canonical representation of a state gets calculated. Then the canonical representation is used to check if the value of the state is present in the transposition table. If it is present, the value can simply be returned. If it is not present, the game tree under the state gets searched. The value gotten from the search is then stored in the transposition table with the canonical representation as the key. This strategy will reduce the size of the transposition table.

The second strategy does not get the canonical representation of a state to use it as the key for the transposition table. Instead it just uses the state as the key to check if its value is present in the transposition table. If the value is present then that value gets returned. If it is not, the value of the state gets calculated. Then all the symmetries of the state get calculated. The value from the state is then saved in the transposition table for each of the symmetries as keys. In this strategy, the canonical representation should not be calculated each time a state is encountered. This makes the second strategy faster than the first strategy. However, the transposition table will not be reduced in size when using the second strategy.

4.3 Implementation

The starting point and baseline for this section is the naive template provided by our assessors. Dots-and-Boxes has a very wide game tree, which makes minimax already slow for tiny grids. This template will therefore have to be optimized, which happens in two steps.

4.3.1 implementing transposition table. To optimize the minimax algorithm by using a transposition table, the state of a game of Dots-and-Boxes needs to be represented correctly. To represent which lines have been drawn we have decided to use a bitmap. Each line is assigned a number. The horizontal line at the top left is assigned to the value zero. The horizontal to its right is assigned to the value one and so on for all the horizontal lines. Then the same is done for the vertical lines starting at the top left going to the bottom right. However, here the top left vertical line is represented by the value equal to the number of horizontal lines. The vertical line to its right is assigned to the number equal to the number of horizontal lines plus one and so on. To calculate the bitmap we then iterate over the drawn lines. For each of these drawn lines the n-th bit in the bitmap is set to one starting from the left of the bitmap, with n equal to the value assigned to the line. This bitmap now represents which lines are drawn. This, however, is not enough to uniquely represent a state of a game of Dots-and-Boxes. We also need to add the current player id and the amount of points each of the two players have. Together these three values, the bitmap, the points and the current player, are used as keys in the transposition table. The value of a state is either a '1', a '−1' or a '0'. A '1' denotes that the current player can win from the state if both players play optimally. A '−1' denotes that the opponent of the current player can win from the state if both players play optimally. Lastly, a '0' represents that the game will end in a tie if both players play optimally.

Our algorithm which uses the transposition table optimization checks if a state is present in the transposition table. If it is present then its value gets returned. If the state is not present then the game tree below the node gets searched and its value is then saved in the transposition table with the state as its key.

4.3.2 implementing symmetries. The optimization of Section 4.3.1 utilized the equivalence between different trajectories to enhance the search. Now, we aim to leverage the fact that multiple states are completely symmetric. For example, consider two states that are rotations of each other. They will inevitably have the same value since one can apply the same strategy with actions rotated accordingly. By capitalizing on these symmetries, it is possible to avoid re-exploring symmetric states.

To implement the symmetries three methods are made. The first method maps the values of the drawn lines to the values of the drawn lines if they are rotated by 90 degrees. The second method is used to map the values of the drawn lines to the values of the drawn lines when they are flipped over the x-axis. And a last method is made to map the values of the drawn lines to the values of the drawn lines when they are flipped over the y-axis. All these methods use mathematical formulas to map these numbers to their symmetric numbers.

As discussed in Section 4.2.2 there are two strategies to exploit the symmetries of certain states of a game of Dots-and-Boxes. We implemented both of these strategies. To get the canonical representation of a state for the first strategy as is discussed in 4.3.1, the bitmap of every symmetric state is calculated. Then, the bitmap with the smallest value is selected.

4.4 Experimental results

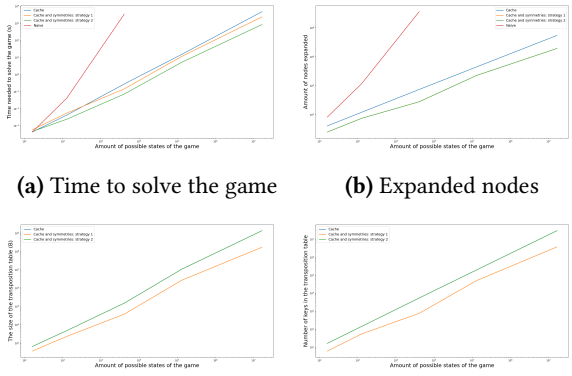
In Section 4.3 four different algorithms were discussed: the naive minimax, the algorithm using the transposition table and the two different strategies to exploit symmetries. The first strategy used canonical representations to check if values are stored in the transposition table for a state, while

Dimension	Amount of states
1×1	16
1×2	128
2×2	4096
2×3	131072
3×3	16777216

Table 2. Amount of states for each dimension

the second did not use canonical representations of states to avoid computing these representations every time a state is encountered.

In this section, we compare these four algorithms using four metrics. The first metric is the amount of time the algorithm needs to solve a game. The second metric is the amount of nodes in the game tree which need to be expanded by the algorithm. If a node needs to be expanded, it means that the state is not present in the transposition table when it is reached by the algorithm. The third metric is the size of the transposition table in bytes. The last metric is the amount of keys stored in the transposition table. These metrics are plotted in function of different board sizes, more specifically in function of the amount states of a game. A game of Dots-and-Boxes has $|S| = 2^{r(c+1)+(r+1)c}$ amount of states. For this experiment, we considered the following dimensions: 1×1, 1×2, 2×2, 2×3 and 3×3. In Table 2 you can see the amount of states for each of these dimensions.



(a) Time to solve the game **(b)** Expanded nodes

(c) Transpostion table size **(d)** Transposition table keys

Figure 5. Comparison of four different minimax algorithms

(a) Time. As can be seen in Figure 5a, the naive minimax algorithm is the slowest. The time needed to solve games of dimensions 2×3 and 3×3 by the naive algorithm have been omitted because this would have taken too long to calculate. The algorithm which uses the transposition tables to cache the values of states is faster than the naive solution since it does not have to expand a state which it has already visited. The algorithms which exploit the symmetries are the fastest. However, the second strategy of exploiting the symmetries is considerably faster than the first strategy. This is because the second strategy does not have to compute the canonical representation of a state as described in Section 4.2.2.

(b) Amount of nodes expanded. The naive algorithm has to expand every node in the game tree it encounters. In Figure 5b, it can be seen that fewer nodes have to be expanded by the three other algorithms since the algorithms uses the transposition tables to cache the values of states it already visited. The graph lines of the algorithms which exploit the symmetries coincide. Two upward deviations are visible on the graph lines of these two algorithms. These deviations occur at the positions of the 1×2 and 2×3 dimensions. These deviations can be explained because the games with these dimensions are rectangular. Rectangular games can only exploit four symmetries while square games can exploit eight as is discussed in Section 4.2.2.

(c) Transposition table size. In this metric the naive algorithm is not considered since it does not use a transposition table. In Figure 5c, the graph lines of the algorithm which only implements the transposition table to cache the results and the algorithm which exploits symmetries using the second strategy coincide. These algorithms both have a larger transposition table size than the algorithm which exploits symmetries by using the first strategy. This is because this algorithm only has to store the canonical representation of symmetric states, while the other two algorithms have to store all the states.

(d) number of keys stored in transposition table. In this metric the naive algorithm is not considered either since it does not use a transposition table. In Figure 5d, the graph lines of the same algorithms coincide as was the case in Figure 5c. The number of keys stored in the transposition table for these two algorithms is greater than the number for the algorithm which uses the first strategy to exploit symmetries. This happens for the same reason as previously explained for the transposition tabel size.

5 Full Dots-and-Boxes

The solution to Dots-and-Boxes explained in Section 4 only work for small games, which is illustrated by the experimental results of Section 4.4. This category explores a more advances strategy that works for any game size. For minimax, the search space grows exponentially as the game size increases, leading to longer computation times. Monte-Carlo Tree Search (MCTS) offers a solution to this problem. MCTS is a suitable algorithm to explore the state space of a Dots-and-Boxes game by iteratively expanding a tree structure, sampling moves, and updating statistics to make informed decisions.

5.1 Monte-Carlo Tree Search

MCTS is an advanced search algorithm that combines simulation and statistical sampling to evaluate states in a best-first manner [5, 7]. The Monte-Carlo tree that is generated during the process contains nodes representing different board states, each of which is augmented with the number of visits and a corresponding value. The algorithm iteratively performs four steps until a stop condition is met.

1. *Selection*: The Monte-Carlo tree is recursively traversed from the root to a leaf node, following a selection policy that aims to balance the exploration of moves with limited simulations and the exploitation of moves with high value estimates. This selection policy ensures a fair exploration of the game tree, allowing for a comprehensive evaluation of different move possibilities. The algorithm traverses the game tree by selecting child nodes based on their evaluation scores. The evaluator is responsible for assigning these scores or values to the game states.
2. *Expansion*: Once a leaf node has been reached during the traversal of the Monte-Carlo tree, its successors or child nodes are added to the tree. From these successors, one child node is selected for simulation.
3. *Simulation*: a specific strategy is employed to play the game starting from the state that was selected during the expansion step. This strategy typically involves making moves in a uniformly random manner.
4. *Backpropagation*: The outcome of the simulation is employed to update the value estimates of all nodes along the selected path from the root to the leaf.

The Upper Confidence Bound for Trees (UCT) [9] function is another component of the MCTS algorithm. It is used to balance the exploration and exploitation aspects of the search process. This balance can be adjusted by changing the UCT's exploration constant.

5.2 Sliding window for varying sized games

With the use of a sliding window algorithm a solver for a game of Dots-and-Boxes with dimensions $r_1 \times c_1$ can be reused to play a game of Dots-and-Boxes with dimensions $r_2 \times c_2$, where $r_1 < r_2$ and $c_1 < c_2$. Suppose you have an algorithm which solves a 2×2 game of Dots-and-Boxes and you need to play a game of 7×7 Dots-and-Boxes. This can be done by iterating over all the 2×2 grids on the 7×7 board. For every 2×2 grid you let the 2×2 solver decide which action is the best to play from that 2×2 state and what value is associated with the state. During The iteration you keep track of the maximum of these values and the best actions associated with it. After the iteration you play one of these best actions on the 7×7 Dots-and-Boxes game.

5.3 Implementation

5.3.1 Monte-Carlo Tree Search implementation. We have implemented the MCTS using the *mcts.py* algorithm provided by OpenSpiel [1]. UCT's exploration constant is set to 2. We decided to set the max number of simulations equal to 80 and the rollout count to 1. Working with this value is not optimal to find the best action in a state of Dots-and-Boxes, but it is necessary to not have to wait too long for the algorithm to decide which action to take. The evaluator used is a random rollout evaluator provided by OpenSpiel. The evaluator computes the average outcome obtained by playing random actions from the given state until the game concludes.

5.3.2 Sliding window implementation. To get an action for a state of a game of Dots-and-Boxes of varying size, we decided to implement the concept described in Section 5.2. For the solver of the smaller grid, we decided to use the solution of the minimax algorithm of a game of size 2×2. We stored this solution in a transposition table. From this solution we can get the best actions to take in a certain 2×2 state and also how valuable it is to play from a certain state. We keep track of all the best actions associated with the states with the highest values. We then decide to play one randomly chosen action out of these actions. We could have used the same method but using the solution of a 3×3 grid. However, we decided to not use this due to storage size constraints.

5.4 Experimental results

To observe how well a certain algorithm plays a game of Dots-and-Boxes we decided to let algorithms play 200 games of Dots-and-Boxes of size 7×7 against each other. The first 100 games one plays the first move the last 100 games the other plays the first move. We compared three algorithms: An algorithm which plays a random move as a baseline, the MCTS algorithm as described in Section 5.3.1, and the sliding window algorithm using the 2×2 minimax solution as described in Section 5.3.2.

Player		Number of games won		Point difference	
Player 1	**Player 2**	**Player 1**	**Player 2**	**Player 1**	**Player 2**
Random	MCTS	2	98	-2534.0	2534.0
MCTS	Random	95	5	2390.0	-2390.0
Random	SW	0	100	-4368.0	4368.0
SW	Random	100	0	4296.0	-4296.0
MCTS	SW	0	100	-3822.0	3822.0
SW	MCTS	100	0	3942.0	-3942.0

Table 3. Results of 100 7×7 Dots-and-Boxes games played by competing algorithms

The results of this experiment can be seen in Table 3. In this table "player 1" is the player which plays the first move. In the table the algorithm which uses the sliding window

method is denoted by "SW". As can be seen in Table 3 the MCTS algorithm wins nearly every game against the algorithm which plays random actions. The sliding window algorithm wins every game against the two other algorithms. We think this can be explained by the fact that the sliding windows algorithm knows the best action for every 2×2 state with full certainty. Another reason is because the MCTS can not perform many simulations and rollouts because of the time constraint.

Algorithm	Time needed (s)
Random	0.285
MCTS	814.429
SW	1.715

Table 4. The time needed for an aglorithm to play 100 games of Dots-and-Boxes of size 7×7 against itself

Another useful metric to compare is how long it takes for an algorithm to decide which move to play. To measure this metric we decided to let the algorithm play 100 games of Dots-and-Boxes of size 7×7 against itself. The results of this experiment can be seen in Table 4. It is clear that the algorithm which plays random actions is the fastest, since it does not need to do complex calculations to come up with an action. The algorithm which uses the sliding window is a bit slower, but still relatively fast. This can be explained by the fact that the iteration over all the possible 2×2 grids is not an expensive computation and the fact that the best actions for a certain state are stored in a transposition table. The MCTS algorithm is the slowest since it needs to perform the four steps as described in Section 5.1 every time it needs to take an action.

6 Conclusion

The observed dynamics and convergence of ϵ-greedy and lenient Boltzmann Q-learning applied on matrix games are in line with Game Theory. NEs can be observed and are the final state of an agent if the hyperparameters of the Q-learning algorithm are tuned to suit the application. Lenient Boltzmann Q-learning allows for a more continuous exploration of the available actions, compared to ϵ-greedy Q-learning.

The minimax algorithm can be used to solve small instances of Dots-and-boxes but has long computation times for larger board sizes as the search space grows exponentially as the game size increases. Optimizing the algorithm by using transposition tables and exploiting symmetries decrease these computation times, but are still insufficient to solve large Dots-and-Boxes games in a reasonable time. There is a trade-off to be made in the way that you implement optimizations for minimax, as the transposition table increases in storage size as the necessary time to solve the game decreases.

MCTS is a more advanced search algorithm that works for any game size of Dots-and-Boxes. We propose a sliding window extension of the minimax algorithm to reuse a minimax algorithm for small game sizes for larger game sizes. This sliding window algorithm beats the MCTS algorithm, scales to any game size and requires significantly less time to play a game.

6.1 Future work

Currently, our implementation of the sliding window concept, as described in Section 5.3.2, only supports solvers for 2×2 grids. This, however, can be solved by implementing generalised methods to map the drawn lines from the smaller grid to the drawn lines of the bigger grid.

Our implementation of the sliding window concept can not be used to play $1 \times N$ and $N \times 1$ Dots-and-Boxes since the 2×2 solver combined with the sliding window approach needs grids with sizes greater than or equal to 2. This can be fixed by using a $1 \times M$ or $M \times 1$ solver, where $M < N$, for game instances where this is needed.

It could be interesting to compare how using different solvers for different sizes in the sliding window implementation affect how well the algorithm plays a game of Dots-and-Boxes.

Another aspect of our proposed sliding window minimax algorithm is the method which selects the to be executed action on the main board from the various solution of the sub-boards. Currently, a random move is selected from the best actions gotten after iterating over all the 2×2 grids. Improvements could be made to this selection mechanism.

Code

The code used for this project can be found in the directory of Jeff Christoffels (r0625400), on path `/cw/lvs/NoCsBack/vakken/ac2223/H0T25A/ml-project/r0625400/` on the departmental computers. The code used for Section 3 is located in directory *task_2* and the code used for Section 4 is located in directory *task_3*. The agent using the sliding window minimax algorithm explained in Section 5 can be found in the file *dotsandboxes_agent.py*. The code used for the remainder of Section 5 is located in directory *task_4*.

Workload

We estimate that we spent more than 130 hours each on the project, which is more than we expected for a 3 ECTS course. One of the reasons for this high number of working hours is the lack of documentation for the OpenSpiel framework, which makes it necessary to browse through examples to get a grip on the framework. We both met up physically at least once a week for a full day to work on the project and divided the tasks equally.

References

[1] Openspiel: A framework for reinforcement learning in games. *CoRR*, abs/1908.09453, 2019.

[2] J. Barker and R. Korf. Solving dots-and-boxes. *Proceedings of the AAAI Conference on Artificial Intelligence*, 26(1):414–419, Sep. 2021.

[3] E. R. Berlekamp. *The dots and boxes game: sophisticated child's play*. CRC Press, 2000.

[4] D. Bloembergen, K. Tuyls, D. Hennes, and M. Kaisers. Evolutionary dynamics of multi-agent learning: A survey, 2015-01-01.

[5] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[6] L. Busoniu, R. Babuska, and B. De Schutter. A comprehensive survey of multiagent reinforcement learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 38(2):156–172, 2008.

[7] G. Chaslot, S. Bakkes, I. Szita, and P. Spronck. Monte-carlo tree search: A new framework for game ai. 01 2008.

[8] A. Kianercy and A. Galstyan. Dynamics of softmax q-learning in two-player two-action games. *CoRR*, abs/1109.1528, 2011.

[9] L. Kocsis and C. Szepesvári. Bandit based monte-carlo planning. volume 2006, pages 282–293, 09 2006.

[10] R. Lahkar and W. H. Sandholm. The projection dynamic and the geometry of population games. *Games and Economic Behavior*, 64, 2008.

[11] T. W. Neller and M. Lanctot. An introduction to counterfactual regret minimization regret in games. *EAAI-13: The 4th Symposium on Educational Advances in Artificial Intelligence*, 2013.

[12] M. J. Osborne. Solutions for an introduction to game theory. *Chinese Science Bulletin*, 48, 2004.

[13] L. Panait and K. Tuyls. Theoretical advantages of lenient q-learners: An evolutionary game theoretic perspective. 2007.

[14] L. Panait, K. Tuyls, and S. Luke. Theoretical advantages of lenient learners: An evolutionary game theoretic perspective. *Journal of Machine Learning Research*, 9:423–457, 03 2008.

[15] S. Russel and P. Norvig. *Artificial intelligence—a modern approach 3rd Edition*. 2012.

[16] A. D. Tijsma, M. M. Drugan, and M. A. Wiering. Comparing exploration strategies for q-learning in random stochastic mazes. In *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–8, 2016.

[17] K. Tuyls and A. Nowé. Evolutionary game theory and multi-agent reinforcement learning. *The Knowledge Engineering Review*, 20(1):63–90, 2005.

[18] M. Wunder, M. L. Littman, and M. Babes. Classes of multiagent q-learning dynamics with epsilon-greedy exploration. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 1167–1174, 2010.

[19] Y. Zhuang, S. Li, T. V. Peters, and C. Zhang. Improving monte-carlo tree search for dots-and-boxes with a novel board representation and artificial neural networks. In *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 314–321, 2015.