Quixo optimal agent strategy

Borella Simone
s317774
borellasimone00@gmail.com

January 30, 2024

Abstract

The game Quixo, played on a 5x5 grid, involves two players which aim is to align five identical symbols. This game requires innovative strategies to win, and here are described the strategies exploration steps to find an optimal agent strategy capable of solving Quixo game in an efficiently way.

1 Introduction

Quixo is known for its complexity and the multitude of potential moves at each turn, making it a subject for the application of artificial intelligence techniques. My objective is to analyze and present different approaches and methodologies that can be employed to develop an agent capable of making optimal decisions in the context of Quixo.

To achieve this objective, we experimented with three distinct approaches:

- MiniMax with alpha-beta pruning
- Deep Q-learning
- Value iteration with backward induction

Each approach brings its own set of strengths and challenges to the table, offering a diverse perspective on the problem.

Since, as explained in results, Quixo 5x5 is a draw game, which means that if both player agents are optimal neither player can win, in testing phase the initial turn is not specified, while is specified in Quixo 4x4 testing, which is a first-player-win game.

2 MiniMax with alpha-beta pruning

2.1 Introduction

The MiniMax algorithm is a decision-making algorithm used in two-player games with perfect information. Its primary objective is to determine the optimal move for a player, assuming that the opponent will also make optimal moves. The algorithm is based on the principle of minimizing the possible loss for a worst-case scenario.

2.2 Implementation

The game states are represented as nodes of a tree and each level of the tree corresponds to a turn in the game. For each leaf node in the tree a value is assigned based on an heuristic evaluation function that estimates the desirability of that game state for the current player.

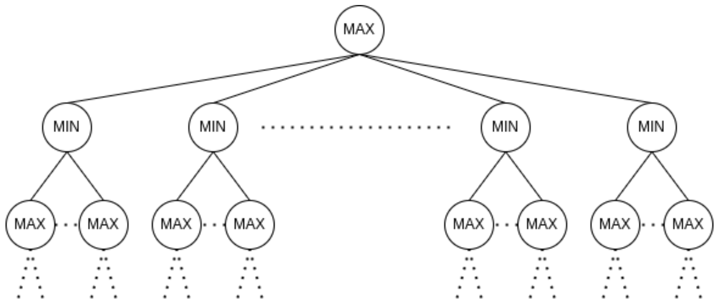

Figure 1: MiniMax state tree

The heuristic evaluation function is described as:

$$H(N, C_{max}, O_{max}) = N^{C_{max}} - N^{O_{max}} \tag{1}$$

where N is the board size, C_{max} is the maximum number of current player symbols in the same line and O_{max} is the maximum number of opponent player symbols in the same line, where a line can be an horizontal, vertical or diagonal line.

Starting from the root node, the algorithm recursively explores the tree by considering all possible moves at each level. At a Max node, the algorithm selects the move that leads to the maximum value among its child nodes. At a Min node, the algorithm selects the move that leads to the minimum value among its child nodes.

Alpha-beta pruning is an optimization technique used in the minimax algorithm to reduce the number of nodes evaluated in the search tree. During the tree recursive visit two parameters are maintained, alpha and beta which respectively represent the maximum value that the maximizing player has found so far at any choice and the minimum value that the minimizing player has found so far at any choice. During the traversal of the tree, when the maximizing player encounters a node with a value greater than or equal to beta, it realizes that the minimizing player)will never choose this path because the opponent has a better option and the entire subtree can be pruned. Similarly this happens when a minimizing player encounters a node with a value less than or equal to alpha.

2.3 Results

Since the action space is wide, having at most $4 \cdot N$ possible moves for each state, where N is the board size, the tree will have about $(4N)^{tree\_height}$ leaf nodes to compute.

Tree height	Leaf nodes
2	256
3	4096
4	65536

Figure 2: Tree leaf nodes with $N = 4$

Tree height	Leaf nodes
2	400
3	8000
4	160000

Figure 3: Tree leaf nodes with $N = 5$

To be computational efficient, the maximum depth I took in account is 3, since with a higher depth the agent is extremely slow.

Here are shown some testing results in a 5x5 table ($N = 5$):

Agent	Opponent Agent	Win rate	Lose rate	Draw rate
MiniMaxAgent(2)	RandomAgent	100.0%	0.0%	0.0%
MiniMaxAgent(2)	MiniMaxAgent(2)	49.0%	51.0%	0.0%
MiniMaxAgent(3)	MiniMaxAgent(2)	100.0%	0.0%	0.0%

Table 1: Agent performance comparison with $N = 5$

Here are shown some testing results in a 4x4 table ($N = 4$):

Agent	Opponent Agent	Initial player	Win rate	Lose rate	Draw rate
MiniMaxAgent(2)	RandomAgent	0	100.0%	0.0%	0.0%
MiniMaxAgent(2)	RandomAgent	1	99.0%	1.0%	0.0%
MiniMaxAgent(2)	MiniMaxAgent(2)	0	0.0%	0.0%	100.0%
MiniMaxAgent(2)	MiniMaxAgent(2)	1	0.0%	0.0%	100.0%
MiniMaxAgent(3)	MiniMaxAgent(2)	0	100.0%	0.0%	0.0%
MiniMaxAgent(3)	MiniMaxAgent(2)	1	0.0%	0.0%	100.0%

Table 2: Agent performance comparison with $N = 4$

3 Deep Q-learning

3.1 Introduction

Quixo game can be easily seen as a Markov Decision Process (MDP), which means that given a state the action that maximizes the future reward depends only on the current state.

Deep Q-Learning is a reinforcement learning technique that combines Q-Learning, an algorithm for learning optimal actions in an environment, with deep neural networks. It aims to enable agents to learn optimal actions in complex, high-dimensional environments. By using a neural network to approximate the Q-function, Deep Q-Learning can handle environments with large state spaces.

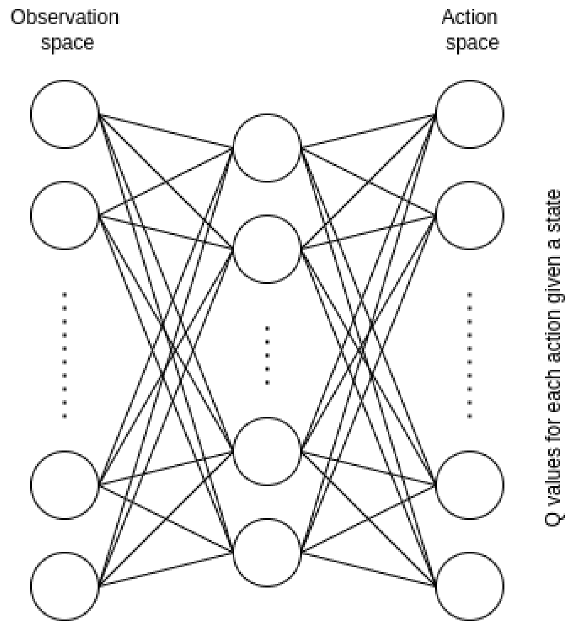

Figure 4: Q-network

Q-learning aims to learn the action-value function, denoted as Q-function, which represents the expected cumulative reward of taking an action in a particular state and following a certain policy. Q-learning is based on the Bellman equation, which expresses the relationship between the value of a state-action pair and the values of its successor states.

The update rule is given by:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \cdot \left[(R_{t+1} + \gamma \cdot \max_{a'} Q(s_{t+1}, a')) - Q(s_t, a_t) \right] \tag{2}$$

3.2 Implementation

Since the observation space counts $3^{N \cdot N}$ possible state combinations and the action space counts a maximum of $4 \cdot N$ possible actions, a tabular Q-learning would be too expansive in terms of memory occupation, since for each state there should be a number of action-values equal to the number of actions.

Assuming that each value is a 4 byte integer the occupied memory is equal to:

$$Memory\_occupation = observation\_space \cdot action\_space \cdot value\_occupation = 3^{N \cdot N} \cdot 4 \cdot N \cdot 4 \quad bytes$$

which is equal to 656.8 MB with $N = 4$ and 15.4 TB with $N = 5$.

This is why a deep Q-learning solution with a neural network for function approximation is necessary.

The neural network is composed by three fully connected layers, the input layer (state input) with $N \cdot N$ neurons, an hidden layer with 256 neurons and an output layer (action-value output) with $4 \cdot N$ neurons.

The reward function is pretty simple, if at the end of an episode the agent wins the reward is positive, otherwise 0:

$$R(s,a) = \begin{cases} 200 - episode\_number & \text{agent wins} \\ -(200 - episode\_number) & \text{agent lose} \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

In this way theoretically the agent strategy will converge to a policy derived by the action value function which objective is to win in the minimum number of steps.

Choosing the appropriate value for the exploration-exploitation tradeoff parameter ϵ is a critical decision in reinforcement learning. This parameter is commonly used in strategies such as epsilon-greedy policies, which determine how the agent balances between exploring new actions and exploiting the current best-known actions. In this case a GLIE (Greedy in the limit of infinite exploration) ϵ parameter is chosen with the following equation:

$$\epsilon_k = \frac{b}{b+k}, \quad b = k \cdot \frac{\epsilon_{final}}{1 - \epsilon_{final}} \tag{4}$$

where k is the current episode number.

The idea behind GLIE is to ensure that the agent explores sufficiently during the initial stages of learning when knowledge about the environment is limited, gradually becoming more greedy as it accumulates more experience and converges toward an optimal policy.

3.3 Results

Several experiments were done changing reward function and hyperparameters.

Below are explained the training steps followed using both the reward function described in Equation 3:

1. 100000 episodes training against RandomAgent
2. 100000 episodes training against RandomAgent and MiniMaxAgent(2)
3. 100000 episodes training against MiniMaxAgent(2)

Testing results:

Agent	Opponent Agent	Win rate	Lose rate	Draw rate
DeepQLearningAgent	RandomAgent	87.0%	13.0%	0.0%
DeepQLearningAgent	MiniMaxAgent(2)	0.0%	100.0%	0.0%

Table 3: Agent performance comparison with $N = 5$

Agent	Opponent Agent	Initial player	Win rate	Lose rate	Draw rate
DeepQLearningAgent	RandomAgent	0	84.0%	16.0%	0.0%
DeepQLearningAgent	RandomAgent	1	67.0%	33.0%	0.0%
DeepQLearningAgent	MiniMaxAgent(2)	0	0.0%	100.0%	0.0%
DeepQLearningAgent	MiniMaxAgent(2)	1	0.0%	100.0%	0.0%

Table 4: Agent performance comparison with $N = 4$

4 Value iteration with backward induction

4.1 Introduction

The solution proposed here is based on the findings presented in the cited paper [ST20]. The paper propose a solution for finding an optimal strategy, assuming perfect players, treating the Quixo game as a Markov Decision Process (MDP) and using the well known Value Iteration algorithm assigning an outcome (Win, Loss, Draw) for each state of the game.

4.2 Implementation

Here the objective is to find the optimal state-value function (state-values are also called outcomes here) of the MDP.

Implementation details are referred to Quixo 5x5 ($N = 5$).

A state of the game consists of a board and an active player (X player is assumed to be the active player). The initial state is the empty board. A state is terminal if its board contains a line of Xs or Os tiles. The children of a given state are all states obtained by a move of the active player (and then swapping the Xs tiles with Os tiles to keep the X active player representation). A terminal state has no children since the game is over and there is no valid moves. The parents of a state are defined analogously. The set of states and parent-child relations induce the game graph of Quixo, which is neither a tree nor an acyclic graph.

Each state has a state value, also called outcome which can be either active-player-Win, active-player-Loss, or Draw. For brevity, the active-player part is omitted, and a Win (resp. Loss and Draw) state denotes a state whose outcome is Win (resp. Loss and Draw).

For terminal states the outcome is defined as follows (take in account that every time a move is taken the active player is swapped):

- Win if there is a line of Xs
- Loss if there aren't Xs lines and there is a line of Os

For non-terminal states, the outcome is inductively defined as follows:

- Win if there is at least one Loss child
- Loss if all children are Win
- Draw otherwise

The most natural representation of a state in memory is to use an array of 25 elements where each entry takes values from the set 0, 1, 2 to map empty, X, O. Assuming a typical 1-byte char element, such a representation requires 25 bytes per state.

To reach a more compact state representation, the sequence of 25 numbers could be seen as a single number written in ternary basis. Hence each state corresponds to a unique integer in $\{0, \dots, 3^{25} - 1\}$. Since $\log_2 3^{25} \approx 40$ bit, a single 8 byte variable is sufficient to store a state.

This representation offers some decisive advantages. It enables very fast computation of all basic operations with bitwise operators.

In order to strongly-solve the game, we need to record the outcome of all possible states. Three possible outcomes (Win, Loss or Draw) means that 2 bits are necessary to store each outcome. Using a typical state-value associative array requires at least $64bits + 2bits$ per entry, which sums up to more than $6.5TB$. It is possible to enumerate all possible states in a pre-determined order creating a

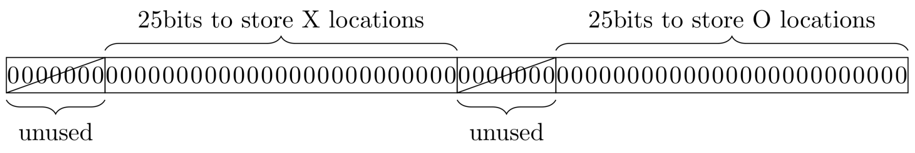

Figure 5: State representation

bijection between the set of all 3^{25} states and the set of natural numbers $\{0, \dots, 3^{25} - 1\}$ storing the outcomes in a giant bit array. Again, $2bits$ per entry yields a total size of $2 \cdot 3^{25} = 197GB$.

The solution to the memory occupation is storing only part of the states outcomes in RAM using backward induction. Introducing the concept of class $C_{x,o}$ as the class of states having x Xs and o Os, to evaluate a generic state $C_{x,o}$ we just need to know the outcomes of $C_{o,x}$ and $C_{o,x+1}$, since an available move could move an empty tile that becomes X or could move an X tile, swapping the players we obtain the two possible state classes. The largest class is $C_{8,8}$ which contains $\binom{25}{8} \cdot \binom{17}{8} \approx 2.6 \cdot 10^{10}$ states, which corresponds to $6.1GB$, which multiplied by 4 (class outcomes, reverse class outcomes, next class outcomes, reverse next class outcomes) reaches an approximatively maximum memory occupation of $24.4GB$.

Another hidden problem here is how to create a bijection between states of class $C_{x,o}$ and the set of natural nummbers $\{0, \dots, \binom{25}{x} \cdot \binom{25-x}{o} - 1\}$.

In order to do that the 64-bit state variable are split into two 32-bit variables. For the variable representing the location of the Os, we remove the digits where Xs are located, and then shift remaining digits to the right as shown in Figure 6.

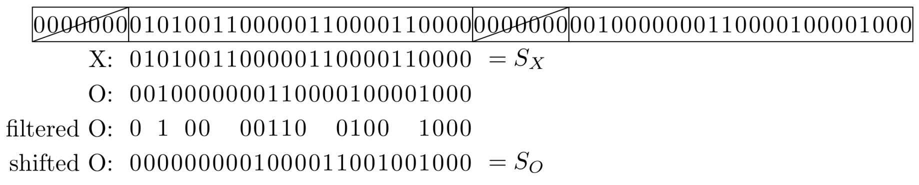
0000000
0101001100000110000110000
0000000
0010000000110000100001000
X: 0101001100000110000110000 = S_X
O: 0010000000110000100001000
filtered O: 0 1 00 00110 0100 1000
shifted O: 0000000001000011001001000 = S_O

Figure 6: Sx, So extraction from state representation

The index of the state whitin its class is defined as:

$$class\_state\_index = ord(S_x) \cdot \binom{25 - x}{o} + ord(S_o) \tag{5}$$

where $ord(\cdot)$ is an order function that returns the order of a number among numbers with the same population count (Hamming weight).

In order to obtain fast computation $pop(\cdot)$ and $ord(\cdot)$ functions are precomputed.

I'm reporting here the algorithms used in this solution.

Algorithm 1 shows the basic Value Iteration algorithm which is not feasible to run because of the amount of memory needed to store all states outcome.

Algorithm 2 shows Value Iteration algorithm using backward induction making feasible to compute outcomes of states belonging to class $C_{x,o}$ with the knowledge of outcomes of states belonging to classes $C_{o,x}$ and $C_{o,x+1}$.

A parallelization technique is involved to speed up the computation. In particular for each time there is a computation over all states of a class, the states in that class are divided by the number of threads and each is responsible of its states. A particular attention is given to mutual access to shared resources using mutexes and conditional variables.

Algorithm 1 Value Iteration

```
 1: for all states $s$ do
 2:     if there is a line of Xs in $s$ then
 3:         outcome[$s$] $\leftarrow$ Win
 4:     else if there is a line of Os in $s$ then
 5:         outcome[$s$] $\leftarrow$ Loss
 6:     else
 7:         outcome[$s$] $\leftarrow$ Draw
 8:     end if
 9: end for
10: repeat
11:     for all states $s$ such that outcome[$s$] = Draw do
12:         if at least one child of $s$ is Loss then
13:             outcome[$s$] $\leftarrow$ Win
14:         else if all children of $s$ are Win then
15:             outcome[$s$] $\leftarrow$ Loss
16:         end if
17:     end for
18: until no update in the last iteration
```

Algorithm 2 Backward induction with Value Iteration

```
 1: for $n = 25$ to 0 do
 2:     for $x = 0$ to $\frac{n}{2}$ do
 3:         $o \leftarrow n - x$
 4:         if $n < 25$ then
 5:             Load outcomes of classes $C_{x,o+1}$ and $C_{o,x+1}$
 6:             Compute outcomes of classes $C_{x,o}$ and $C_{o,x}$ using VI
 7:             Save outcomes of classes $C_{x,o}$ and $C_{o,x}$
 8:             Unload all outcomes
 9:         end if
10:     end for
11: end for
```

4.3 Results

The program was written in C++ and was run on a Ubuntu 22.04LTS machine equipped with 32GB of RAM and powered by a 16-core Intel Core i9 CPU. The computation for Quixo 4x4 lasted about 30 minutes, whereas for Quixo 5x5, it took about 60 hours (2 days and a half). Quixo 4x4 has produced about 10.3 MB of data while Quixo 5x5 produced 197 GB of data.

Here are shown some testing results in a 5x5 table ($N = 5$):

Agent	Opponent Agent	Win rate	Lose rate	Draw rate
QuixoIsSolvedAgent	MiniMaxAgent(2)	18.2%	0.0%	81.8%
QuixoIsSolvedAgent	MiniMaxAgent(3)	11.0%	0.0%	89.0%
QuixoIsSolvedAgent	QuixoIsSolvedAgent	0.0%	0.0%	100.0%

Table 5: Agent performance comparison with $N = 5$

Here are shown some testing results in a 4x4 table ($N = 4$):

Agent	Opponent Agent	Initial player	Win rate	Lose rate	Draw rate
QuixoIsSolvedAgent	MiniMaxAgent(2)	0	100.0%	0.0%	0.0%
QuixoIsSolvedAgent	MiniMaxAgent(2)	1	27.6%	72.4%	0.0%
QuixoIsSolvedAgent	MiniMaxAgent(3)	0	100.0%	0.0%	0.0%
QuixoIsSolvedAgent	MiniMaxAgent(3)	1	23.2%	76.2%	0.6%
QuixoIsSolvedAgent	QuixoIsSolvedAgent	0	100.0%	0.0%	0.0%
QuixoIsSolvedAgent	QuixoIsSolvedAgent	1	0.0%	100.0%	0.0%

Table 6: Agent performance comparison with $N = 4$

These results confirm all the results explained in the cited paper [ST20], and so Quixo 4x4 is a Active-Player-Win game while Quixo 5x5 is a Draw game, which means that two optimal players will play endlessly.

References

[ST20] Sébastien Tixeuil Yasumasa Tamura Satoshi Tanaka, Francois Bonnet. Quixo is solved. 2020.