Quixo Is Solved

Satoshi Tanaka, François Bonnet, Sébastien Tixeuil, Yasumasa Tamura

Quixo Is Solved

Satoshi Tanaka[1] François Bonnet[1] Sébastien Tixeuil[2] Yasumasa Tamura[1]

[1] Tokyo Institute of Technology, Tokyo, Japan
[2] Sorbonne Université, CNRS, LIP6, Paris, France

Abstract

Quixo is a two-player game played on a 5×5 grid where the players try to align five identical symbols. Specifics of the game require the usage of novel techniques. Using a combination of value iteration and backward induction, we propose the first complete analysis of the game. We describe memory-efficient data structures and algorithmic optimizations that make the game solvable within reasonable time and space constraints. Our main conclusion is that Quixo is a Draw game. The paper also contains the analysis of smaller boards and presents some interesting states extracted from our computations.

1 Introduction

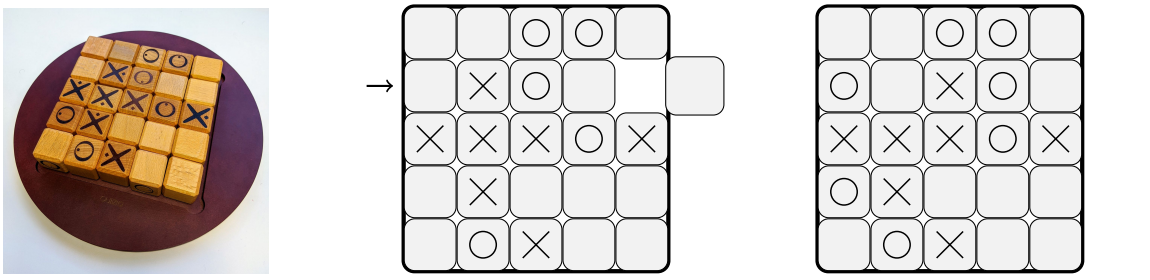
1.1 Quixo

Quixo is an abstract strategy game designed by Thierry Chapeau in 1995 and published by Gigamic [1, 2]. Quixo won multiple awards, both in United States [3, 4, 5, 6] and in France [7, 8]. While a four-player variant exists, Quixo is mostly a two-player game that is played on a 5×5 grid, also called *board*. Each grid cell, also called *tile*, can be empty, or marked by the symbol of one player: either X or O.

At each turn, the active player first *(i)* takes a tile – empty or with her symbol – from the border (*i.e.* excluding the 9 central tiles), and then *(ii)* inserts it, with her symbol, back into to the grid by pushing existing tiles toward the hole created by the tile removal in step *(i)*. The winning player is the first to create a line of tiles all with her symbol, horizontally, vertically, or diagonally. Note that if a player creates two lines with distinct symbols in a single turn, then the opponent is the winner.

Figures 1a and 1b show the real game and our corresponding representation. Figure 1c depicts the resulting board after a valid turn by player O from the board depicted in Figure 1b: player O first *(i)* takes the rightmost (empty) tile of the second row, and then *(ii)* inserts it at the leftmost position shifting the other tiles of this second row to the right. A complete game on a smaller 4×4 board is given in Figure 8 on page 12.

Quixo bears an immediate resemblance with some classical games such as Tic-Tac-Toe, Connect-Four, or Gomoku. However there are two major differences: *(1)* the board is "dynamic"; a placed X or O may change its location in subsequent turns, *(2)* the game is unbounded (in term of turns). The first point is



(a) Real game (b) Simplified illustration (c) After a valid move from player O

Figure 1: The two-player game of Quixo

what makes the game interesting to play: dynamicity makes Quixo very difficult for a human player to plan more than a couples of turns in advance. The second point raises immediately a natural scientific question about termination. Trivially, players could "cooperate" to create an infinite game, as official rules do not specify any terminating rules, such as the 50-move or the threefold repetition rules of chess. For example, by always taking the same tiles and having only one X and one O on the board. However, it remains unclear whether an infinite game exists when both players play to win.

1.2 Related work

From the seminal work on Nim [9] to more recent results on Poker [10] or Go [11], studying games has always been a prolific research topic. Considered as recreational mathematics, results were initially obtained by theoretical analysis and/or handmade computations. Later, in order to solve more complex games, it became necessary to use computer-assisted techniques. However, even for complex games, the outcome is sometimes known without any computations. Hex was proven by Nash to be a First-Player-Win game using a now-classic strategy-stealing argument [12]. Yet, it is still extremely complex to find optimal strategies; larger boards (up to 10×10) are regularly solved [13] but a winning strategy for the original 11×11 is still missing.

In 1988, Allen and Allis independently proved that Connect-Four is a First-Player-Win game [14]. Awari and Checkers were both proven to be Draw games, respectively by Romein and Bal in 2003 [15], and by Schaeffer et al. in 2007 [16]. Among connection games, it is worth mentioning that Gomoku and Renju have also been proved to be First-Player-Win games [17, 18].

Note that aforementioned results about Go and Poker are quite different. These works propose excellent strategies, well-above human capacities, but they do not consist in a proper solution of either game. The exact outcome and optimal strategies remain unknown. In contrast, here, we are interested in solving Quixo with the usual meaning of obtaining exact results, similar to Connect-Four, Awari, Checkers.

1.3 Objectives and challenges

Our main goal is to solve Quixo, which means finding the optimal outcome of the game assuming perfect players. As for any combinatorial game, there are only three possible outcomes: first-player-Win, second-player-Win, or Draw. While a second-player-Win seems unlikely, there is no easy observation that would permit to discard it.[1] It is improbable to obtain analytical results, so we focus on computing this optimal outcome and the corresponding optimal strategies. More precisely, we are looking for outcomes of all states, not only from the initial state. In game terminology, this is called strongly-solving the game. Finally, in addition to the real 5×5 game of Quixo, we also analyze a variant using 4×4 grid.

Even on the 5×5 grid, Quixo's game "tree" is not extremely large, with respect to other games. The number of positions is upper bounded by $2 \cdot 3^{25} \approx 1.7 \cdot 10^{12}$ configurations – 2 possibilities for the active player, and 3 options for each cell in the grid. These numbers are in a similar order of magnitude as the numbers of positions in Connect-Four, which was solved 30 years ago [14].

However, the game "tree" of Quixo is very different from other similar game "trees". For most games, the "trees" are directed acyclic graphs (DAG), assuming the merging of identical positions reached from different histories. Conversely, for Quixo, the game "tree" contains cycles (again, assuming the same kind of position merging). Indeed, especially in the late game, when the grid is mostly full of Xs and Ox, most moves do not add symbols, only reorganize them. Therefore, a simzople minimax algorithm (with or without alpha-beta pruning) may never terminate.

As a consequence, instead of searching the game "tree" with a DFS algorithm (as done by minimax or alike algorithms), it is necessary to use a more costly approach. Ideally, one would like to analyze the whole game "tree", but it is currently impossible to store it all at once in memory on commodity hardware.

1.4 Results overview

Our solution involves a combination of backward-induction and value-iteration algorithms, implemented using a state representation that is both time and space efficient. Based on our computations, the

[1] A typical strategy-stealing argument cannot be applied, at least not in an obvious way.

regular 5×5 Quixo is a Draw. In more details, neither player has a winning strategy if both players play optimally, and the game continues forever. On smaller grids, the first player wins. Interestingly, on the 4×4 grid, it takes at least 21 moves (11 moves from the first player and 10 from the opponent) to win. Since $21 > 16$, it is always necessary to re-use existing tiles.

Outline. Section 2 presents some basic definitions and terminology used in the paper. Sections 3 and 4 describe respectively the data structures and the algorithms used to solve Quixo. Section 5 summarizes our findings and highlights some unexpected observations. It also includes a nice animation of an optimal game run. Finally, section 6 concludes the paper with a list of open problems.

2 Preliminaries

By convention, the *first player* is player X and the *second player* is player O. The *board* corresponds to the 25 tiles and the *active player* denotes the player playing next. Note that, contrarily to TTT or Connect4, the active player cannot be deduced automatically from a given board. Therefore a *state* of the game consists of a board and an active player. The *initial state* is the empty board (that is, the board with 25 empty tiles) with player X as active player.

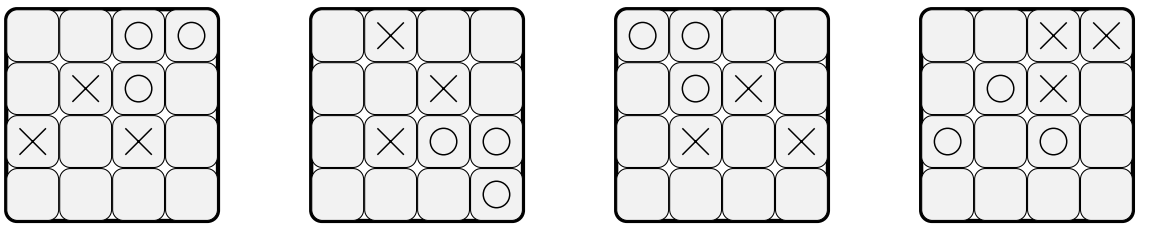
A state is *terminal* if its board contains a line of Xs or Os tiles. The *children* of a given state are all states obtained by a move of the active player. A terminal state has no children since the game is over and there is no valid moves. The *parents* of a state are defined analogously. The set of states and parent-child relations induce the *game graph* of Quixo (referred to as the game "tree" in the prequel). As mentioned earlier, this graph is neither a tree nor an acyclic graph.

Outcomes. 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). The outcome of a terminal state is trivially defined. 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.

Symmetries and swapping. Rotating or mirroring the board does not change the state value. Therefore states can be grouped in equivalence classes. Said differently, states with symmetrical boards can be merged into a single node in the game graph. This optimization divides approximately by eight the number of states: four being due to rotations, and two to vertical mirroring. Note that the horizontal mirroring is the same as vertical mirroring and 180° rotation. Also, swapping the active player and flipping all Xs and Os to Os and Xs respectively creates a new equivalent state. Figure 2 illustrates these notions. All four states are equivalent.

In the remaining of the paper, all states have X as active player. By an abuse of notation, we then identify the state and its board, omitting the active player.



Active player: X (a) State S1 | Active player: X (b) State S2 (rot 90°) | Active player: X (c) State S3 (mirror) | Active player: O (d) State S4 (swap X–O)

Figure 2: Equivalent states with respect to symmetries and swapping

3 Solving Quixo – data structures

This section considers only 5×5 Quixo but explanations can easily be adapted for smaller grids.[2] First we describe our memory efficient representation of states in memory. Then we focus on the more general problem of storing intermediate results.

3.1 Optimized state representation

Naive approaches. The most natural representation of a state in memory is probably to use a 1-dimensional (or 2-dimensional) array of 25 elements where each entry takes values from the set $\{0, 1, 2\}$ to map $\{\text{empty}, \text{X}, \text{O}\}$. Assuming a typical 1-byte char element, such a representation requires 25 bytes per state.

Of course, using a whole byte to store only 3 values wastes space. Instead, 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 $\{0, \ldots, 3^{25} - 1\}$. Since $\log_2(3^{25}) \approx 40$, a single 8-byte variable is sufficient to store a state. With respect to memory space, this representation is optimal. However, all basic operations on states are costly: checking if there is a line of Xs or Os, executing a move, etc.

Our optimized approach. Still using only 8 bytes, we represent a state as depicted on Figure 3. The 25 first least-significant bits indicate the location of Os on the board. After skipping 7 bits, the 25 following bits indicate the location of Xs. For example, the state of Figure 1b is represented in Figure 4.

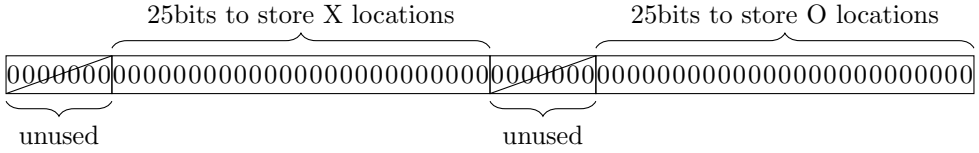


Figure 3: State representation in 64bits (LSB on the right)

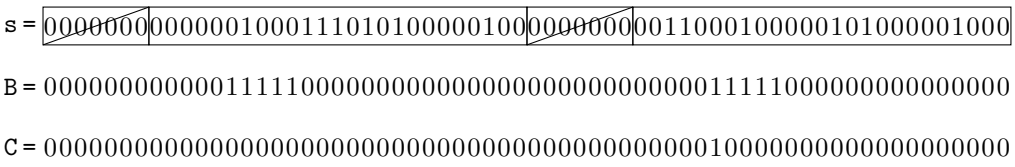


Figure 4: Our 64bits representation of the state of Figure 1b and constants `B` and `C` used to compute the move creating the state of Figure 1c using `(((s & B) >> 1) & B) | (s & ~B) | C`

This representation offers some decisive advantages. It enables very fast computation of all basic operations. Given a state `s`, and some appropriate pre-computed constants (see Figure 4), all the following operations can be done efficiently (assuming `<<`, `>>`, `&`, `|`, and `~` denote the classical bitwise "left shift", "right shift", "and", "or", and "not", respectively):

- Swapping the players: `s << 32 | s >> 32`
- Checking the existence of a tile at a given location:
  `s & A != 0`, with a different `A` for each pair of tile and symbol.
- Checking the existence of a given line of Xs or Os:
  `s & A == A`, with a different `A` for each pair of line and symbol.
- More interestingly, moves can also be computed quickly (with different `B` and `C` for each move):

[2] Note that larger grids cannot be solved using our state representation: 128bits variables would be necessary.

Left-pushing move: `(((s & B) << 1) & B) | (s & ~B) | C`
Right-pushing move: `(((s & B) >> 1) & B) | (s & ~B) | C`
Down-pushing move: `(((s & B) >> 5) & B) | (s & ~B) | C`
Up-pushing move: `(((s & B) << 5) & B) | (s & ~B) | C`,

The formulas are valid both when considering a marked tile or an empty tile. Moreover, it also allows computing the previous states.

Unfortunately, rotations and symmetries are still costly to compute. In fact, we believe that there is no efficient way to compute rotations with a compact data structure. Based on our observations, it is faster to avoid symmetry optimizations, and simply compute independently values for all symmetrical states. In the next section, we thus investigate how to store the outcomes of 3^{25} states.

3.2 Optimized storage of results

Using our optimized state representation, computations can be done quickly. It remains to consider the problem of storing the outcome of each state. Indeed, in order to strongly-solve the game, we need to record the outcome of all possible states. Three possible outcomes (Win/Loss/Draw) means that 2 bits are necessary to store each outcome. Using a typical (state: value) associative array requires at least 64 bits + 2 bits per entry, which sums up to more than 6.5TB.[3]

As mentioned earlier, it is possible to enumerate all possible states in a pre-determined order. It is therefore natural to only store the outcomes in a (giant) bit array. Again, 2 bits per entry yields a total size of $2 \cdot 3^{25} = 197$GB. Although more reasonable, renting a server with 200 GB of RAM may still require a significant investment. We further reduce memory requirements.

The obvious solution is to avoid storing all outcomes at the same time in RAM. Using backward induction (see Section 4), we only need to have a subset of already computed values to compute the new outcomes. For example, to compute all states containing 10 Xs and 8 Os with 8 Xs and 10 Os, it is sufficient to know the (inductively computed) outcomes of states containing either 8 Xs and 11 Os, and 10 Xs and 9 Os.[4] Therefore we partition the 3^{25} states based on the number of Xs and Os. Let $\mathcal{C}_{x,o}$ denotes the class of states containing x Xs and o Os.

The largest class is $\mathcal{C}_{8,8}$ which contains $\binom{25}{8} \cdot \binom{17}{8} \approx 2.6 \cdot 10^{10}$ states. Using 2 bits per state, it corresponds to ≈ 6.1GB of RAM. Since we can implement our algorithm using at most two classes loaded in memory at once, it becomes possible to solve the game on a more typical 16GB-RAM computer. This partitioning seems easy but there is an hidden problem:

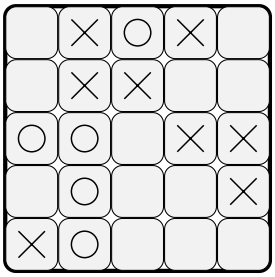
- Creating a bijection between the set of all 3^{25} states and the set of natural numbers $\{0, \ldots, 3^{25}-1\}$ is straightforward and "fast enough" to compute (in both directions). As explained earlier, one can see the 25 cells as the 25-digit ternary representation of a number (0 for empty, 1 for X, and 2 for O).
- Creating a bijection between $\mathcal{C}_{x,o}$ and the set of natural numbers $\left\{0, \ldots, \binom{25}{x} \cdot \binom{25-x}{o} - 1\right\}$ is less straightforward and more difficult to implement, especially in an efficient way. We further describe our scheme in the remaining of the section.

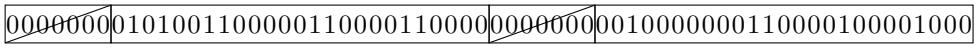
Bijections computation. Let us explain with an example the bijection we use in our implementation. Consider the state depicted in Figure 5 and its state representation (as defined earlier). Let us split the 64-bit variable into two 32-bit variables. For the variable representing the location of the Os, we remove the digits where Xs are located, and then shift remaining digits to the right (see example). In the class $\mathcal{C}_{8,5}$, the index of this state is then

$$ord(S_X) \cdot \binom{25-8}{5} + ord(S_O), \tag{1}$$

[3] In practise, this amount of space is likely much higher to to memory alignment. Furthermore, typical data structure would likely use 128 bits per entry, and most likely a substantial overhead (*e.g.* pointers). So, a more realistic estimation for full storage is in the order of 15TB.

[4] Note that this is a simplified explanation. Indeed, as mentioned in Section 2, we consider only states with X as active player. In our real computations, we need to swap players after each move, so children of states containing 10 Xs and 8 Os may have 8 Xs and 10 or 11 Os.





X: 0101001100000110000110000 $= S_X$

O: 0010000000110000100001000

filtered O: 0 1 00 00110 0100 1000

shifted O: 0000000001000011001001000 $= S_O$

Figure 5: A state in the class (8,5), its 64bits representation, and the steps to compute its index

where $ord(\cdot)$ is an order function that returns the order of a number among numbers with the same population count (aka. Hamming weight). For simplicity, the population count is returned by function $pop(\cdot)$.

For a given class $\mathcal{C}_{x,o}$, there are $\binom{25}{x} = ord(\underbrace{1\dots1}_{x}\underbrace{0\dots0}_{25-x}) + 1$ valid positions of the Xs in the grid. After placing the Xs, there remain $\binom{25-x}{o} = ord(\underbrace{0\dots0}_{x}\underbrace{1\dots1}_{o}\underbrace{0\dots0}_{25-x-o}) + 1$ valid positions of the Os in the grid. Combining these two observations leads to Equation 1 and guarantees that each state is associated to a unique natural number, and vice-versa.

In order to obtain fast computations, functions $pop(\cdot)$ and $ord(\cdot)$ are pre-computed and kept in memory at all time. Example values are presented in Table 1. Overall, it takes less than a second to compute all such values, and it uses approximately 0.5GB of RAM.[5]

Table 1: Population count $pop(x)$ and order $ord(x)$ of the first integers x

x	$pop(x)$	$ord(x)$	x	$pop(x)$	$ord(x)$
00000000	0	0	00001100	2	5
00000001	1	0	00001101	3	2
00000010	1	1	00001110	3	3
00000011	2	0	00001111	4	0
00000100	1	2	00010000	1	4
00000101	2	1	00010001	2	6
00000110	2	2	00010010	2	7
00000111	3	0	00010011	3	4
00001000	1	3	00010100	2	8
00001001	2	3	00010101	3	5
00001010	2	4	00010110	3	6
00001011	3	1	00010111	4	1

[5]There are 2^{25} input values x and both $pop(x)$ and $ord(x)$ can be stored in a 32-bit variable. Hence, we obtain a total of $2^{25} \cdot 32 \cdot 2 = 256$MB. It is also necessary to store the inverse function, namely the unique x corresponding to a given pair $(pop(x), rd(c))$. For efficiency, the x is stored in a 64-bit variable (to match the size of states), which makes a total of $2^{35} \cdot 64 = 256$MB too.

4 Solving Quixo – algorithms

4.1 Basic algorithms

Value iteration. After designing data structures, we now present our algorithms. As explained in Section 1.3, due to cycles in the game "tree", minimax algorithm cannot be used for Quixo. The most natural algorithm for solving such games is the *Value Iteration* (VI) algorithm (recalled in Algorithm 1). In the pseudo-code, the children of a state denote the set of states that are reachable after one move.[6] This algorithm follows closely the definition of outcome provided in Section 2:

- Lines 1 to 7 deal with terminal states: a state is an immediate Win or Loss when there is a line of one of the symbols. Since we consider only positions where X is the active player, the existence of lines of Xs should be checked first. Indeed, by the rules, if there are lines of both symbols, the last player to move loses the game.
- Lines 8 to 14 iterate over the set of states until converging to a stable outcome. At the end of the computation, there may remain some Draw states.

Algorithm 1 Value Iteration (VI)

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

Backward induction. Quixo cannot be solved directly applying this algorithm since it would require to store all outcomes at once in RAM (and thus would be too slow due to memory caching). Fortunately, we can use the classes $\mathcal{C}_{x,o}$ we defined in Section 3.2. Indeed, for any state of $\mathcal{C}_{x,o}$, its children belong to $\mathcal{C}_{o,x} \bigcup \mathcal{C}_{o,x+1}$ (due to player swap after each move). Thus, it becomes possible to compute all outcomes of $\mathcal{C}_{x,o} \bigcup \mathcal{C}_{o,x}$ using only $\mathcal{C}_{x,o+1} \bigcup \mathcal{C}_{o,x+1}$. Starting from states with 25 Xs or Os, and using *backward induction*, we can compute all outcomes having only four classes of states in RAM at any given moment. The corresponding pseudo-code is given in Algorithm 2.

Algorithm 2 Backward Induction using VI internally

```
for n = 25 to 0 do
  for x = 0 to ⌈n/2⌉ do
    o ← n − x
    if n < 25 then
      Load outcomes of classes C_{x,o+1} and C_{o,x+1}
    Compute outcomes of classes C_{x,o} and C_{o,x} using VI
    Save outcomes of classes C_{x,o} and C_{o,x}
    Unload all outcomes
```

[6] Some practical implementation details are omitted from pseudo-code. For example, since we only consider states with active player X, we need to swap the tiles/players after each move.

This algorithm is likely to be able to solve Quixo, but, in practice, it is too slow to run on commodity hardware. Unfortunately, it is difficult to evaluate precisely its complexity because it depends on the number of internal (value) iterations, which is itself difficult to predict. The topology of the game "tree" has a strong impact on the required number of iterations to converge.

4.2 Optimized algorithms

We propose two algorithmic enhancements that significantly reduce the computation time.

4.2.1 Use parent link

To be a Win state, there should be at least one Loss child. Reversing this statement, we obtain that every parent of a Loss state is a Win state. This simple observation can be used to improve the computation. As soon as a state is found to be a Loss, we can compute all its parents and update their outcome to Win. Eventually they would have been updated to Win in Algorithm 1, but updating them immediately makes it possible to skip searching if states are Win (Lines 10 to 11) and may allow other states to be updated faster too. Note that parents of a given state can also be computed efficiently using a similar method as for computing its children (see Section 3.1). The pseudo-code is given in Algorithm 3. Note that the optimization is also used inside the internal iterations.

Algorithm 3 Update outcomes of $\mathcal{C}_{x,o}$ using terminal states

```
1: for all states s ∈ C_{x,o} do
2:   if there is a line of Xs in s then
3:     outcome[s] ← Win
4:   else if there is a line of Os in s then
5:     outcome[s] ← Loss
6:     for all parents p ∈ C_{o,x} of s do
7:       outcome[p] ← Win
```

4.2.2 Use Win-or-Draw outcome

Internal iterations require checking the outcomes of all children of a given state (see Lines 10 and 12 of Algorithm 1). Some of the children belong to already inductively-computed classes, while the others belong to classes currently being computed. More explicitly, for a state $s \in \mathcal{C}_{x,o}$, some children belong to $\mathcal{C}_{o,x}$ and some belong to $\mathcal{C}_{o,x+1}$. The outcome of this latter class has been already computed inductively. It is possible to check them only once by introducing a *new temporary outcome*: WinOrDraw. For a given state $s \in \mathcal{C}_{x,o}$, among the children of s in $\mathcal{C}_{o,x+1}$:

- If there is at least one Loss state, then it is possible to immediately decide that s is a Win state (as before).
- If all children are Win states, then the outcome of s is initialized to Draw (as before). It means that it is still necessary to check children of s in $\mathcal{C}_{o,x}$ to decide whether s is a Win, Draw, or Loss.
- If there is at least one Draw state (and no Loss state), then the outcome of s is initialized to WinOrDraw. As for the previous case, it is still necessary to check children of s in $\mathcal{C}_{o,x}$ to decide the real outcome of s. However, we can already eliminate the Loss option since there is at least one Draw child.

The corresponding pseudo-code is given in Algorithm 4. Note that the test of Line 6 exists to avoid overriding a Win outcome already computed by Line 4.

4.2.3 Complete algorithm

Combining value iteration, Backward induction, and our two optimizations, we obtain the complete algorithm we used to solve Quixo. It is summarized in Algorithm 5. Using the additional WinOrDraw outcome allows for an additional subtle optimization. Inside internal iterations, the algorithm looks only

Algorithm 4 Update outcomes of $\mathcal{C}_{x,o}$ using inductively-computed outcomes of $\mathcal{C}_{o,x+1}$

1: **for all** states $c \in \mathcal{C}_{o,x+1}$ **do**
2: **if** outcome$[c] = Loss$ **then**
3: **for all** parents $s \in \mathcal{C}_{x,o}$ of c **do**
4: outcome$[s] \leftarrow Win$
5: **else if** outcome$[c] = Draw$ **then**
6: **for all** parents $s \in \mathcal{C}_{x,o}$ of c s.t. outcome$[s] = Draw$ **do**
7: outcome$[s] \leftarrow WinOrDraw$

for Loss states (*i.e.*, states with only Win children). Win states are immediately updated when a child outcome is finalized as Loss. Therefore, it is not necessary to check if a state is Win. Since WinOrDraw states cannot become Loss states, there is no need to ever look at their children, hence the condition in Line 12.

Algorithm 5 Complete algorithm used to solve Quixo 5×5.

1: **for** $n = 25$ **to** 0 **do**
2: **for** $x = 0$ **to** $\lceil n/2 \rceil$ **do**
3: $o \leftarrow n - x$
4: **for all** states $s \in \mathcal{C}_{x,o} \bigcup \mathcal{C}_{o,x}$ **do**
5: outcome$[s] \leftarrow Draw$
6: Update outcomes of $\mathcal{C}_{x,o}$ using terminal states (Algo 3)
7: Update outcomes of $\mathcal{C}_{o,x}$ using terminal states (Algo 3)
8: **if** $n < 25$ **then**
9: Update outcomes of $\mathcal{C}_{x,o}$ using inductively-computed outcomes of $\mathcal{C}_{o,x+1}$ (Algo 4)
10: Update outcomes of $\mathcal{C}_{o,x}$ using inductively-computed outcomes of $\mathcal{C}_{x,o+1}$ (Algo 4)
11: **repeat**
12: **for all** states $s \in \mathcal{C}_{x,o} \bigcup \mathcal{C}_{o,x}$ such that outcome$[s] = Draw$ **do**
13: **if** all children $c \in \mathcal{C}_{x,o} \bigcup \mathcal{C}_{o,x}$ of s are Win **then**
14: outcome$[s] \leftarrow Loss$
15: **for all** parents $p \in \mathcal{C}_{x,o} \bigcup \mathcal{C}_{o,x}$ of s **do**
16: outcome$[p] \leftarrow Win$
17: **until** no update in the last iteration
18: **for all** states $s \in \mathcal{C}_{x,o} \bigcup \mathcal{C}_{o,x}$ such that outcome$[s] = WinOrDraw$ **do**
19: outcome$[s] \leftarrow Draw$

4.3 Parallelization

Although Algorithm 5 is sequential, it is possible to run some of its parts in parallel in order to take advantage of multi-cores found in current commodity hardware. Let us first observe that previously computed outcomes can be accessed concurrently safely since they are only read by multiple threads.

A high level parallelization is possible, by executing concurrently external iterations (Line 2 of Algorithm 5). On the positive side, no synchronization is needed on the outcome variables (as they are written by different threads in different memory locations). On the negative side, though, significantly more memory is needed (each thread may need as much memory as the sequential algorithm we presented in Algorithm 5). Furthermore, the number of states of $\mathcal{C}_{x,o}$ is large when $x = o$ or $x = o + 1$. Now, since the other parts need to wait until the large states parts are computed, the speed improvement may be marginal and not scale linearly in the number of threads. Due to these concerns, we chose not to use this (simple) option.

Our scheme for parallelization follows the rule: Replace each "for all states ... do" with a "divide by K" and execute K threads in parallel. Of course, since threads now use the same memory locations for write access, we must use synchronisation mechanisms (*e.g.* mutexes) to guarantee exclusive accesses to the currently-computed outcomes.[7]

[7]Implementation details: Due to our usage of `vector<bool>`, extra-caution was required. Accessing different elements

4.4 Deriving the optimal strategy

Using Algorithm 5, it is possible to compute all state outcomes. One may think that always choosing a Win action deterministically permits to win the game. Unfortunately, such a strategy does not guarantee winning since the Quixo game "tree" contains cycles. Hence, it is possible to enter a cycle where all states outcomes are Win, yet the game never finishes. Note that this behavior is unlike games that do not allow cycles in the game "tree" such as Connect-four.

It is possible to devise a probabilistically winning strategy by choosing a Win action uniformly at random: indeed, from any Win state, there exists a sequence of steps that does not belong to an infinite cycle. So, in an expected finite number of steps, the player wins.

However, the random strategy is not necessarily optimal with respect to the number of steps taken to win. Instead, we focus on the strategy to win in the minimum number of steps (assuming the loosing player always chooses the action that delays her loss the most). Now, to actually compute the steps to win or lose, we now store the number of steps to the final outcome, using a new *step* variable. The *step* variable is defined as follows:

- In a terminal state, *step* is 0,
- If the state is Win, *step* is one plus the minimum of the *step*s of Loss children,
- If the state is Loss *step* is one plus the maximum of the *step*s of Win children.

Previous algorithms can be adapted to compute this additional *step* variable. The Value Iteration algorithm changes to Algorithm 6.

Algorithm 6 Value Iteration with steps to Win or Loss

```
1: for all states s do
2:    if there is a line of Xs in s then
3:       outcome[s] ← Win
4:       step[s] ← 0
5:    else if there is a line of Os in s then
6:       outcome[s] ← Loss
7:       step[s] ← 0
8:    else
9:       outcome[s] ← Draw
10: i = 0
11: repeat
12:    for all states s such that outcome[s] = Draw do
13:       if at least one child of s exists whose outcome[c] is Loss and step[c] is i − 1  then
14:          outcome[s] ← Win
15:          step[s] ← i
16:       else if all children of s are Win then
17:          outcome[s] ← Loss
18:          step[s] ← min(children steps) + 1
19:    i + +
20: until no update in last iteration
```

of the same `vector<bool>` may lead to incorrect values: multiple bits are (typically) stored in the same byte, and only byte access can be guaranteed to be atomic by the processor.

5 Results

5.1 5×5 Quixo

Our main result is that Quixo is a Draw game. In other words, if perfect players play the game, no one wins, that is, the game never finishes.

Using a single-thread computation,[8] it takes approximately 19 500 minutes (just under two weeks) to obtain this result. Using multithreading, as described in Section 4.3, the running time shrinks to around 1 900 minutes (i.e. ≈ 32 hours) using up to $K = 32$ threads.

Additional observations. From our computations, we were able to extract some interesting data that we present in the sequel. Table 2 shows the total number of Win, Loss, and Draw states. As the number of Draw states is much smaller than those of Win or Loss states, it may come to a surprise that the initial state is Draw. However, we also look at the distribution of these states. Most of the Draw states are located near the top of the game "tree" (*i.e.*, with few marked tiles). Figure 6 (when X is next to play) depicts the percentages of Win, Loss, and Draw states for some classes of states. We display classes where the number of Xs and Os differ by at most 1. Intuitively, choosing the empty tile is a good strategy. The larger the number of marked tiles, the smaller the draw states percentage is. In other words, the latter part of the game is more complex.

Table 2: Total Win, Loss and Draw states numbers

Win	Loss	Draw
441,815,157,309	279,746,227,956	125,727,224,178

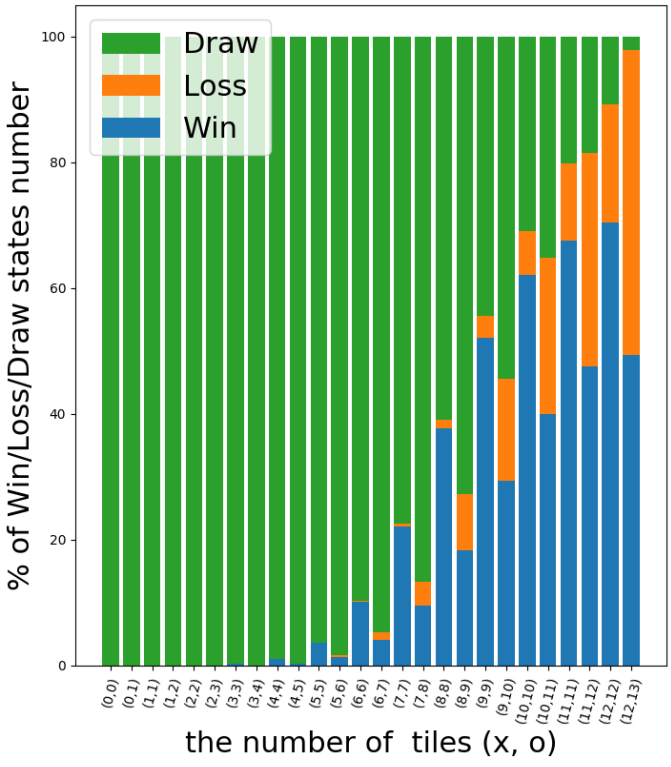
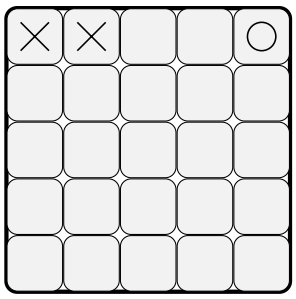


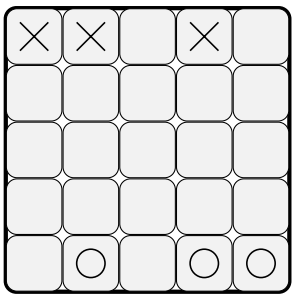
Figure 6: Percentage of Win, Loss, and Draw states for some classes $\mathcal{C}_{x,o}$. Player X is next to play. The board size is 5×5

[8] We used a Ubuntu 18.04LTS server equipped with 32GB of RAM and powered by a 16-core Intel Core i9-9960X CPU.

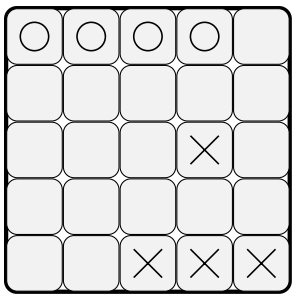
Some interesting states. Figure 7a shows an example where player X can win (obviously, in more than one step). In the previous step, player O chose a marked tile. Choosing a marked tile early in the game yields a big disadvantage. Moreover, the outcome of all reachable states in $\mathcal{C}_{2,1}$ with active player X is X wins. Figure 7b shows another example where player X can win. Until this state, both players only chose empty tiles. Figure 7c shows an example where player X loses. Such states are still complex for a human to understand the outcome.



(a) Player X can win. This state is one of those with the smallest numbers of tiles such that the outcome is not draw.



(b) Player X can win. This state is one of those with the smallest numbers of tiles such that the outcome is not draw and both players have chosen empty tiles only.



(c) Player X loses. The number of Xs and Os tiles is the same but the active player loses.

Figure 7: Some interesting states on the 5×5 board. Player X is next to play.

5.2 4×4 Quixo

Contrarily to the real game, the 4×4 variant is a Win for the first player. Intuitively, the smaller board makes it easier to create a line. However, winning is not trivial; it requires up to 21 moves when the opponent follows an optimal strategy. A complete optimal game is given in Figure 8.

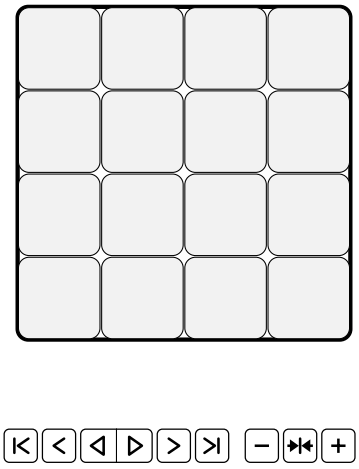
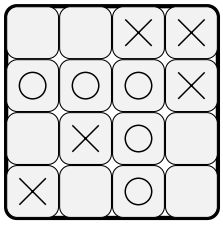
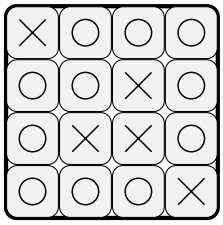


Figure 8: Optimal play in 4×4 Quixo. The animation is unfortunately not working with all pdf readers. It works with Adobe Acrobat Reader. Non-animated play in Appendix A

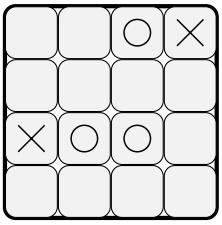
Additional observations. Some states are obviously not reachable, e.g. a state containing a single O not on an edge. Some other unreachable states are much less obvious, such as the state in Figure 9a.



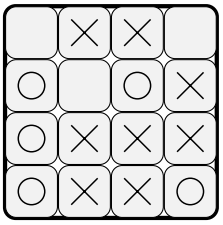
(a) Unreachable state. No previous state.



(b) Player X loses in 1 step.



(c) Player X loses in 22 steps.



(d) Draw state. O can come back to this state (or a symmetric one) with the next O step even if X plays optimally.

Figure 9: Some interesting states on the 4×4 board. Player X is next to play.

Globally, there are 41 252 106 reachable states, which accounts for 95.8% of the 3^{16} states. Therefore, ignoring unreachable states in the computation would not be significant.

Using the algorithm described in Section 4.4, Table 3 shows the number of steps to Win and Loss for optimal strategies. The winner selects an action that minimizes the number of steps to a Win, and the loser selects an action that maximizes the number of steps to a Loss.

Normally, a winner wins in an odd number of steps, and a loser loses in an even number of steps. However, some states yield an optimal player to lose in 1 step. One such example is shown in Figure 9b. In all next states, there is a line of O, so X loses in 1 step.

Another interesting result is that there are some states that lose in 22 steps although no state wins in 23 steps, and the initial state wins in 21 steps. Figure 9c is the example of a state to lose in 22 steps; only its symmetric states are the those that lose in 22 steps.

Table 3: Steps to Win and Loss of an optimal strategy

steps	Win states	Loss states	steps	Win states	Loss states
0	4,697,505	4,530,779	12	0	182,954
1	15,277,446	528	13	100,374	0
2	0	3,775,611	14	0	66,280
3	2,419,938	0	15	29,314	0
4	0	2,970,384	16	0	18,014
5	1,740,992	0	17	6,656	0
6	0	1,982,339	18	0	4,084
7	1,214,497	0	19	1,012	0
8	0	1,034,097	20	0	520
9	658,834	0	21	57	0
10	0	438,138	22	0	8
11	287,864	0	23	0	0
			total	26,434,489	15,003,736

6 Conclusions and open questions

To summarize, the official 5×5 Quixo is a Draw game; neither player can win. Smaller 3×3 and 4×4 variants are First-Player-Win games.[9]

Given that the 5×5 board is already a Draw game, one may expect larger instances to be Draw games too. We conjecture that it is the case, but we were not able to prove it.

In a different direction, one may be interested in the complexity of Quixo. Mishiba and Takenaga already studied the complexity of a generalization of Quixo [19]. They proved the game to be EXPTIME-complete. In the paper, they consider arbitrary large boards, but players still have to align only five identical symbols. Keeping the required line length equal to the board size may change the complexity.

[9]The 3×3 version is not discussed in this paper and is left to the reader. The first player wins in 7 moves.

Based on the previous generalization, a natural question arises; can the first player (or unlikely the second player) create a line of four symbols when playing on the 5×5 board? Changing the two lines losing rule into a winning rule may also change the global outcome.

Finally, a last research direction would be to compute human-playable optimal strategies. We strongly solved Quixo on 4×4 and 5×5 grids. However, playing an optimal strategy remains difficult for humans. Nim is mathematically solved [9], and following an optimal strategy is not so difficult for humans. Finding an optimal strategy for Quixo that can be remembered by humans is still an open problem.

References

[1] Quixo page on BoardGameGeek website. Accessed: 2020-02-09.

[2] Quixo page on Gigamic website. Accessed: 2020-02-09.

[3] Mensa Select Top 5 Best Games, 1995. in USA.

[4] Games Magazine "Games 100 Selection", 1995. in USA.

[5] Games Magazine "Best New Strategy Game", 1995. in USA.

[6] Parent's Choice Gold Award, 1995. in USA.

[7] As d'Or Festival International des Jeux, 1995. in Cannes, France.

[8] Oscar du Jouet-Toy Oscar, 1995. in Paris, France.

[9] Charles L Bouton. Nim, a game with a complete mathematical theory. *The Annals of Mathematics*, 3(1/4):35–39, 1901.

[10] Michael Bowling, Neil Burch, Michael Johanson, and Oskari Tammelin. Heads-up limit hold'em poker is solved. *Science*, 347(6218):145–149, 2015.

[11] D. Silver et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[12] Martin Gardner. *The Scientific American Book of Mathematical Puzzles and Diversions*. Simon and Schuster, 1959.

[13] Jakub Pawlewicz and Ryan B. Hayward. Scalable parallel dfpn search. In *Proceedings of the 8th International Conference on Computers and Games*, pages 138–150, 2013.

[14] Louis Victor Allis. A knowledge-based approach of connect-four. *ICGA Journal*, 11(4):165, 1988.

[15] John W Romein and Henri E Bal. Solving awari with parallel retrograde analysis. *Computer*, 36(10):26–33, 2003.

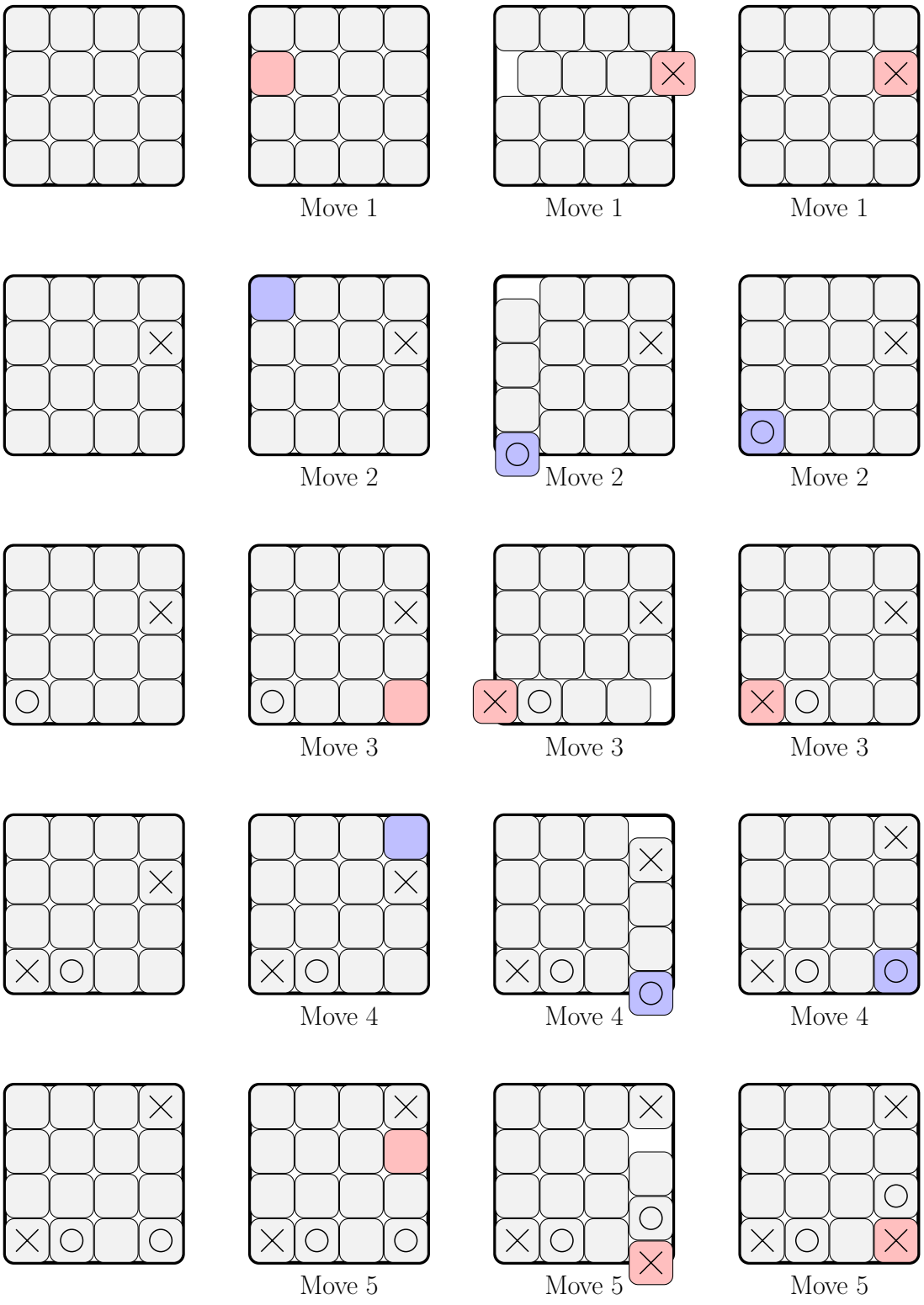
[16] Jonathan Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers is solved. *science*, 317(5844):1518–1522, 2007.

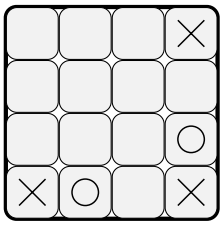
[17] L. V. Allis, H. J. van den Herik, and M. P. H. Huntjens. Go-moku solved by new search techniques. *Computational Intelligence*, 12(1):7–23, 1996.

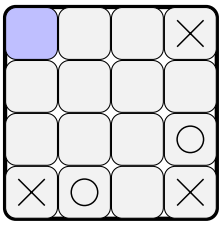
[18] János Wágner and István Virág. Solving renju. *ICGA journal*, 24(1):30–35, 2001.

[19] Shohei Mishiba and Yasuhiko Takenaga. 一般化QUIXOの計算複雑さ. *IEICE general conference*, page 26, 2017. (in Japanese).

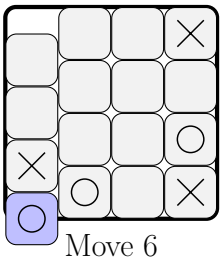
A Inanimate version of Figure 8



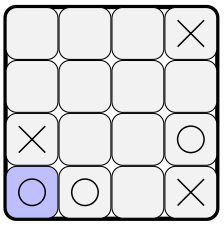




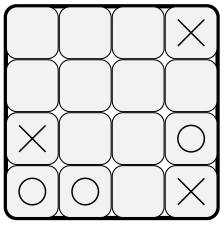
Move 6

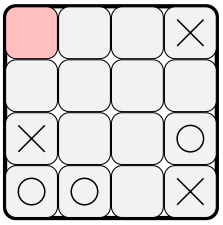


Move 6

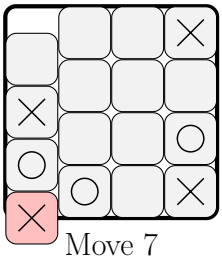


Move 6

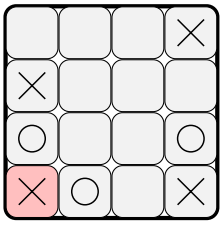




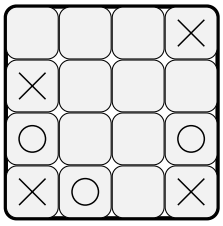
Move 7

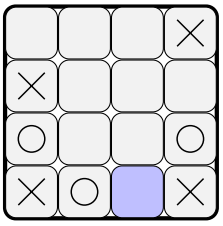


Move 7

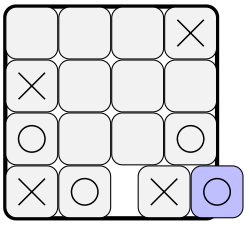


Move 7

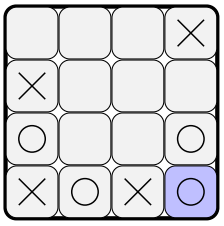




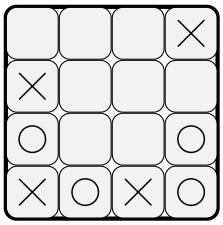
Move 8

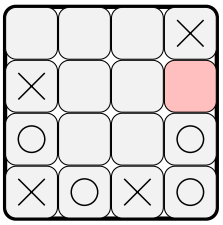


Move 8

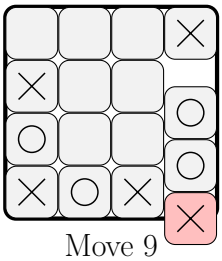


Move 8

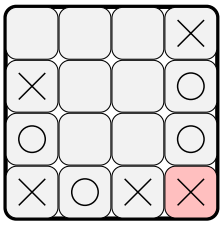




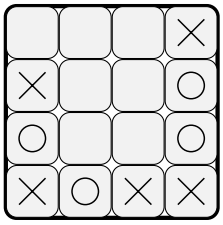
Move 9

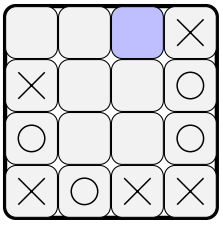


Move 9

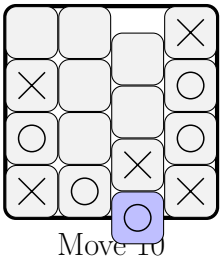


Move 9

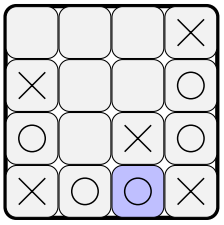




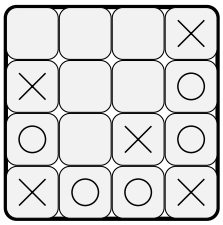
Move 10

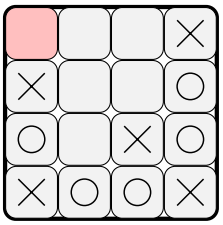


Move 10

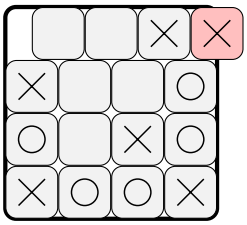


Move 10

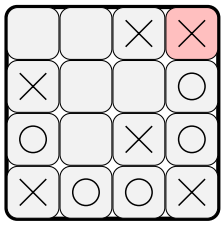




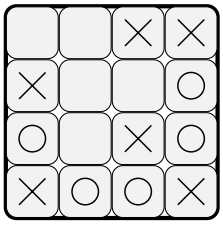
Move 11

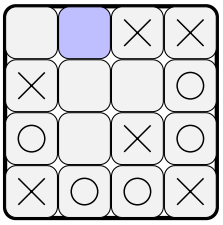


Move 11

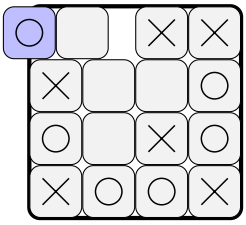


Move 11

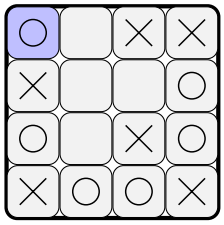




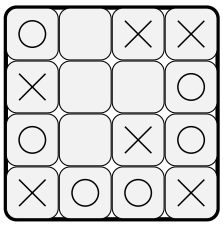
Move 12

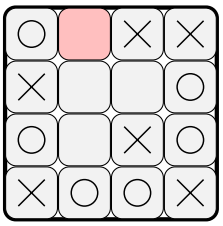


Move 12

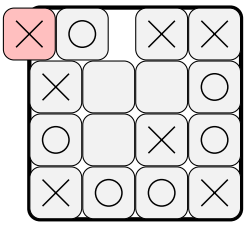


Move 12

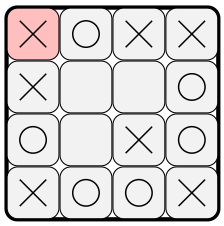




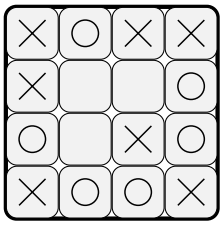
Move 13

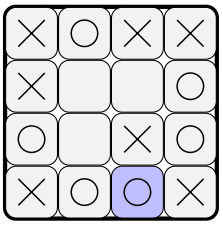


Move 13

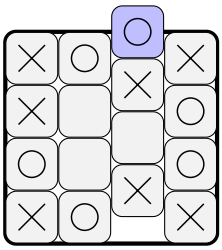


Move 13

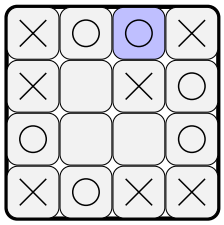




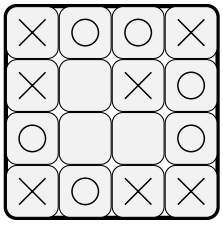
Move 14

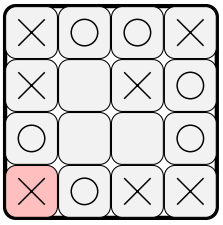


Move 14

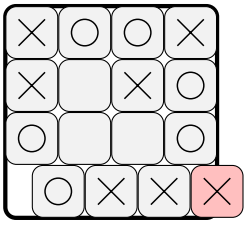


Move 14

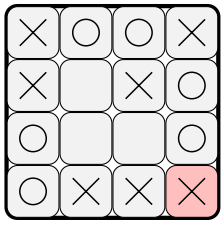




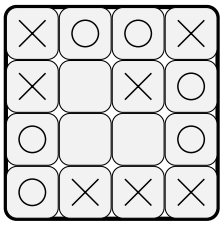
Move 15

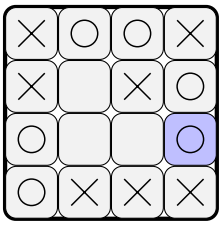


Move 15

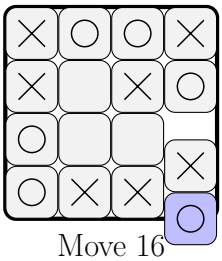


Move 15

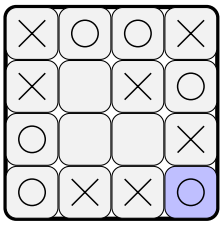




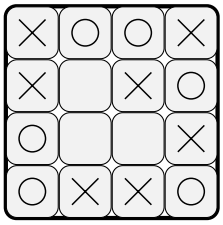
Move 16

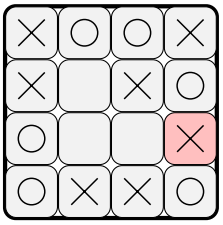


Move 16

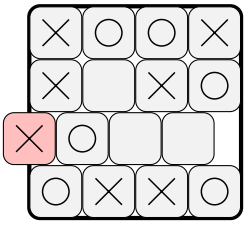


Move 16

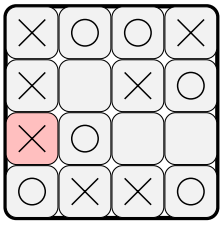




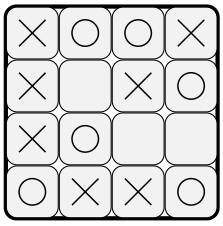
Move 17

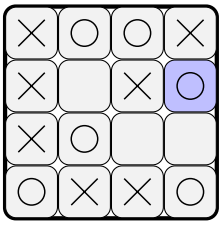


Move 17

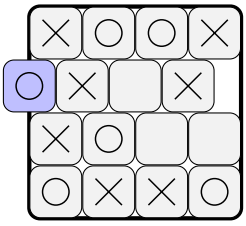


Move 17

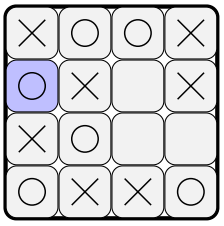




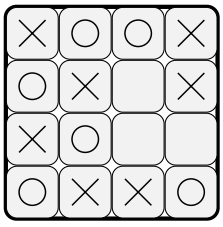
Move 18

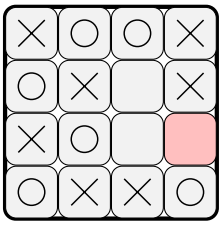


Move 18

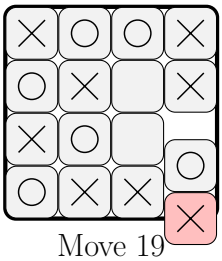


Move 18

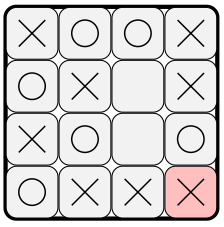




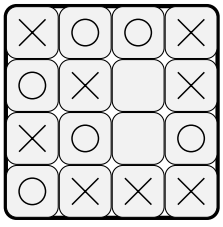
Move 19

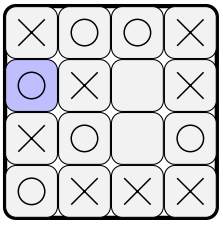


Move 19

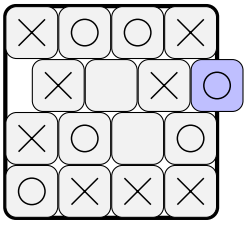


Move 19

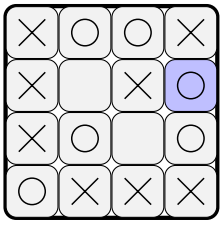




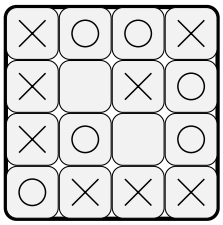
Move 20

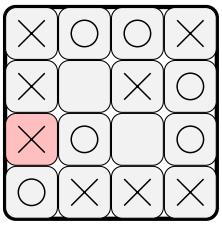


Move 20

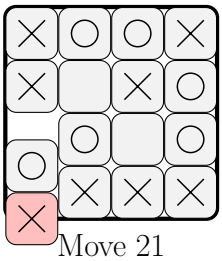


Move 20

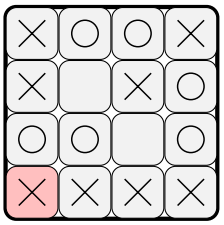




Move 21



Move 21



Move 21