Quixo Is Solved[⋆]

Satoshi Tanaka[1], François Bonnet[1], Sébastien Tixeuil[2], and 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. 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.

Keywords: Quixo · Strongly Solved · Draw Game.

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,[3] both in France and in United States. 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: 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 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 full game is given in Appendix A.

[⋆] A Japanese version of this paper was published and presented at the 25th Game Programming Workshop (GPW'20) [6]. S. Tanaka received the GPW Research Encouragement Award and the IPSJ Yamashita SIG Research Award for his work.

[3] As d'Or Festival International des Jeux (1995), Oscar du Jouet (1995), Mensa Select Top 5 Best Games (1995), Games Magazine "Games 100 Selection" (1995), Games Magazine "Best New Strategy Game" (1995), Parent's Choice Gold Award (1995). [2]

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 what makes the game interesting to play: dynamicity makes Quixo very difficult for a human player to plan more than a couple of turns in advance. The second point raises immediately a natural 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.

1.2 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 (e.g. strategy stealing argument) that would permit to discard it. 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, i.e. strongly-solving the game. In addition to the real 5×5 game of Quixo, we also analyze variants using 3×3 or 4×4 grids.

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. This number is in a similar order of magnitude as the numbers of positions in Connect-Four, which was solved 30 years ago [4].

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. 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 simple minimax algorithm (with or without alpha-beta pruning) may never terminate.

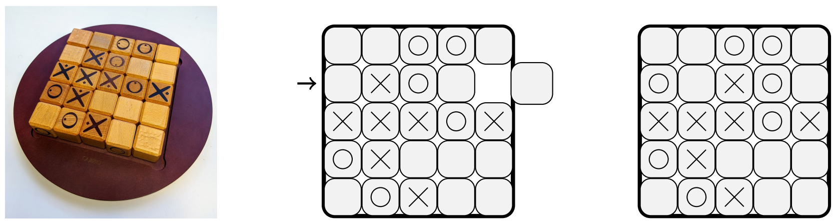

(a) Real game (b) Simplified illustration (c) After a move from O

Fig. 1: The two-player game of Quixo

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.

1.3 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 regular 5×5 Quixo is a Draw; 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.

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: 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. This optimization divides approximately by eight the number of states: four being due to rotations, and two to mirroring. 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.

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.

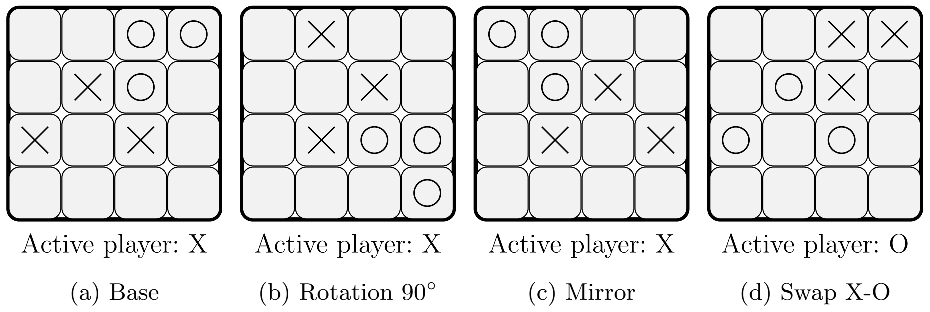


Fig. 2: Equivalent states with respect to symmetries and swapping

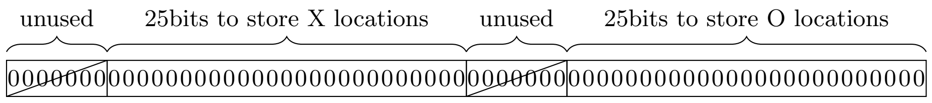


Fig. 3: State representation in 64bits (LSB on the right)

3 Solving Quixo – Data Structures

This section considers only 5×5 Quixo but explanations can easily be adapted for smaller grids. First we describe our memory efficient representation of states in memory. Then we focus on the more general problem of storing intermediate results. Due to space constraints, some explanations are omitted and can be found in the full paper [7].

3.1 Bitboard state representation

We use 64 bits to encode a state as depicted on Figure 3. 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 A, B, C (see Figure 4), all the following operations can be done efficiently (`<<`, `>>`, `&`, `|`, and `~` denoting usual bitwise operations):

- Swapping the players: `s << 32 | s >> 32`
- Checking the existence of a tile at a given location: `s & A != 0`
- Checking the existence of a given line of Xs or Os: `s & A == A`
- More interestingly, moves can also be computed quickly; e.g. for a down-pushing move: `(((s & B) >> 5) & B) | (s & ~B) | C`

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.

$$\texttt{s} = \boxed{\cancel{0000000}}\boxed{0000001000111010100000100}\boxed{\cancel{0000000}}\boxed{0011000100000101000001000}$$

$$\texttt{B} = 0000000000001111100000000000000000000000000011111000000000000000$$

$$\texttt{C} = 0000000000000000000000000000000000000000000010000000000000000000$$

Fig. 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`

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.[4]

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. 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. While this partitioning is easy, 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). 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 $\left\{0, \ldots, \binom{25}{x} \cdot \binom{25-x}{o} - 1\right\}$ is less straightforward and more difficult to implement in an efficient way. Due to space constraints, implementation details are omitted here and appear in the full paper

[4] In practice, this amount of space is likely much higher due to memory alignment. So, a more realistic estimation for full storage is in the order of 15TB.

4 Solving Quixo – Algorithms

4.1 Computing outcome

Value iteration After designing data structures, we now present our algorithms. As explained in Section 1.2, 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.[5] This algorithm follows closely the definition of outcome provided in Section 2.

Algorithm 1 Value Iteration (VI)

```
for all states $s$ do
    if there is a line of Xs in $s$ then
        outcome[$s$] $\leftarrow Win$
    else if there is a line of Os in $s$ then
        outcome[$s$] $\leftarrow Loss$
    else
        outcome[$s$] $\leftarrow Draw$
repeat
    for all states $s$ such that outcome[$s$] $= Draw$ do
        if at least one child of $s$ is Loss then
            outcome[$s$] $\leftarrow Win$
        else if all children of $s$ are Win then
            outcome[$s$] $\leftarrow Loss$
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.

This algorithm is likely to be able to solve Quixo, but, in practice, it is too slow. 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.

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

Algorithm 2 Backward Induction using VI internally

```
1: for $n = 25$ to $0$ do
2:     for $x = 0$ to $\lceil n/2 \rceil$ do
3:         $o \leftarrow n - x$
4:         if $n < 25$ then
5:             Load outcomes of classes $\mathcal{C}_{x,o+1}$ and $\mathcal{C}_{o,x+1}$
6:         Compute outcomes of classes $\mathcal{C}_{x,o}$ and $\mathcal{C}_{o,x}$ using VI
7:         Save outcomes of classes $\mathcal{C}_{x,o}$ and $\mathcal{C}_{o,x}$
8:         Unload all outcomes
```

Algorithmic optimizations We propose two algorithmic enhancements that significantly reduce the computation time. Due to space constraints, pseudocodes and detailed explanations are omitted and available in the full paper.

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).

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.

Complete algorithm. Combining value iteration, Backward induction, and our two optimizations, we obtain the complete algorithm we used to solve Quixo. Note that we also added some parallelization to reduce computation time (see [7]).

4.2 Deriving an optimal strategy

Using previously described algorithms, 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 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). 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. Since 5×5 Quixo is a Draw (Section 5), there is obviously no (optimal) winning strategies. Hence we used this modified algorithm only for the smaller variants of Quixo, namely on 3×3 and 4×4 grids.

4.3 Implementations

In order to increase confidence in our results, all computations (except the one described in Section 4.2) have been computed and verified with two independent implementations using slightly different optimizations. The complete source code is available in a public Github repository. [3]

5 Results

5.1 5×5 Quixo

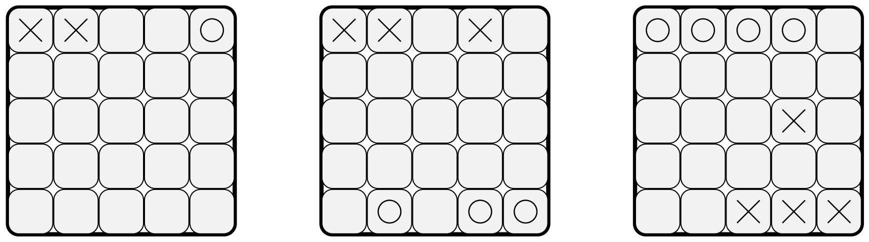
Our main result is that *Quixo is a Draw game*. In other words, if perfect players play the game, no one wins, that is, the game never finishes.

Using a single-thread computation,[6] it takes approximately 19 500 minutes (just under two weeks) to obtain this result. Using multithreading the running time shrinks to around 1 900 minutes (i.e. $\approx$ 32 hours) using up to 32 threads. For comparison, it takes around 13.5s and 0.1s for 4×4 and 3×3 grids respectively.

Some additional observations. Table 1 shows the total number of Win, Loss, and Draw states. As the number of Draw states is smaller than those of Win or Loss states, it may come to a surprise that the initial state is Draw. However, when looking at the distribution of these states, it appears that most of the Draw states are located near the top of the game “tree” (*i.e.*, with few marked tiles).

Figure 5 displays a few selected states whose results are not trivial.

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



(a) Player X can win. One of the states with the smallest numbers of tiles such that the outcome is not draw.

(b) Player X can win. One of the states 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.

Fig. 5: 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 Appendix A.

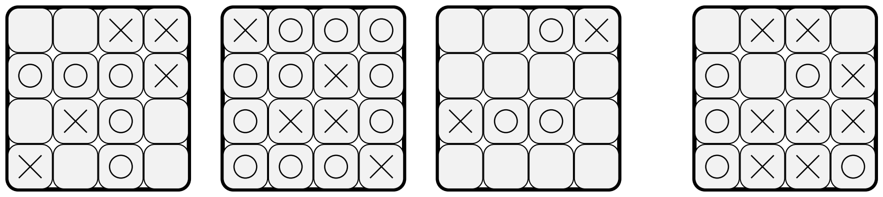
Some additional observations. Some states are obviously not reachable, e.g. a state containing a single O not on an edge. Some other unreachable states are much less obvious, such as the state in Figure 6a. Globally, there are 41 252 106 reachable states, which accounts for 95.8% of the 3^{16} states. Therefore, ignoring unreachable states in the computation would not be significant.

Using the algorithm described in Section 4.2, we computed the optimal strategies and the numbers of steps required to win/lose. Typically, 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 6b. 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 6c is the only example (and symmetric states) of a state to lose in 22 steps.

Table 1: Total Win, Loss and Draw states numbers

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



(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.

Fig. 6: Some interesting states on the 4×4 board. Player X is next to play.

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.[7] 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.

Mishiba and Takenaga proved that a generalization of Quixo is EXPTIME-complete [5]. They consider arbitrary large boards, but players still have to align only five identical symbols. Based on this 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.

References

1. Quixo page on BoardGameGeek website, https://boardgamegeek.com/boardgame/3190/quixo, accessed: 2021-09-17
2. Quixo page on Gigamic website, https://en.gigamic.com/game/quixo, accessed: 2021-09-17
3. Quixo source code repository, https://github.com/st34-satoshi/quixo-cpp
4. Allis, L.V.: A knowledge-based approach of connect-four. ICGA Journal **11**(4), 165 (1988)
5. Mishiba, S., Takenaga, Y.: 一般化QUIXOの計算複雑さ. IEICE general conference p. 26 (2017), (in Japanese)
6. Tanaka, S., Bonnet, F., Tixeuil, S., Tamura, Y.: Quixoの強解決. In: Proceedings of the 26th Game Programming Workshop. pp. 181–188 (2020), (in Japanese)
7. Tanaka, S., Bonnet, F., Tixeuil, S., Tamura, Y.: Quixo is solved (2020), https://arxiv.org/abs/2007.15895

[7] The 3×3 version is not discussed here and left to the reader (Win in 7 moves).

A Optimal play in 4 × 4 Quixo

Notes

– Unfortunately the animation does not work with all pdf readers. It should work with Adobe Acrobat Reader.
– LNCS probably cannot accept animations, so if the paper is accepted, it may be necessary to remove this animation. However, we still wanted to include it in the submission because it is fun to write animations in Latex!

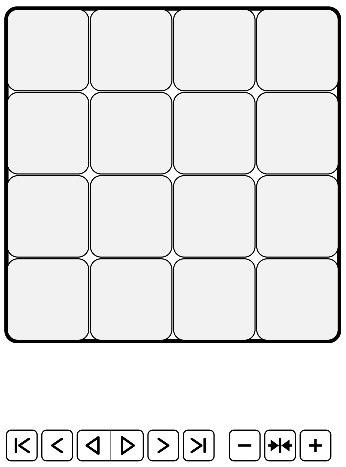


Fig. 7: Optimal play in 4 × 4 Quixo.