Constructing a forward-looking adaptation of retrograde analysis through the lens of Onitama

Elderbroek, Liam

Liam Elderbroek

Constructing a forward-looking adaptation of retrograde analysis through the lens of Onitama

Bachelor thesis

20th August 2025

Thesis supervisors: prof.dr. F.M. Spieksma (MI)
dr. R. van Vliet (LIACS)

Leiden University
Mathematical Institute
LIACS

Contents

Abstract

In this thesis we study the board game Onitama[Sat], focusing on methods of determining this game's optimal strategies in order to find strong and weak solutions for this game and generalisations thereof. We apply a general-purpose algorithm for strongly solving similar chess-like games called *retrograde analysis*[And+10], and prove its infeasibility to strongly solve Onitama by providing upper and lower bounds for the number of distinct game states it has to analyse. We introduce a new algorithm based on retrograde analysis that we formally prove is capable of weakly solving the game for all non-losing players instead, called *forward-looking* retrograde analysis. We also provide a useful symmetry that can be applied to the state space of this game to cut its size in half, significantly reducing retrograde analysis' runtime.

This analysis is done on a generalisation of the game with respect to the dimensions of the game board. The largest board size retrograde analysis is able to solve in under two hours is 3×4, with forward-looking retrograde analysis being able to increase this to 4×3. While making use of symmetries, forward-looking retrograde analysis is also able to solve the 3×4 board roughly 98.30 % faster than retrograde analysis.

1 Introduction

Deterministic sequential games that have perfect information lend themselves well to strategy analysis. Due to all information being known to all players, and all subsequent game states being based solely on the decisions the players make, the outcome of the game is entirely dependent on the *strategies* the players employ. These strategies can be analysed and compared with each other, forming a notion of *optimal* strategies, i.e., ones that guarantee the best possible outcome for the current player, regardless of what their opponents do.

Such optimal strategies have already been researched for a variety of games, such as chess[Ewe02], checkers[Sch+07], shogi[ISR02] and Go[Wer04]. These games have accordingly been divided into classes based on how difficult such strategies are to compute. In particular, there are separate definitions for games being *weakly* solved and *strongly* solved. A game is considered weakly solved, if for both players an optimal strategy can be found from the initial state covering all of their opponent's potential moves with reasonable computational resources. A game is considered strongly solved if an optimal strategy can be found covering all valid game states with reasonable computational resources. Checkers, for example, has been weakly solved[Sch+07] in 2007, while not yet having been strongly solved. Chess, however, remains unsolved—even weakly—to this day.

In this thesis, we research the solvability of the board game Onitama[Sat]. Onitama is a deterministic, sequential, two-player, perfect information strategy game, thus lending itself neatly to this kind of analysis. Methods such as *Monte Carlo tree search*[Arn21] and *theory of mind*[Boe24] have already been studied in relation to this game, in an attempt to find well-performing moves to play in real time. In this thesis, we instead analyse the game statically, and try to find optimal strategies that weakly and strongly solve generalisations of this game with respect to the size of the game board.

We analyse such strategies by means of state graphs: directed graphs where the nodes represent the different game states, and the edges represent the moves the current player in each of these source game states can perform to end up in the target state.

First, in Section 1, we explain the rules of the game, and define the relevant terminology for this thesis.

In Section 2 we discuss *retrograde analysis*[And+10], an existing algorithm that makes use of these state graphs, that is fit for strongly solving chess-like games. Using combinatorics and computational complexity theory, in Section 3 we then argue that the number of game states in Onitama's state graph grows so rapidly with the size of the board, that strongly solving this game becomes an increasingly less feasible task.

In Section 4, we then introduce a new algorithm developed for this thesis, based on retrograde analysis, fit for *weakly* solving chess-like games for all players that cannot lose under optimal play. These two algorithms are compared in runtime, as well as in how much of the state space they analyse.

Finally, in Section 5, we introduce a useful symmetry that can be applied on the game states that cuts the size of the state space in half when looking for optimal strategies.

1.1 Rules

We start by briefly explaining the rules of Onitama.

Onitama[Sat] is played with two players on a 5×5 board, with the players taking on teams *red* and *blue*. Each player has five pawns in their own team's colour: four students and one master. The players take on opposite ends of the board, with the starting positions of the pawns being in the row in front of the associated player, with the master taking the centre, as is depicted in Figure 1. In this, and all subsequent figures, the chess pawns represent students and the chess kings represent masters.

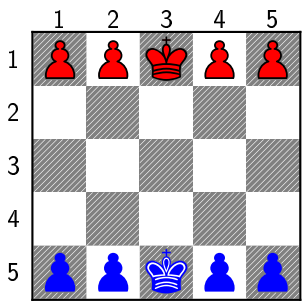

Figure 1: The initial state of the 5×5 board.

Five move cards are drawn randomly from the deck under a uniform distribution. In standard play this deck consists of sixteen unique move cards. Two of these drawn cards are placed in front of each player, and one is set aside—all face-up. These cards depict tiles relative to an origin. This origin can be associated with the current position of any of the current player's pawns. The offsets from the origin of a card to a tile on the same card describe how a player is allowed to move one of their pawns across the board. Each card also has an associated colour. The colour of the card that is set aside, dictates which player starts.

A player's turn ordinarily consists of moving one of their pawns in accordance with an offset on one of the cards they hold, after which the player has to exchange the used card for the one that is set aside. The turn is then passed on to the opponent.

A move is only allowed if the moved pawn's destination is still on the board, and is not on a tile already occupied by a pawn of the same colour. If a pawn is moved to a tile occupied by a pawn of opposing colour, the opposing-coloured pawn is *captured* and taken out of play.

Figure 2 shows an example of how a move can alter the game state. In this figure the two cards on each side depict the hands of each respective player, with the card next to the board being the set aside card.

If no valid move is possible, the player only has to exchange one of their cards with the one set aside, after which the player's turn ends—without having moved a pawn.

The game ends when a player captures the opponent's master, or when a player's master reaches the centre tile on the opposite end of the board. In both cases, the player who performed the final move is considered the winner.

In this game there is the potential for ending up in a state that has been visited before within the same game, meaning there is the potential for infinite play. For the sake of analysis we consider an infinite game as 'ending' in a draw. This happens if neither player ever makes a move to a state that would end the game.

1.2 Definitions

We start by giving more rigorous definitions of the concepts of this game through mathematical constructs, which we continue to use through the rest of this thesis to aid in theorems and proofs.

There are two *players* in the game, corresponding to the colours *red* and *blue*. For any colour c, let $\overline{c}$ denote its converse (red $\leftrightarrow$ blue). Both of these players take control of a number of *pawns*.

Definition 1 (player)**.** *A* player π *is either* red (r) *or* blue (b)*.* $\overline{\pi}$ *denotes* π*'s opponent.*

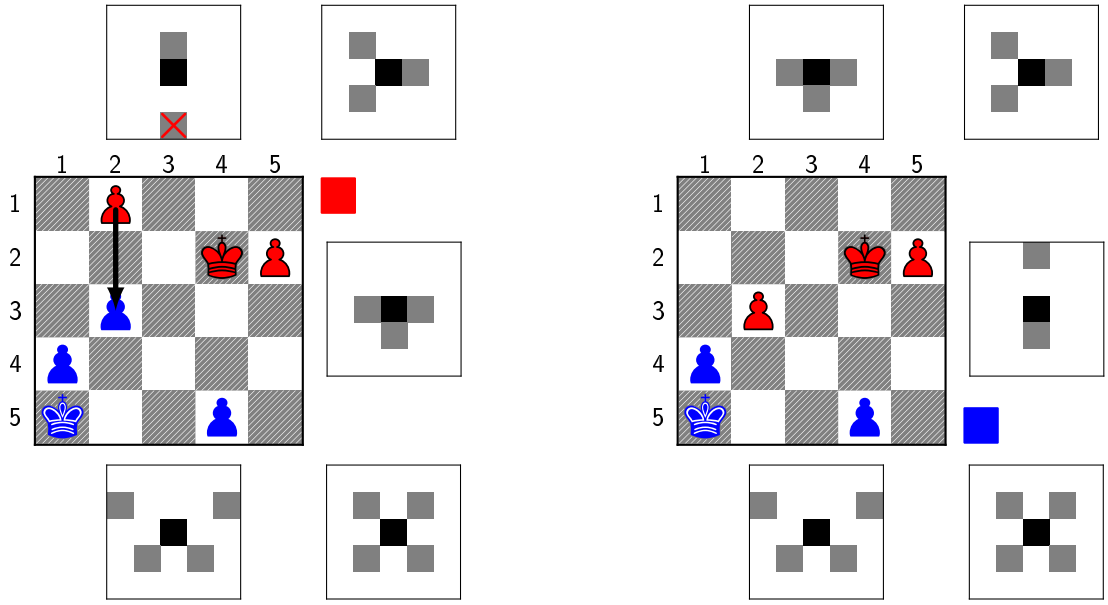

Figure 2: Example of a move. It is Red's turn, and they decide to move their pawn on tile $(2, 1)$ to tile $(2, 3)$ using their first card, capturing a blue student. Note how in the next state the current player is switched, with the used card and the set aside card being swapped accordingly. The used card is rotated halfway because in the next turn the other player will have to take this card into their hand, oriented for them.

Definition 2 (pawn). *A* pawn p *is either a* master *or a* student*, pertaining to a particular player* π*.* $p \in P \coloneqq \{m_r, m_b, s_r, s_b\}$*.* $\overline{p}$ *denotes the same pawn type (master or student) of the other player (red* $\leftrightarrow$ *blue).*

The *game board* consists of a grid of *tiles*, each potentially holding a single pawn.

Definition 3 (tile). *A* tile t *is either empty* $(\varnothing)$ *or holds a pawn;* $t \in T \coloneqq P \cup \{\varnothing\}$*.*

Definition 4 (board). *A* board B *is an* $n \times m$ *grid of tiles:* $B \in T^{n \times m}$*, with* $n \geq 1$*,* $m \geq 2$*.* n *and* m *denote the width and height of the board respectively. For any board* B*,* B_{xy} *denotes the tile in the* x*-th column from the left and the* y*-th row from the top, where the* red *player is always seated at the top.*

Note that a pawn is *solely defined by its type and its colour*—its position on the board is irrelevant. Two boards are deemed equal if all their associated tiles are equal, meaning that boards are invariant over transposition of identically coloured students.

There are two special tiles on a board: the temple arches.

Definition 5 (temple arch). $t_r \coloneqq (\lfloor \frac{n+1}{2} \rfloor, 1)$ *and* $t_b \coloneqq (\lceil \frac{n+1}{2} \rceil, m)$ *are the* temple arches *of respectively the red and blue players. These are the starting positions of the masters. If a master reaches the temple arch of opposing colour, the game ends.*

Pawns are moved around the board according to *move cards* held by each player.

Definition 6 (card). *A* card C *depicts a set of offsets from the origin. A card permits the player to move any of their pawns in accordance with any of these offsets.* $C \subset [-2..2]^2$*. Each card is also associated with a colour:* $\chi \colon \mathcal{P}([-2..2]^2) \to \{r, b\}$*. The colour of the set-aside card in the initial state dictates which player's turn it is first.*

All these game elements combine to form a single *game state*: a single point in time during a play of the game.

Definition 7 (game state). *A* game state $S = (\pi, B, C_s, C_R, C_B)$ *is a 5-tuple consisting of the current*

player π (either red or blue), a board $B \in T^{n \times m}$ and five cards ($\subset [-2..2]^2$), in the order of the set-aside card C_s, the multiset of the two cards $C_R \coloneqq \{C_{r1}, C_{r2}\}$ belonging to the red player, and the multiset of the two cards $C_B \coloneqq \{C_{b1}, C_{b2}\}$ belonging to the blue player. X denotes the set of all game states reachable from the initial state.

Much like the positions of identically coloured students, the order of the two cards held by each player does not matter for game state equality.

For the initial state $S_I = (\pi, B, C_s, C_R, C_B)$ we have the following:

- five move cards are randomly drawn: $C_{r1}, C_{r2}, C_{b1}, C_{b2}, C_s \in \mathcal{P}([-2..2]^2)$, with the two players' hands $C_R = \{C_{r1}, C_{r2}\}$ and $C_B = \{C_{b1}, C_{b2}\}$ (note that these hands are *unordered multisets*);
- the current player is the one indicated by the set aside card: $\pi = \chi(C_s)$;
- on the board, the masters are on their own temple arches: $B_{t_r} = m_r$, $B_{t_b} = m_b$. The students occupy the other tiles in those rows: $B_{x,1} = s_r$ for $x \in [1..n]$ s.t. $(x, 1) \neq t_r$, and $B_{x,m} = s_b$ for $x \in [1..n]$ s.t. $(x, m) \neq t_b$. The rest of the tiles are empty: $B_{xy} = \varnothing$ for $x \in [1..n]$, $y \in [2..m-1]$. The initial 5×5 board state is displayed in Figure 1.

Definition 8 (terminal state)**.** *A game state is called* terminal *if and only if at least one of the following holds:*

- $\exists m \in \{m_r, m_b\} : B_{xy} \neq m \forall (x, y) \in [1..n] \times [1..m]$ *(a master got captured and got taken off of the board);*
- $B_{t_r} = m_b$ *or* $B_{t_b} = m_r$ *(a master reached the temple arch of the opponent).*

$\mathcal{T} \subset X$ denotes the set of all terminal states that are reachable from the initial state.

Definition 9 (move)**.** *A* move $m \coloneqq (o, d, C) \in \mathcal{M} \coloneqq ([1..n] \times [1..m])^2 \times \mathcal{P}([-2..2]^2)$ *holds an* origin o *and* destination d*, as well as a* card C *such that* $d - o \in C$*.*

A move $m = (o, d, C)$ is considered *valid* for a specific game state $S = (\pi, B, C_s, C_R, C_B)$ if and only if $C \in C_\pi$, $B_o \in \{m_\pi, s_\pi\}$ and $B_d \notin \{m_\pi, s_\pi\}$, i.e., if and only if the move is defined by one of the current player's cards, and it moves one of the current player's pawns to either an empty tile, or one occupied by an opponent's pawn.

We define M_S as the set of all valid moves pertaining to a game state S. In the rare case that by this definition $M_S = \varnothing$ with $S \notin \mathcal{T}$, we instead define $M_S = C_\pi$—the two cards the player has choice between discarding. For terminal states $S \in \mathcal{T}$ we do simply set $M_S = \varnothing$.

For each move $m = (o, d, C)$ we also define a mapping $m \colon \{S \in X \mid m \in M_S\} \to X$ from any state for which this move is valid to its next state with the move applied: let $S = (\pi, B, C_s, C_R, C_B)$ such that $m \in M_S$, then $m(S) \coloneqq (\pi', B', C'_s, C'_R, C'_B)$, where

- $\pi' = \overline{\pi}$—the current player's opponent;
- $C'_{\overline{\pi}} = C_{\overline{\pi}}$, $C'_s = C$ and $C'_\pi = (C_\pi \setminus \{C\}) \cup \{C_s\}$—the used card is swapped with the set-aside card;
- $B'_{xy} = B_{xy}$ for all $(x, y) \neq o, d$, remaining unaltered, with only $B'_o = \varnothing$ and $B'_d = B_o$—the pawn moving from its origin to its destination.

In the rare case that there are no valid conventional moves, and therefore move $m = C \in \mathcal{P}([-2..2]^2)$ is a card to discard, then $m(S)$ is the same, except that $B' = B$.

2 Retrograde analysis

For deterministic turn-based games, each decision point for a player can be considered a game state, where each valid move alters the game state into a new one. We can render the connections between

these game states by means of a graph, where the vertices represent the game states reachable from the initial state, and the arrows represent the valid moves between these game states.

Definition 10 (state graph). *Let $G = (V, A)$ be a directed graph, with the nodes $V = X$ consisting of all reachable game states from the initial state, and the edges being between two states reachable via single valid move: $A := \{(S, S') \in V^2 \mid \exists m \in M_S \text{ s.t. } m(S) = S'\}$. We call G the* state graph *for the game Onitama.*

In this graph, each *play* of the game—each sequence of game states and moves from an initial state—can be represented as a path through the state graph. As by definition each terminal state has no valid moves, each vertex corresponding to a terminal state has no outgoing edges, meaning each path that ends up in a terminal state has to end there.

For the remainder of this thesis, when referring to a 'path through the state graph,' we implicitly assume this path to either end in a terminal state, or go on forever.

In this section we define strategies based on these state graphs, associate those with optimal play, and discuss an algorithm fit for finding such optimal strategies for Onitama.

2.1 Positional strategies

As all non-terminal game states mark a decision point for a player, and every game state inherently stores all relevant history within itself, each deterministic strategy should always provide the same move to every individual game state, regardless of the history of the game.

For this reason, deterministic strategies can be reduced from algorithms assigning a game state its move, to simply the complete mapping itself. We call such mappings '*positional strategies.*'

Definition 11 (positional strategy). *A* positional strategy[1] *is a deterministic mapping $x \colon X \to \mathcal{M}$, assigning a specific move to be performed for each reachable, non-terminal game state.*

Positional strategies say nothing about the quality of their outcomes; they merely assign moves to game states. We would like to have a notion of a *'perfect'* strategy—one that cannot be improved for a single player by the sole actions of that player.

Definition 12 (Perfect positional strategy). *A positional strategy x is called* perfect *if and only if for each player π no positional strategy x' exists in which all of $\overline{\pi}$'s moves are given by x, and when starting in some state $S \in X$, π ends up with a better outcome than they get through x.*

Every perfect positional strategy x also associates each node with a quality $Q_x \colon X \to \{Win, Draw, Loss\}$, *dictating the best outcome the current player can enforce by following x. The ordering of these qualities is obviously $Win > Draw > Loss$.*

Perfect positional strategies induce Nash equilibria: under a perfect positional strategy, no single player can improve their outcome by deviating from it, so long as their opponent continues playing according to the perfect positional strategy. In Section 2.2 we show such a perfect positional strategy always exists.

For perfect positional strategies, each player is guaranteed their best possible outcome given their opponent's strategy.

Once a perfect positional strategy starting in the initial state has been found, one could argue that such deterministic games have no merit being played anymore, as by virtue of the perfect positional strategy, the optimal way of play has already been found. As such, all games would be played in the exact same way; the game is, in essence, solved.

Definition 13 (Strong solution). *A game is considered* strongly solved *if there exists an algorithm that can construct a perfect positional strategy covering all game states reachable from all initial states*

[1] Referred to as *pure* positional strategies in [And+10].

with feasible computational resources[2].

Many games are too complicated to be solved strongly; their state space is too large to analyse for a perfect positional strategy in a feasible amount of time. For such complicated games, there is also the notion of a *weak solution*, for which the space that needs to be analysed can be greatly reduced.

Definition 14 (Weak solution)**.** *A game is considered* weakly solved *if there exists an algorithm that can construct* part of *a perfect positional strategy* $x\colon X' \to \mathcal{M}$, $X' \subset X$ *such that from all initial states, both players have their next optimal move defined by* x *regardless of what move their opponent would perform, with feasible computational resources*[2].

Weak solutions, in comparison with strong solutions, allow for a large part of the state graph to be pruned for analysis. Because weak solutions only concern themselves with strategies where at least one player plays optimally, weak solutions do not have to consider game states that are only reachable through *suboptimal* play by *both* players.

If, for example, from a vertex S with current player π, a move to a state that is losing for $\overline{\pi}$ (and thus winning for π) is found, then the optimal move out of S has been found. Therefore, all other moves out of S are suboptimal. Weak solutions do not concern themselves with strategies where both players play suboptimally, so when analysing beyond any suboptimal move out of S, one would only still need to search for optimal moves for $\overline{\pi}$, as π will have at that point already played suboptimally. If in some state S' beyond S, an optimal move for $\overline{\pi}$ has been found, then no other move out of S' has to be analysed anymore, as beyond that point both players will have played suboptimally. Subsequent states can therefore be left completely unanalysed. This pruning can severely decrease the number of nodes and edges that would need to be analysed, cutting down on the amount of resources necessary to find a solution.

In this thesis, we consider all possible initial states for Onitama as issuing individual subgames to the main board game. As such, with 'solving Onitama' we mean finding a (weak or strong) solution from a given initial state, after the cards have already been dealt, and an staring player has thus been assigned.

We first discuss an algorithm capable of *strongly* solving Onitama, given enough time and resources: retrograde analysis.

2.2 Retrograde analysis

In finite games, perfect positional strategies can often be computed using a simple Minimax algorithm. Such algorithms work bottom-up, evaluating the quality of each terminal state, and then iteratively moving up the state tree, mapping each node to both the optimal outcome for that state's player as well as the move resulting in that optimal outcome.

Such a strategy, however, only works for *finite* games with state *trees*. Once infinite play is allowed (and occasionally beneficial), evaluating the quality of terminal states and working bottom-up is no longer guaranteed to work, as not all games are guaranteed to terminate under perfect play by all players.

We call such games *chess-like*. A game is chess-like if it is strictly competitive[3], has perfect information, the outcome of each game is either *win*, *lose* or *draw* with infinite play being associated with draws, and each game state is at a finite distance from the initial state and has a finite number of immediate successors[Ewe02]. Onitama is one such chess-like game.

In [And+10], an algorithm for solving games with potential infinite play is discussed: *retrograde analysis*. Retrograde analysis is able to compute perfect positional strategies for chess-like games for which an explicit state graph is given. Retrograde analysis' steps are formally worked out for Onitama in

[2] For this thesis, 'feasible computational resources' means taking under two hours to compute on an AMD Ryzen 5 2600 CPU with 16 GB of DDR4 SDRAM.

[3] A game is considered *strictly competitive* if an outcome is better for one player if and only if it is worse for another.

Algorithm 1. (Note that this algorithm already contains several generalisations in preparation for Algorithm 2. In particular, for the regular retrograde analysis algorithm the full state graph is given, meaning there are no unexpanded leaf nodes: $V_L = \varnothing$, and all unlabelled states are fully expanded: $V_F = V_U$. Additionally anything to do with draw nodes in `AnalyseEdge` is superfluous for retrograde analysis, as it only labels Draws as a final step.)

Retrograde analysis starts off by labelling each terminal state with its associated outcome (in our case always losing for what would be its current player). It then iteratively picks an unlabelled edge (u, v) from an unlabelled state u to a labelled state v, which it then labels according to u's interests using the information provided by v:

- if v is losing, then this edge provides a winning move for u's player: it will label u winning, (u, v) optimal, and all other outgoing edges from u redundant;
- if v is winning, then this edge guarantees a loss for u's player. This is thus only the optimal move for u if *all* moves out of u lead to a winning node: if (u, v) is the last unlabelled edge out of u, it will label it optimal and label u losing. Otherwise, it will simply label (u, v) redundant.

Through these rules, a node is only labelled after all of its outgoing edges have been considered and labelled as well, with only a single outgoing edge per vertex being labelled 'optimal.'

Once no such unlabelled edges from an unlabelled node to a labelled one exist anymore, all remaining unlabelled edges are between two unlabelled nodes. Furthermore, every node for which all edges have been labelled, has itself been labelled as well, meaning all remaining unlabelled nodes have at least one outgoing edge still unlabelled. These two facts combined tell us that every unlabelled node has an infinite path starting in that node going through exclusively unlabelled nodes and edges.

Note that if for any of these unlabelled nodes there is an edge to a losing node, it would have been marked optimal and the node would have been labelled. Therefore, every outgoing edge from these nodes either leads to an unlabelled node, or to a winning node.

As an infinite game never assigns a winner nor a loser, they effectively form *draws*—which is preferred over the loss that might follow upon deviating from this unlabelled path. In accordance with this preference, every unlabelled node will be labelled a 'draw,' with any arbitrary one of its unlabelled outgoing edges labelled optimal (and the rest redundant).

As such, this algorithm always produces a perfect positional strategy.

A more formal proof of this algorithm can be found in [And+10].

2.3 Optimal strategies

This algorithm assigns a quality and an optimal move to *every* game state reachable from the initial state: it *strongly* solves the game. Table 1 shows which of the two players (if any) wins if both players play optimally in games played on a variety of board sizes with all cards being the *boar* card in Figure 3, allowing each pawn to move a single tile forward, left or right. Because all cards are the *boar*, the set-aside card in particular is a boar as well. The colour of the *boar* card is red, meaning Red is first to move.

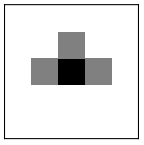

Figure 3: The boar card.

m \ n	1	2	3	4	5
2	Red	Red	Red	Red	Red
3	Blue	Blue	Blue	Blue	–
4	Red	Red	Red	–	–
5	Blue	Draw	–	–	–

Table 1: Winner of the game for various board sizes $n \times m$ if both players play optimally, with exclusively *boar* cards. All tests without results exceeded two hours in runtime and were terminated.

Algorithm 1 Retrograde analysis on state graph $G = (X, A)$

$x(S) \leftarrow$ Unknown for all $S \in X$ ▷ *For each state, set its 'optimal move' to* Unknown.
$q(S) \leftarrow$ Unlabelled for all $S \in X$ ▷ *q stores the quality of all nodes. Initially no nodes are labelled.*

▷ *Analyses edge (S, S_n), with A_U storing all unlabelled edges.* ◁
function AnalyseEdge(S, S_n, A_U, q, x)
    if $q(S_n) =$ Lose **then**
        ▷ *If S_n is losing for S_n's current player, it is winning for S's current player.* ◁
        $q(S) \leftarrow$ Win
        $x(S) \leftarrow m$ s.t. $m(S) = S_n$
        $A_U \leftarrow A_U \setminus \{(s, s_n) \in A_U : s = S\}$ ▷ *Label all other outgoing edges redundant.*
    else if $q(S_n) =$ Win **then**
        ▷ *If S_n is winning for S_n's current player, it is losing for S's current player.* ◁
        if $\{(s, s_n) \in A_U : s = S\} = \{(S, S_n)\}$ **then** ▷ *If this is S's last unlabelled outgoing edge.*
            if $\exists m \in M_S$ s.t. $q(m(S)) =$ Draw **then** ▷ *Prefer a draw over a loss.*
                $q(S) \leftarrow$ Draw
                $x(S) \leftarrow m$
            else
                $q(S) \leftarrow$ Lose
                $x(S) \leftarrow m$ s.t. $m(S) = S_n$
    else if $q(S_n) =$ Draw **then** ▷ *Only occurs in Algorithm 2.*
        if $\{(s, s_n) \in A_U : s = S\} = \{(S, S_n)\}$ **then** ▷ *If this is S's last unlabelled outgoing edge.*
            $q(S) \leftarrow$ Draw
            $x(S) \leftarrow m$ s.t. $m(S) = S_n$
    $A_U \leftarrow A_U \setminus \{(S, S_n)\}$

function RetrogradeAnalysis($A_U, G = (V, A), q, x$) ▷ *A_U stores all unlabelled edges.*
    while $\exists (S, S_n) \in A_U : q(S) =$ Unlabelled and $q(S_n) \neq$ Unlabelled **do**
        ▷ *Use the information of S_n's label to label its incoming edge, and possibly also S.* ◁
        AnalyseEdge(S, S_n, A_U, q, x)
    ▷ *Determine all unlabelled fully expanded vertices, defined as having no outgoing paths through exclusively unlabelled vertices ending in an unlabelled leaf vertex.*
    (Only necessary for Algorithm 2; on a completed state graph $V_F = V_U$.) ◁
    $V_U \leftarrow \{S \in V : q(S) =$ Unlabelled$\}$ ▷ *Unlabelled vertices.*
    $V_L \leftarrow \{S \in V_U : S$ has no outgoing edges in $A\}$ ▷ *Leaf nodes (states that are not expanded).*
    $V_F \leftarrow \{S \in V_U :$ there is no path in $A \cap {V_U}^2$ from S to a node in $V_L\}$ ▷ *Fully expanded states.*
    ▷ *Label all unlabelled, fully expanded vertices draws.* ◁
    for all $S \in V_F$ **do**
        $q(S) \leftarrow$ Draw
        $x(S) \leftarrow m$ s.t. $m(S) \in V_F$ or $q(m(S)) =$ Draw
        $A_U \leftarrow A_U \setminus \{(s, s_n) \in A_U : s = S\}$

▷ *The last player to move is the winner, so the 'current player' during the terminal state has lost.* ◁
$q(S) \leftarrow$ Lose for all $S \in \mathcal{T}$
RetrogradeAnalysis(A, G, q, x)

It is worth noting that for $m = 2$, Red, the starting player, always wins. This is because for $m = 2$, the two rows occupied by the different players are directly in front of one another, meaning the blue master will have a red pawn directly in front of it. Because the red player holds a *boar* card, they can move that pawn forward and immediately capture the master, winning them the game.

For $m = 3$ and $n \leq 3$, Blue always wins. This has to do with the fact that in the initial state there is only a single empty row between the two players. As such, once Red moves any of their pawns forward, that pawn will be right in front of a blue pawn, allowing Blue to capture that pawn immediately in their subsequent turn. This blue pawn then also ensures any horizontal movement by Red will place their pawn in front of a blue pawn as well, allowing for that red pawn to be captured. In short, regardless of which pawn Red chooses to move and where they choose to move it to, Blue will always be able to follow it up by capturing that pawn while giving Red no chance of retaliation. As such, Blue can simply keep capturing red pawns each turn until the red player is forced to move, and in turn sacrifice, their master, guaranteeing Blue their win.

This strategy does not directly extend to $n > 3$, as this provides Red the opportunity to move one of their students that is next to two other red pawns forward, and after that one gets captured, move one of those neighbouring pawns horizontally, in front of the blue pawn. This red pawn is then guarded by the other neighbour. As such, if the blue pawn were to capture this red pawn by moving forward onto row 1, Red is able to subsequently capture this blue pawn—which could, through strategically picking the first red pawn to move, be made to be the blue master, making it so that Red wins.

The 2×5 board results in a draw, which may take the form of both players moving their master forward, and then moving their pawns back and forth horizontally in perpetuity, as is shown in Figure 4. Deviation from this pattern by moving the student forward allows the opponent to move *their* student forward as well, after which the original player would be forced to move one of their pawns forward, onto a tile guarded by the opponent. The opponent could then capture this pawn while simultaneously guarding both tiles the other player would be allowed to move their last remaining pawn onto, ensuring the opponent would win. Deviation by moving the master forward allows for the opponent to move their student forward, making sure that regardless of what the other player's next move would be, they could capture the master next.

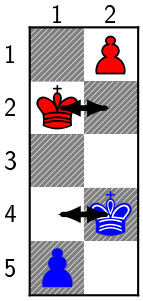

Figure 4: 2×5 draw strategy with only *boar* cards.

Similar results hold for moving the students forward from the initial state, instead of the masters, or if one moves their master forward whereas the other moves their student. Neither player can ensure themself a win, so the best-case scenario for both players is to continue playing forever, in a draw.

2.4 Runtime

For sufficiently complex games, the number of game states—and with it retrograde analysis' search space—blows up dramatically, potentially demanding more computational resources than the maximum specified for a 'strong solution' in Definition 13, making it so these games cannot be considered 'solved' anymore.

Table 2 shows the combined runtime of constructing the state graph and executing retrograde analysis on it, for a variety of different board sizes. This table shows rapid growth in runtime as the size of the board increases, in some instances going from mere seconds to over an hour by adding just a single row or column.

As Table 3 shows, most of this time is spent on building up the state graph, with only a miniscule fraction being taken up by retrograde analysis itself. When optimising this algorithm, it is therefore significantly more fruitful to try to reduce the size of the state space, making it so the construction of the state graph does not take as long, as there are fewer states in all to expand.

m \ n	1	2	3	4	5
2	0.000 s	0.000 s	0.004 s	3.797 s	4862.745 s
3	0.000 s	0.002 s	39.135 s	–	–
4	0.000 s	0.063 s	4503.315 s	–	–
5	0.000 s	0.250 s	–	–	–

Table 2: Average runtime for the state graph construction and execution of retrograde analysis for various board sizes $n \times m$ with exclusively *boar* cards. All tests without results exceeded two hours in runtime and were terminated. Averaged over 10 runs.

m \ n	1	2	3	4	5
2	0.000 s	0.000 s	0.000 s	0.010 s	0.586 s
3	0.000 s	0.000 s	0.048 s	–	–
4	0.000 s	0.000 s	0.597 s	–	–
5	0.000 s	0.003 s	–	–	–

Table 3: Average runtime for retrograde analysis on given state graphs of various board sizes $n \times m$ with exclusively *boar* cards. For all tests without results, the construction of the state graph exceeded two hours in runtime, meaning retrograde analysis could not be tested for those sizes. Averaged over 10 runs.

Regardless, the runtime grows so rapidly that a full 5×5 board seems far out of reach for retrograde analysis. We thus argue Onitama too complicated to strongly solve within the confines of this thesis.

3 The complexity of Onitama

Onitama, while having a very simple ruleset, has a lot of combinatorial elements that result in this game having a remarkably complex state space.

In this section we discuss crude upper and lower bounds for the various game elements of Onitama, and use these to formulate an argument against the feasibility of retrograde analysis strongly solving this game on its conventional 5×5 board.

We first find upper and lower bounds for the number of board configurations reachable from an initial state in terms of the board's dimensions, which we then in subsequent subsections expand to also take the distribution of the cards and the current player into account.

3.1 Upper bound for the number of board configurations

On a valid board there are only one or two masters ever present. For each player there are also anywhere from 0 to $n-1$ students in play. Each of these pawns could be on any one of the $n \cdot m$ tiles of the board, as long as two pawns do not occupy the same tile. Furthermore, transpositions of students of the same colour leaves the board invariant—we are looking for *combinations* of tiles for the students as opposed to *permutations*. This results in the following upper bound for the number of board configurations:

$$\sum_{M=1}^{2} \sum_{S_r=0}^{n-1} \sum_{S_b=0}^{n-1} 2 \binom{n \cdot m}{M, S_r, S_b, (nm - M - S_r - S_b)}. \tag{1}$$

In this expression, the three consecutive sums represent the number of masters, red students, and blue students still in play respectively. For each, we take the multinomial, where we count the number of distinct ways the $n \cdot m$ tiles can be divided over the M masters, S_r red students and S_b blue students, leaving the remaining tiles empty.

We multiply this value by 2 because the multinomial merely concerns itself with the number M of

masters still in play—not the colours of these masters. In the case of $M = 1$, this single master could belong to any of the two players, and in the case of $M = 2$, both of these masters need to be of different colours, leaving only two options to colour them both.

A multinomial coefficient is largest if its 'objects' (the upper value) are dispersed as evenly as possible over the 'bins' (the lower values). For $m \geq 4$, this multinomial is thus largest if $M = 2$ and $S_r = S_b = n - 1$. Assuming this multinomial attains its largest possible value across all $2 \cdot n \cdot n$ terms in the summations, the number of theoretically possible board states is in the order of

$$\mathcal{O}\left(4n^2 \binom{n \cdot m}{2, (n-1), (n-1), (nm - 2[n-1] - 2)}\right) = \mathcal{O}\left(\frac{n^2\,(nm)!}{([n-1]!)^2\,(n[m-2])!}\right). \tag{2}$$

This upper bound of distinct board configurations evaluated on all board sizes from a 1×2 board up to a 5×5 board can be found in Table 4.

m \ n	1	2	3	4	5
2	6	188	3582	55 512	766 250
3	12	1002	53 298	2 352 924	93 623 700
4	20	3320	365 004	34 048 112	2 905 957 700
5	30	8390	1 589 940	258 727 220	38 813 144 750

Table 4: Upper bounds provided by Equation (1) for the number of distinct board configurations of an $n \times m$ board.

It is immediately evident that the size of the state space grows dramatically with the size of the game board. It is also noteworthy that in almost all cases, adding one extra *column* grows the state space more than adding an extra *row* does. This has to do with how the initial state is defined for these differently-sized boards, and what effect that has on possible subsequent states. Going by this definition, adding a row simply adds a row of empty tiles between the two mandatory rows of pawns in the initial state. This increases the board size, increasing the upper variable in the multinomial coefficients in Equation (1), which in turn increases the number of possible pawn configurations. Adding a column, however, adds an additional pawn for each player in the starting configuration. This means that in addition to the larger upper variable in the multinomial coefficient, it also increases the students' values in the multinomial coefficient, brining them comparatively closer to the amount of empty tiles, as opposed to further away. This means the multinomial coefficient itself will also be closer to its maximum attainable value, where adding an extra row would only remove it further from its maximum. Furthermore, the added students add onto the length of the latter two sums in Equation (1), which results in a comparatively larger increase than for rows.

Equation (1) truly provides an upper bound; this number could contain some states that are inaccessible from the initial state through normal play. Take for example a game state where both masters occupy each other's temple. Once the first master reaches the other's temple, the game immediately ends, preventing the second master from reaching the first's temple. Despite this state's unreachability, it is still counted by Equation (1). The upper bounds in Table 4 are thus not sharp.

3.2 Lower bound for the number of board configurations

We also provide a crude *lower* bound for the number of possible board configurations reachable from the initial state of an $n \times m$ game, with $n, m \geq 3$. We do this by providing an algorithm that can reach any given board state that adheres to a number of conditions, with which we can then guarantee that each state that adheres to these conditions is actually reachable from the initial state.

For this lower bound we assume the game starts in its initial state, with the five cards consisting exclusively of copies of the *boar* card, as seen in Figure 3. This allows both players to move any of their pawns a single tile forward, left or right. As such, the swapping of the used card with the one that is set aside becomes irrelevant, because the set-aside card is always the boar card, and each player

always holds two copies of the boar card, meaning they can only ever swap a boar card for another boar card. For the rest of this section, we therefore no longer mention the swapping of the cards, and have that happen implicitly. The boar card dictates that it is initially Red's turn.

Our methodology is as follows: we first capture a student from one of the players, after which this player is able to stall for the rest of the game by keeping its remaining pawns on their initial row, constantly moving one of their pawns left and right onto the single empty tile on this row, allowing the other player to fill the remaining $m-1$ rows arbitrarily.

We thus start by capturing a student.

Lemma 1. *Let $n, m \geq 3$. If m is even, let $B \in T^{n \times m}$ such that $B_{1,1} = B_{1,m} = \varnothing$, $B_{1,2} = s_r$, and B_{xy} is the same as the initial board state for all other tiles (x, y). If m is odd, let $B \in T^{n \times m}$ such that $B_{n,1} = B_{n,m} = \varnothing$, $B_{n,m-1} = s_b$, and B_{xy} is the same as the initial board state for all other tiles (x, y). Then B is reachable from the initial state.*

Proof. This process is visualised for a 5×5 board in Figure 5.

We start with the initial board configuration. We work towards B by applying consecutive moves as Red and Blue.

If m is even, we aim to capture the blue student starting on tile $(1, m)$. If m is odd, we instead aim to capture the red student starting on tile $(n, 1)$. Let x_c be the column of the student to be captured, and let y_c be 2 if m is even and $m-1$ if m is odd. (x_c, y_c) is the tile on which the student will be captured.

For their first moves, Red moves their student on tile $(x_c, 1)$ forward to tile $(x_c, 2)$ and Blue moves their student on tile (x_c, m) forward to tile $(x_c, m-1)$. At this point, tiles $(x_c, 1)$ and (x_c, m) are empty, as is the case in B.

After this, the next few moves ($m-3$ if m is even, $m-4$ if m is odd odd) for the player who is to capture their opponent's pawn consist of moving their student back and forth horizontally, hopping onto and off of tile (x_c, y_c) repeatedly. In the meantime, the other player moves their student one tile forward $m-3$ times. After these moves, the to-be-captured student is on (x_c, y_c), with the capturing student one tile horizontally removed: $(x_c + 1, y_c)$ if m is even and $(x_c - 1, y_c)$ if m is odd.

It is at this point the capturing player's turn, meaning they can move their student horizontally over to tile (x_c, y_c) and capture the other player's student.

The capturing student will then be on tile (x_c, y_c), one student of the opponent is captured, and all other pawns have not moved from their initial positions, meaning we have reached B from the initial state. □

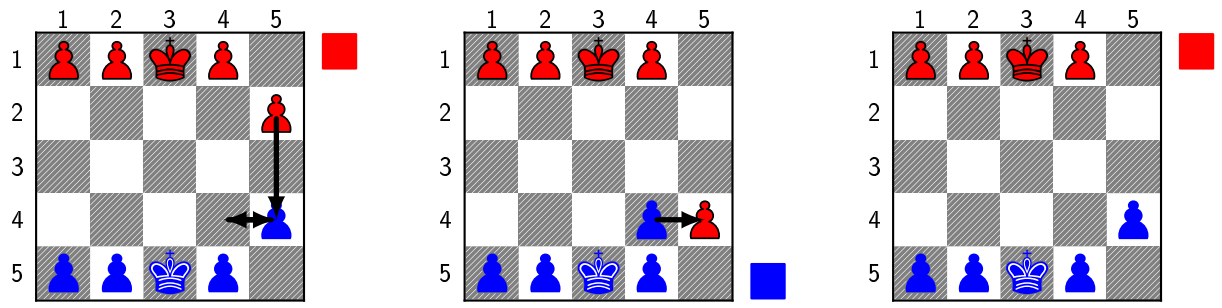

Figure 5: An overview of the student capturing process.

Starting in this board state, we now fill the middle $m-2$ rows with the pawns of the capturing player in any configuration, leaving the other player's pawns as they are. We will count the number of distinct

ways these pawns of the capturing player can be distributed across the $m-1$ rows not occupied by the opponent's pawns as a lower bound for the number of board states reachable from the initial state.

For readability, for the remainder of this section we will be assuming the case where m is even: Red is the capturing player and Blue is the player whose student got captured. All results still analogously hold for the case that m that is odd, and Blue is the capturing player instead.

We will also only be focussing on the red player's moves, and assume Blue constantly moves one of their pawns horizontally onto and off of the same empty tile on row m repeatedly, keeping off of the top $m-1$ rows.

Theorem 1. *Let $n, m \geq 3$ and let $B \in T^{n \times m}$ be a board with $n-2$ blue students and one blue master, all in their initial position, with tile $(1, m)$ empty. Furthermore, let all $n-1$ red students be contained in the upper $m-1$ rows of B, with $(1,1)$ empty, and the red master being either on its temple or in the centre $m-2$ rows. The red temple is either empty or contains the red master. Then B is reachable from the initial state.*

Proof. We start with board $B' \in T^{n \times m}$ such that $B'_{1,1} = B'_{1,m} = \varnothing$, $B'_{1,2} = s_r$, and B'_{xy} is the same as in the initial board state for all other tiles (x, y). B' satisfies the conditions for Lemma 1, meaning B' is reachable from the initial state.

Let $R \subset [1..n] \times [1..m-1]$, $|R| = n$ be the set of the coordinates of the red pawns on board B, and $R_y \coloneqq \{(i, j) \in R \mid j = y\}$ the coordinates in R in row y. Also, let $M \subset [2..n]$ be the columns corresponding to the pawns that still have to be moved: $x \in M$ if an only if $B_{x,1} \neq B'_{x,1}$. We will be referring to the pawn that is currently on tile $(1, 2)$ as the 'first pawn.' Note that as $n \geq 3$, the first pawn is always a student.

We will be moving all red pawns into place, going over the target positions row by row from bottom (row $m-1$) to top (row 3), and within each row from right (column n) to left (column 1). Row 2 will be dealt with separately. Once a pawn has been permanently moved to its destination, that pawn is considered *placed*. An overview of the process of the placement of these rows is displayed in Figure 6.

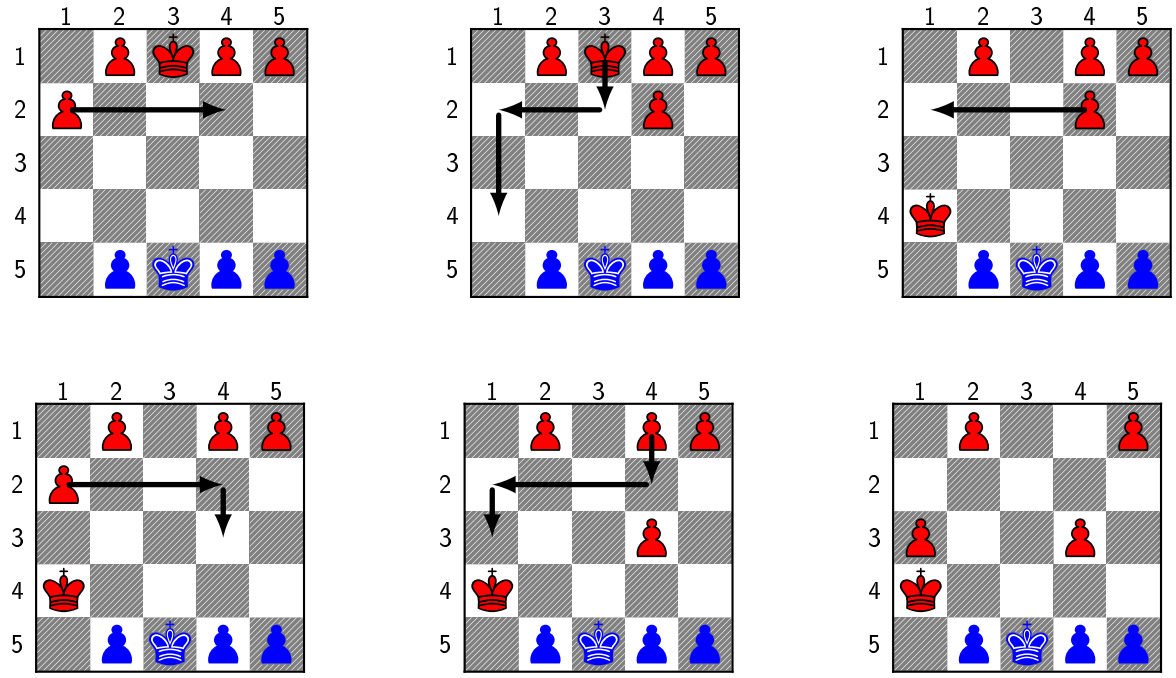

Figure 6: An overview of the placement of the bottom $m-2$ rows.

Let (x, y) iterate over R in the order just described. We need to place one of our pawns onto tile (x, y).

If $B_{xy} = m_r$, this needs to be our master. In the case that $B_{xy} = s_r$, if the first pawn has not yet been placed, we will move *it* instead. Otherwise we place the pawn in row 1 and the column c such that $c \in M$ as small as possible (if it has not yet been placed, but should be), after which $M := M \setminus \{c\}$. In all cases, let (p_x, p_y) be the current location of our chosen pawn.

If our pawn is the master, the first pawn has not yet been placed, and $x = 1$, then move the first pawn horizontally to be one column to the right of the temple. This is possible because $n \geq 3$, and we are only still filling the bottom $m - 2$ rows, meaning row 2 currently only contains the first student, and is otherwise empty.

If $p_y = 1$, which is the case for all pawns except for the first student, move our pawn forward to row 2. This is again possible because we're only still filling the bottom $m-2$ rows, and the only tile on row 2 that could possibly be occupied in this stage is $(1, 2)$, whereas if $p_y = 1$, then $p_x \geq 2$.

From row 2, our pawn can without issue move horizontally to column x, because if necessary for the master, the first pawn has been moved out of the way, and if our pawn is a student then the first pawn should already have been placed beforehand (or is currently being placed). Row 2 is otherwise empty.

Finally, our pawn can move vertically to row y, and be placed. This is possible because all rows between p_y and y are completely empty, due to our iterating row by row from bottom to top.

If the first pawn was moved out of the way to the column right of the temple, move it back to $(1, 2)$.

Once this has been done for rows $m - 1$ through 3, rows 3 through m are identical to B. All that is left is to fix rows 1 and 2. We separately handle three cases: the master has already been placed, the master has not yet been placed, but still has to be, and the master should not be placed. Let s_2 be the number of students in row 2 in board B.

We first handle the case where the master has already been placed. An overview of the process is visualised in Figure 7.

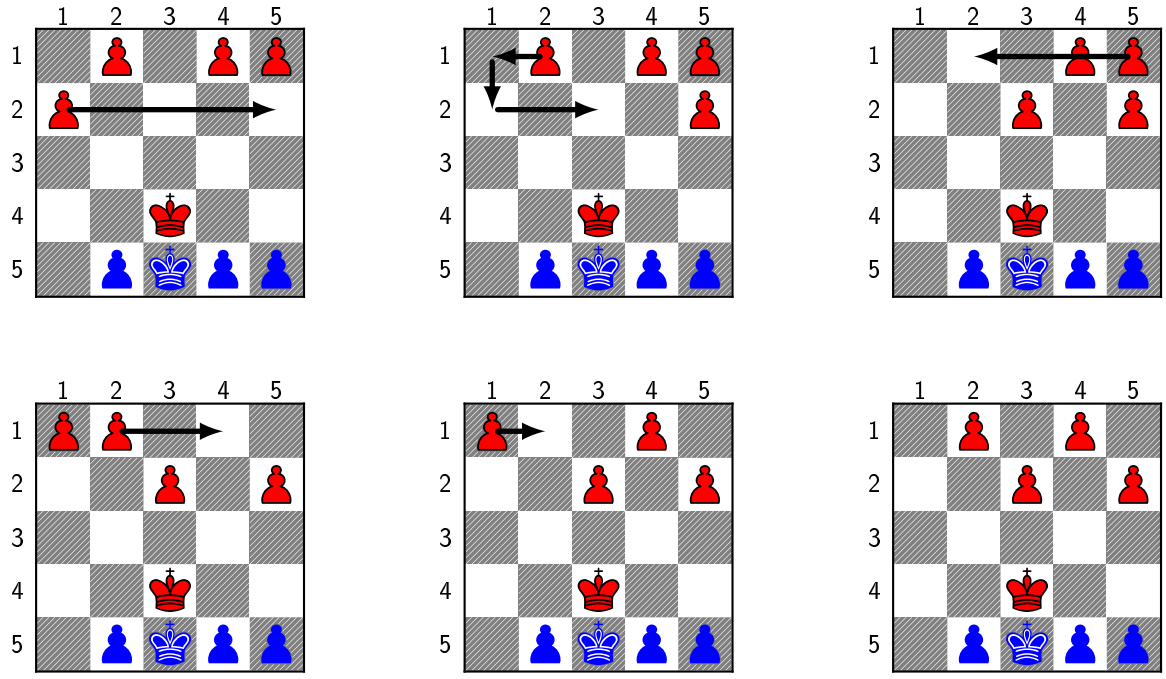

Figure 7: An overview of how rows 1 and 2 are fixed when the master has already been placed.

If the master has already been placed in rows 3 through $m - 1$, then rows 1 and 2 contain exclusively red students. In this case we fill the tiles of R_2 from right to left by moving our pawns in the order

starting with the first pawn if it has not already been placed, followed by the pawns in row 1 from left to right.

For the first s_2 students in this ordering, do the following: if we are placing the first pawn, move it horizontally to the destination column (as at this point this pawn is the only one in its row, this path does not cross any other pawns). If we are instead placing a student starting in row 1, first move it horizontally to column 1 (as it is at this point always the leftmost pawn in row 1, its path is unimpeded), after which it is moved one tile forward to row 2. From here, move it horizontally, placing it onto the destination column. As we are filling them up right-to-left, this is an unimpeded path.

For row 1, all that needs to be done is move the remaining pawns as far to the left as possible from left to right, and then place them from right to left to their corresponding destination tile in R_1, mapped right-to-left. After this, board B has been reached from B'.

Next we handle the case where the master has not yet been placed, but still has to be. An overview of this process is visualised in Figure 8.

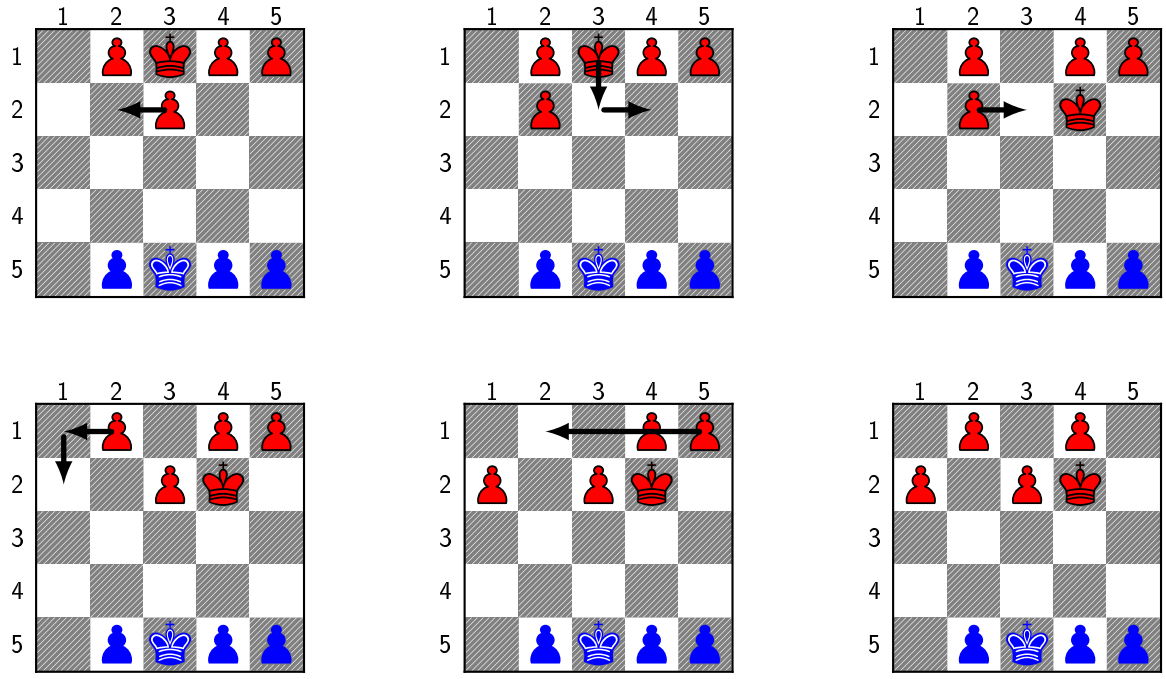

Figure 8: An overview of how rows 1 and 2 are fixed when the master still has to be placed.

If the first pawn has not yet been placed, move it to the rightmost student destination in R_2. After this we place the master. As rows $m-1$ through 3 have already been dealt with, and the master is not allowed to be placed on any other tile in row 1, its destination has to be in row 2. If the first pawn is still in row 2, and the first pawn's column is between the master's current column and the master's destination column, move the first pawn horizontally beyond the temple column so the master's path becomes unimpeded. This is possible as $n \geq 3$. Next, in either case, the master can now be moved one tile forward, and be horizontally placed onto its destination. Finally, if the first pawn has just been moved out of the way, move it back to its destination.

Next, as row 1 now only consists of students, they can easily be moved horizontally such that every empty tile in row 2 that has a student in B has a student above it in row 1. All these students can then move one tile forward, placing them onto their destination.

The students in row 1 then still need to be placed onto their correct spots, which can easily be done as before, because there is no master on this row. After this, board B has been reached from B'.

Finally there is the case that the master has not yet been placed, and should not be either. An overview of this process is visualised in Figure 9.

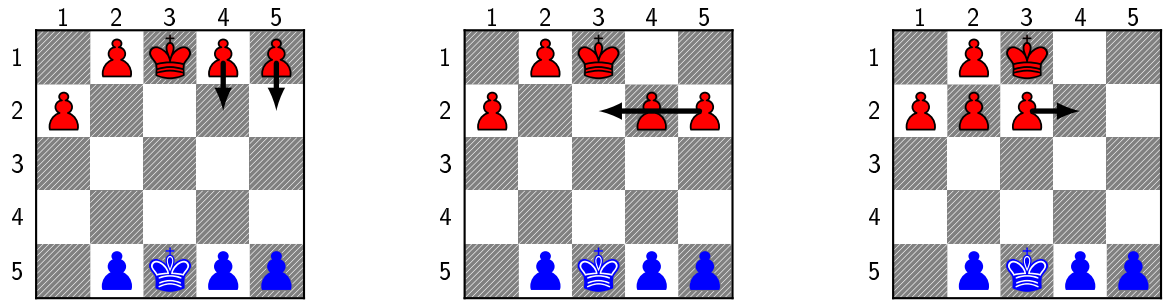

Figure 9: An overview of how rows 1 and 2 are fixed when the master should not be placed.

Move all students in row 1 in columns in M one tile forward. This is possible as the only tile in row 2 that could be occupied is $(1, 2)$ (by the first pawn), with M ranging from columns 2 through n.

Now all pawns in row 2 are red students, meaning we can move all pawns on this row to the left as far as possible (from left to right), followed by placing them onto their corresponding tile in R_2 (from right to left).

As all pawns in row 1 with columns in M have not been moved, row 1 is also equal to row 1 in B, meaning we have reached board B from B'.

In all cases, every row is now equal to its corresponding row in B, meaning board B has been reached from B'. As B' is itself reachable from the initial state, by transitivity B is also reachable from the initial state. □

Theorem 1 gives us a set of boards that are guaranteed to be reachable from the initial state.

Corollary 1. *The number of reachable board states Theorem 1 gives us, is expressed by*

$$\sum_{M=0}^{1}\sum_{S=1}^{n-1}\binom{n-2}{S-1}\binom{n(m-2)}{M, S, n(m-2)-M-S}. \tag{3}$$

Proof. In this equation, the sums over M and S respectively represent whether the red master is moved out of row 1, and the number of red students S that are moved out of row 1. S is at least 1, because in the theorem's prepositions, tile $(1, 1)$ has to be empty, meaning at least one red student needs to be moved out of row 1. If $M = 0$, the master stays on its temple—as that is the only tile on row 1 the red master is allowed to occupy—and if $M = 1$ it is placed somewhere in the centre $m - 2$ rows.

The rightmost multinomial coefficient represents the number of distinct configurations the S students and M masters could form in the middle $m - 2$ rows, leaving the remaining tiles in these rows empty.

This value is multiplied by $\binom{n-2}{S-1}$, representing the number of ways $S - 1$ students could be chosen out of the $n - 2$ on the first row that need to move out of row 1, in addition to the red pawn that captured the blue student. □

This lower bound of distinct board configurations evaluated for all subgames from a 3×3 board up to a 6×6 board can be found in Table 5.

We wish to find a simple expression showing a lower bound rate of growth for the number of reachable game states with respect to the dimensions of the game board, which will serve as its lower bound

order. To this end we instead analyse the following even lower bound:

$$(3) \geq \overbrace{\frac{n(m-2)}{n-1}}^{\text{First pawn}} \overbrace{(n[m-2]-1}^{\text{Master off temple}} + \overbrace{1)}^{\text{Master on temple}} \overbrace{\binom{n(m-1)-4}{n-2}}^{\text{Remaining } n-2 \text{ students}}. \tag{4}$$

Here, the term $\frac{n(m-2)}{n-1}$ refers to the number of locations the *first pawn* can be placed. Because it has to capture the blue pawn, it has to step forward one row, and therefore has to end up in one of the $m-2$ centre rows, on any one of its n columns. This value is divided by $n-1$ in accordance with the symmetry where in every configuration the first pawn could be swapped with any other red pawn without changing the board state—to count each distinct game state at most once.

m \ n	3	4	5	6
3	15	111	369	870
4	60	792	3796	11 696
5	245	5665	38 760	155 155
6	1008	40 404	392 616	2 035 800

Table 5: Lower bounds provided by Equation (3) for the number of distinct board configurations of an $n \times m$ board.

The factor $(n[m-2]-1+1)$ represents the $n[m-2]-1$ tiles the master could be placed on in the centre $m-2$ rows (disallowing the first pawn's chosen location), in addition to its temple arch where it is allowed to remain.

Finally, the binomial coefficient represents the number of distinct ways the $n-2$ remaining pawns could be placed on the $n(m-1)-4$ valid tiles. The four prohibited tiles subtracted from the $n(m-1)$ tiles in the upper $m-1$ rows are the tiles occupied by the first pawn and the master, as well as the temple arch and tile $(1,1)$, as these are required not to contain a student by Theorem 1.

Equation (4) thus sums up at most as many distinct board states as Equation (3). This value can be strictly smaller, as in the case that the master stays on the temple, the binomial coefficient needlessly discounts a tile from the positions the remaining $n-2$ students could be placed on in counting the master's current and initial positions as being different.

The divisor $n-1$ is also in some cases larger than strictly necessary. Not all of these pawn-swapped symmetries are allowed by Theorem 1. This pawn-swapping is only valid if both pawns could reach the other's location from their starting position, while all other pawns end in the same location as before. The first pawn could, for example, never be swapped out for a pawn placed in row 1. This is because we have secured the *boar* card, which does not allow for backward movement, making it impossible for the first pawn—starting on row 2—to move back to row 1.

This means Equation (4) provides an even lower bound to our lower bound of the number of distinct board states reachable from the initial state. We reduce this even further to a simple expression.

$$\begin{aligned}(4) &= \frac{(n[m-2])^2}{n-1}\binom{n(m-1)-4}{n-2} \geq \frac{(n[m-2])^2}{n}\binom{n(m-1)-4}{n-2} \\ &= n(m-2)^2\binom{n(m-1)-4}{n-2} = n(m-2)^2\frac{[n(m-1)-4]!}{(n-2)!\,[n(m-2)-2]!}\end{aligned}$$

Here, note that the largest $n-3$ factors of $[n(m-1)-4]!$ are multiplying factors greater than or equal to $n(m-2)$. All factors smaller than $[n(m-2)-1]$ are divided out by $[n(m-2)-2]!$. Therefore, the whole $\frac{[n(m-1)-4]!}{[n(m-2)-2]!} \geq [n(m-2)]^{n-3}[n(m-2)-1] \geq [n(m-2)]^{n-3}$, i.e., the $(n-3)$-rd largest of these factors multiplied by itself $n-3$ times. We can thus reduce this further to:

$$\begin{aligned}n(m-2)^2\frac{[n(m-1)-4]!}{(n-2)!\,[n(m-2)-2]!} &\geq \frac{n(m-2)^2}{(n-2)!}[n(m-2)]^{n-3} = \frac{n^{n-2}(m-2)^{n-1}}{(n-2)!} \\ &\geq \frac{n^{n-2}}{(n-2)^{n-2}}(m-2)^{n-1} \geq \frac{n^{n-2}}{n^{n-2}}(m-2)^{n-1} \\ &= (m-2)^{n-1}.\end{aligned}$$

Equation (3) (the number of distinct board states reachable by Theorem 1) is thus bounded from below by $(m-2)^{n-1}$:

$$\sum_{M=0}^{1}\sum_{S=1}^{n-1}\binom{n-2}{S-1}\binom{n(m-2)}{M,S,n(m-2)-M-S}=\Omega\left([m-2]^{n-1}\right). \tag{5}$$

While the complexity is not as great as Equation (2), Equation (5) is still exponential in the width of the board, with the base of this exponent being linear in the height of the board. The size of the board thus incurs a great cost onto the number of distinct reachable game states retrograde analysis would have to analyse.

Equation (3), and by extension Equation (5), are only directly applicable to games with five identical *boar* cards. Despite this, it is not a huge leap to apply the order of this equation to games with other sets of cards, so long as these supply both forward and bidirectional horizontal movement (which the standard deck of Onitama guarantees). Equation (3) provides a very conservative lower bound—only effectively taking one of the two players into account—and is only meant to emphasise the rapid growth of the number of reachable game states with the size of the board. As such, we believe any potential difference in the number of reachable states under an arbitrary set of cards in comparison to having all *boar* cards is compensated by the degree to which this lower bound is conservative. We thus believe the order of this lower bound is still applicable to any set of cards.

For these reasons, we will still use the order of this applied lower bound in arguments for the general case of the game, despite the lower bound itself not being directly applicable. Regardless, all data supplied in this thesis by simulations of the game are provided by games with exclusively *boar* cards.

3.3 Card configurations

Along with the state of the board, a game state is also determined by the distribution of the cards among the players, as well as who the current player is. At the start of the game, five move cards are randomly drawn. Throughout the duration of the game, these cards do not change—merely their distribution over the players' hands changes.

Each player holds two cards in their hand, with the remaining one being set aside. The order of these two cards does not matter, so this adds a constant multiplicative factor of $\binom{5}{2}\binom{3}{2}=30$ to our upper bound of game states.

Ordinarily these cards are drawn randomly from a fixed deck of 16 cards, with no repeats, allowing for $\binom{16}{5}$ initial drawings. We relax this restriction, assuming the power set of offsets in $[-2..2]^2$ as our 'deck' of cards: $\mathcal{P}([-2..2]^2)$. We also allow duplicates among our five drawn cards, possibly reducing our multiplicative term to 1, in the case of five identical cards.

There are $\left|[-2..2]^2\right|=5^2=25$ offsets allowed on a card. A card can hold any subset of these, giving us a total of $\left|\mathcal{P}([-2..2]^2)\right|=2^{25}$ cards in our 'deck.' As we relax the restriction on five unique cards in a drawing, we allow for $\binom{2^{25}+5-1}{5}=\frac{(2^{25}+5-1)!}{(2^{25}-1)!\,5!}\approx 3.545\times 10^{35}$ initial drawings: the number of distinct combinations of five cards that can be drawn from a deck of 2^{25} cards, with replacement.

3.4 The actual number of reachable game states

Tests were performed evaluating the actual number of game states reachable from the initial state for various board sizes. The number of vertices in the state graph—representing the number of reachable game states—can be found in Table 6, and the number of edges in the state graph—representing the number of moves possible in the game—can be found in Table 7.

These values show clear exponential growth, just as determined in Section 3.2. The number of edges in the graph also does not seem to grow significantly quicker than the number of vertices in the graph. This is to be expected, because for our chosen distribution of cards, each non-terminal game state has

m \ n	1	2	3	4	5
2	2	6	658	20 819	493 092
3	3	512	59 666	–	–
4	4	3031	431 793	–	–
5	5	8120	–	–	–

Table 6: Number of game states reachable from the initial state with only *boar* cards on a variety of board sizes $n \times m$. All tests without results exceeded two hours in runtime and were terminated.

m \ n	1	2	3	4	5
2	1	5	1007	50 716	1 701 183
3	2	752	168 860	–	–
4	3	6055	1 569 941	–	–
5	4	18 744	–	–	–

Table 7: Number of edges in the state graph of games with only *boar* cards on a variety of board sizes $n \times m$. All tests without results exceeded two hours in runtime and were terminated.

anywhere from 1 to at most $3n$ outgoing edges, which is still a very manageable number given the small values of n in our experiments.

Comparing Table 6 to its upper and lower bounds in Tables 4 and 5, we see that the actual number of reachable game states is closer to the upper bound than it is to the lower bound. This would suggest the actual complexity of the number of reachable board states with exclusively *boar* cards is closer to Equation (2) than it is to Equation (5).

While comparing these numbers, keep in mind that Table 6 counts game states—which includes the current player—while Tables 4 and 5 count board states—which excludes the current player. As such, for a simple conversion, the upper bound could be multiplied by a factor of two to assume each board state is accessible for both players, and the lower bound could simply be taken as an even more conservative one, that only takes a single player's turns into account.

3.5 The complexity of retrograde analysis

The complexity of retrograde analysis is linear in the number of edges in the state graph[And+10]. Note that each vertex in the state graph (except for the initial state) needs to have at least one incoming edge or it would not be reachable from the initial state. As such, the number of edges in the state graph grows at least as quickly as the number of vertices in the state graph. Therefore, the complexity of strongly solving Onitama with retrograde analysis is roughly equivalent to the complexity of the number of game states reachable from the initial state. By Equation (5), this means the complexity of retrograde analysis is at least exponential in the width of the game board, with the base of the exponent being linear in the height of the game board.

Due to this high complexity, and judging from the runtimes of the construction of the state graphs and their labelling by retrograde analysis in Table 2, we do not expect to be able to strongly solve Onitama on a larger board than described within that table. We would rather cut down on the number of moves and game states that need to be analysed for a solution. We would prefer an adjusted algorithm that can *weakly* solve the game instead; only taking the initial state as input, as opposed to the full state graph.

4 Forward-looking retrograde analysis

Retrograde analysis assumes an explicit state graph is given. Furthermore, it computes the quality of each state and move in the entire game, even if these states could never be reached through perfect

play, or are not required for finding a perfect positional strategy from the initial position: it provides a *strong solution* for the game.

Due to Onitama's complexity, strongly solving this game would require a considerable amount of computational power. We would thus prefer an algorithm that is able to *weakly* solve Onitama; one that only provides a perfect positional strategy from the initial state, and does not have to analyse the entire state graph.

We developed Algorithm 2 to fulfil this purpose. Algorithm 2 can weakly solve the game for all players that cannot lose under optimal play. This algorithm is a derivative of retrograde analysis that looks forward from the initial state, as opposed to backward from the terminal states. This algorithm also prunes a large part of the search space during its analysis.

4.1 Forward-looking retrograde analysis

Forward-looking retrograde analysis maintains its own state graph $G = (V, A)$. G starts out only containing the initial state S_0, and is built up throughout the algorithm's runtime. The algorithm also keeps track of the optimal move $x\colon X \to \mathcal{M}$, as well as the quality $q\colon X \to \{\text{Win, Draw, Lose}\}$ of each game state, which are both initially unknown for all states.

In a depth-first manner it recursively calls `Expand` on every game state S it comes across, starting with S_0. These calls analyse the currently expanding node and its outgoing edges in an attempt to label them.

`Expand` first adds S to the set V_E of expanding game states, to make sure the function is not needlessly called on this state again.

After this, it updates the state graph, adding all nodes and edges out of S to the graph according to the rules of the game. If in doing so it finds a terminal state, this terminal state will immediately be labelled Losing and added to V_E, due to terminal states not having outgoing edges.

After all new vertices and edges have been added to the graph, if a terminal state has been found, there is a new endpoint for retrograde analysis to analyse the whole graph. As such, in this case, retrograde analysis is called on all unlabelled edges between two expanding vertices—ignoring the edges adjacent to vertices on which `Expand` has not yet been called—after which the set A_U of all unlabelled edges is updated by removing all newly labelled edges. Because there is a direct move from the current game state to a terminal (losing) state, S is necessarily marked winning by retrograde analysis, meaning this `Expand` call can return.

If no terminal state was found, it then loops over all outgoing edges (S, S_n) of S. If $S_n \notin V_E$, then S_n still needs to be expanded, so `Expand` is recursively called on S_n. If this call happens to label the current state S we can return from this function call immediately thereafter. If it instead managed to label the initial state S_0, then we are able to exit the algorithm entirely by continuously returning, as a weak solution has been found for all players that cannot lose from the initial state under optimal play.

Regardless of whether S_n was already expanded or not, the algorithm then checks if S_n is labelled, with the edge (S, S_n) still unlabelled. If both are the case, it attempts to label this edge manually, using the `AnalyseEdge` function from Algorithm 1, which tries to label S with the information its currently analysed successor S_n provides:

- If S_n is losing, then S can immediately be labelled winning, with (S, S_n) being its optimal outgoing edge.
- If S_n is winning, then S is only labelled if this is its last unlabelled outgoing edge, in the hope that there is still a non-losing move. If there are no other unlabelled outgoing edges, then if there is an outgoing edge to a draw node, that one is optimal and S is a draw as well, otherwise (S, S_n) is optimal and S is losing.

Algorithm 2 Forward-looking retrograde analysis on the initial game state S_0

$G = (V, A) \leftarrow (\{S_0\}, \varnothing)$
$x(S) \leftarrow$ Unknown for all $S \in X$ ▷ *For each state, set its 'optimal move' to* Unknown.
$q(S) \leftarrow$ Unlabelled for all $S \in X$ ▷ *q stores the quality of all nodes. Initially no nodes are labelled.*
$V_E \leftarrow \varnothing$ ▷ *V_E keeps track of all expanding vertices.*
$A_U \leftarrow \varnothing$ ▷ *A_U keeps track of all unlabelled edges.*

function EXPAND($S, G = (V, A), q, x, V_E, A_U$)
    $V_E \leftarrow V_E \cup \{S\}$ ▷ *Ensure `Expand` is not called on S again.*

    for all $m \in M_S$ **do** ▷ *Place all newly found nodes and edges in the graph.*
        $S_n \leftarrow m(S)$
        $V \leftarrow V \cup \{S_n\}$
        $A \leftarrow A \cup \{(S, S_n)\}$
        $A_U \leftarrow A_U \cup \{(S, S_n)\}$

        if $S_n \in \mathcal{T}$ **then**
            $q(S_n) \leftarrow$ Lose ▷ *A terminal node is losing for the 'current player.'*
            $V_E \leftarrow V_E \cup \{S_n\}$ ▷ *Terminal states have no outgoing edges.*

    if $\exists m \in M_S : m(S) \in \mathcal{T}$ **then**
        ▷ *Use the terminal state as an endpoint for retrograde analysis.* ◁
        RETROGRADEANALYSIS(A_U, G, q, x)
        return ▷ *Because there is a direct move to a losing node, S must have been labelled winning.*

    for all $m \in M_S$ **do** ▷ *Analyse all available moves for the optimal one.*
        $S_n \leftarrow m(S)$
        if $S_n \notin V_E$ **then**
            EXPAND(S_n, G, q, x, V_E, A_U) ▷ *Recurse down to assert the quality of the next state.*

            if $q(S)$ or $q(S_0) \neq$ Unlabelled **then**
                ▷ *If S_0 is labelled, then a weak solution has been found for non-losing players: exit.*
                If S is labelled, then its optimal edge has been found and cannot change. ◁
                return

        if $q(S_n) \neq$ Unlabelled and $(S, S_n) \in A_U$ **then**
            ANALYSEEDGE(S, S_n, A_U, q, x) ▷ *Try to label this edge manually.*
            if $q(S) \neq$ Unlabelled **then return**

EXPAND(S_0, G, q, x, V_E, A_U)
if $q(S_0) =$ Unlabelled **then** RETROGRADEANALYSIS(A_U, G, q, x)

- If S_n is a draw, then S is only labelled a draw with (S, S_n) optimal if this is its last unlabelled outgoing edge, in the hope that there is still a winning move.

Labelling nodes and edges in this part of the algorithm is not strictly necessary, as retrograde analysis would already label these nodes and edges on its next call. Labelling these nodes and edges along the way can, however, speed up the algorithm by eliminating suboptimal search paths early, without having to wait until the next terminal state. As such, if this manages to label S, we can return from this function call.

Once S has been labelled, or all outgoing edges have been analysed, the call can return, continuing the expansion of its predecessor.

Once the initial expand call on S_0 has finished, S_0 has either been labelled already, or the currently mapped out state graph is entirely bounded by labelled vertices, meaning one more call to retrograde analysis labels S_0 as well.

This completes the algorithm, guaranteeing a perfect positional strategy from the initial state S_0. We now prove the correctness of this algorithm in small steps, starting by proving only unlabelled states ever get their label changed.

Lemma 2. *Throughout Algorithm 2, if a game state S gets assigned a label or optimal move, it will hold that same label or move for the entire remaining duration of the algorithm.*

Proof. First note that the functions in Algorithm 2 do not directly set any game state labels, nor optimal moves (with the exception of terminal states, which are always losing and have no optimal moves). All labelling is done in `AnalyseEdge` and `RetrogradeAnalysis`, where the labelling of a game state and setting the state's optimal move always happens in conjunction with one another.

Within `Expand`, after each call that can change S's label (`RetrogradeAnalysis`, `Expand`, `AnalyseEdge`), the algorithm immediately returns if S has been labelled. Therefore, these functions are only ever called if S is still unlabelled.

Within `RetrogradeAnalysis` as well, `AnalyseEdge` is only called on an arbitrary edge (s, s_n) if s is unlabelled. In `AnalyseEdge`, if the source state is labelled winning, all of its other outgoing edges are labelled redundant. If the source state was instead labelled losing or a draw, all other outgoing edges were already labelled. Hence, all unlabelled edges at the point of assigning the draws have their starting point yet unlabelled.

The draws are then assigned, which is only done for a subset of the unlabelled game states. Hence, this is once again only done once for exclusively unlabelled states.

Therefore only unlabelled states ever get their label changed, always and exclusively in conjunction with their optimal move; once a node has been given a label and move, it will keep those for the entire remaining duration of the algorithm. □

Using this fact we prove the required properties of the relation between the qualities of the game states and the qualities of their successors.

Lemma 3. *Let x be a positional strategy provided by Algorithm 2. Every path that starts in a winning or losing node S, and follows a positional strategy x' for which every move from a winning node is given by x, alternates between winning and losing nodes.*

Proof. We first prove that every winning node leads to a losing node. Let S be winning, and let x' be a positional strategy for which every move from a winning node is given by x.

Wins are only assigned in `AnalyseEdge`, with a state being labelled a Win if and only if a successor S_n is found that is labelled a Loss, after which the optimal move is set to (S, S_n), with the other outgoing edges being labelled redundant.

Because after this, by Lemma 2, the quality of both S and S_n, as well as the optimal move $x(S)$, are set in stone, all winning nodes lead to losing nodes through x. As every move from a winning node in x' is given by x, the same holds for x'.

Next, we prove that every valid move from a losing node leads to a winning node. Let S be losing. If $S \in \mathcal{T}$, then by definition S has no valid moves. Let $S \notin \mathcal{T}$.

`AnalyseEdge` is the only function that can label a non-terminal node losing. Such a non-terminal node S is only labelled losing, if the function is analysing an edge whose target S_n is labelled winning, all other outgoing edges have already been labelled, and none of them lead to a draw node.

`AnalyseEdge` is only called on unlabelled edges that still have an unlabelled source node. If it comes across an edge leading to a losing node, then the source is immediately labelled a Win, meaning `AnalyseEdge` will never be called on any one of its outgoing edges again. As such, if in `AnalyseEdge` the target is winning, and it is the last unlabelled outgoing edge of S, then S cannot have any edges leading to losing nodes.

An edge whose source is not labelled is only ever labelled if its target is labelled (during the assigning of the draws in `RetrogradeAnalysis`, the source is already permanently labelled a draw before the outgoing edges are labelled, and the assigning of a winning node in `AnalyseEdge` labels the source winning before it labels all of its outgoing edges). As such, if `AnalyseEdge` comes across the last unlabelled edge out of a certain unlabelled state, all this state's outgoing edges lead to labelled states.

If a state S is thus marked losing, it cannot have any edges leading to an unlabelled state, a losing state, or a draw state: all of its outgoing edges have to lead to winning states.

Hence, for all positional strategies x', every losing node has to lead to a winning node through x'.

This means that for every positional strategy x' for which every move from a winning node is given by x, every path starting in a winning or losing node following x' has to alternate between winning and losing nodes. □

Next we would like to prove something similar for draw nodes: that the successor of each draw node through Algorithm 2 is itself also a draw node.

Lemma 4. *Let x be a positional strategy provided by Algorithm 2. For all draw nodes S, its immediate successor S_n through x is also a draw node.*

Proof. A node S can be labelled a draw in two places: `AnalyseEdge` and `RetrogradeAnalysis`.

Assume S is assigned a draw by an `AnalyseEdge` call for (S, S_n). `AnalyseEdge` assigns S a Draw if and only if (S, S_n) is the last unlabelled edge out of S, S_n is either a Draw or a Win, and S has at least one outgoing edge to a draw node. The optimal move out of S is then also set such that S's successor is a draw node.

As stated before in the proof of Lemma 3, if `AnalyseEdge` comes across the last unlabelled edge out of a certain unlabelled state, all this state's outgoing edges lead to labelled states. If any of these states were to lead to a losing node, then `AnalyseEdge` would have already labelled S winning, meaning no outgoing edge of S can lead to a losing node.

If S is instead labelled by `RetrogradeAnalysis`, then in that call $S \in V_F$, and S's optimal move gets changed to lead to either a draw node directly, or to another node in V_F. That loop assigns all vertices in V_F to draws, meaning S's successor will in both cases lead to a draw node.

Note that the assignment of this optimal move is valid, because of the following observations: First, S has at least one outgoing edge, because $S \notin V_L$. Second, at least one of these outgoing edges has to lead to a node S_n that was also still unlabelled on line 29, because otherwise S would have been labelled by `AnalyseEdge`. For this reason as well, S cannot have any edges leading to losing nodes. Third, this node S_n cannot have a path in $A \cap {V_U}^2$ to a node in V_L, because otherwise S would have such a path as well.

As all of S's neighbouring nodes that were still unlabelled on line 29 are thus in V_F, and all nodes in V_F are subsequently labelled a draw, all neighbouring nodes to S must be labelled after this `RetrogradeAnalysis` call. None of these can be losing nodes, and at least one must be a draw, meaning the optimal move out of S is one such move leading to a draw node.

By Lemma 2, S then remains a Draw for the rest of the algorithm, with its optimal move continuing to lead to a draw node as well.

A draw node S thus always points to another draw node through x. □

Now we can prove that all paths following the positional strategies produced by Algorithm 2 actually deliver upon the quality they assign to each game state.

Lemma 5. *Let x be provided by Algorithm 2. For each positional strategy x' for which every move from a winning node is provided by x, every path following x' starting in a winning node terminates in an odd number of edges and thus contains no cycles.*

Proof. As there are only a finite number of game states, a path can only be infinite if it contains a cycle. A positional strategy is a deterministic mapping from game states to valid moves, meaning if a cycle is formed, it is followed in perpetuity.

Let x' be a positional strategy for which every move from a winning node is provided by x. Consider the path following x' starting in some winning node.

It is proven by Lemma 3 that through x' every winning node leads to a losing node, and that every losing node is either terminal or leads to a winning node. This means that through x' every winning node leads to either a terminal node, or a losing node that leads to another winning node.

There has to be a state S_f on our path that was labelled winning by our algorithm before any other state on our path: the *first* winning state on our path. If through x', S_f leads to a *non-terminal* node S_n, then S_n has to lead to another *winning* node S_w.

For S_f to be labelled a Win by our algorithm, S_n needs to be labelled a Loss first. For S_n to be labelled a Loss without being terminal, all of its potential subsequent states—including S_w—need to be labelled a Win first, despite S_f being the first of the path to be labelled winning. This is only possible if $S_f = S_w$.

In this case, however, we have a bootstrap paradox: S_f can only point to S_n if S_n has been labelled a Loss beforehand. Similarly, S_n can only point to S_f if all successors of S_n—including S_f—have already been labelled a Win. S_n and S_f's labels are thus codependent, and could only have been labelled at the same time, which is impossible. We conclude that S_f cannot lead to a non-terminal node.

Hence, S_f has to lead to a terminal node, meaning our path in x' has to terminate in a losing node. As our path starts in a winning node and alternates between Wins and Losses, the path has to terminate in an odd number of edges and cannot contain a cycle. □

Lemma 6. *Let x be provided by Algorithm 2. For each positional strategy x' for which every move from a winning node is provided by x, every path following x' starting in a losing node S_L terminates in an even number of edges and contains no cycles.*

Proof. Let x' be a positional strategy for which every move from a winning node is provided by x, and let S_L be a losing node.

If $S_L \in \mathcal{T}$, then S_L has no outgoing edges, meaning the path following x' starting in S_L immediately terminates after 0 edges: an even number. Let $S_L \notin \mathcal{T}$.

Lemma 3 tells us S_L leads to a winning state through x'. Lemma 5 then tells us x''s path will from there eventually terminate in an odd number of edges, and will thus not contain any cycles. x''s path starting in S_L thus contains no cycles, and terminates in an even number of edges. □

Lemma 7. *Given a positional strategy x provided by Algorithm 2, every path defined by x starting in a draw node S_D contains a cycle consisting of exclusively draw nodes.*

Proof. We prove this by contradiction. Let S_D be a draw node.

Assume the path defined by x starting in S_D does not contain a cycle. As there is only a finite number of states, and every infinite path thus needs to contain a cycle, this path must have an endpoint S_e.

As S_e is an endpoint—and thus does not lead anywhere—it has to be a terminal state, and therefore has to be a losing node. By Lemma 4, the successor of every draw node through x is again a draw node, meaning S_e cannot be on the path starting in S_D: a contradiction. Hence, every path following x starting in a draw node has to contain a cycle. □

Putting this all together, we can prove forward-looking retrograde analysis provides a perfect positional strategy for each game state it labels.

Theorem 2. *Algorithm 2 defines a perfect positional strategy for each labelled state.*

Note that some states may still be unlabelled after Algorithm 2. We come back to this after the proof.

Proof. Let x be provided by Algorithm 2.

By Lemma 5, for every winning node, x provides a path leading to the current player's win, regardless of their opponent's (the even-numbered) moves. The defined path is thus optimal for the current player, as it guarantees them a win.

By Lemma 6, every path from a losing node leads to the current player's loss, regardless of their own (the odd-numbered) moves. As there is nothing the player can do to prevent their loss, the defined path is optimal for the current player.

By Lemma 7, every path in x starting in a draw node contains a cycle, continuing the game in perpetuity, ensuring the current player does not lose. In Lemma 4's proof it is also shown that no valid moves from a draw node can lead to a losing node, meaning any deviation from x by either player can at best move them to another draw node—not meaningfully altering their situation—and at worst move them to a winning node—providing their opponent the opportunity to win. The defined path is thus optimal for both players.

Hence, Algorithm 2 provides a positional strategy granting the optimal move for the current player for every labelled node: it defines a perfect positional strategy for each labelled game state. □

Just because the algorithm provides a perfect positional strategy covering all states it labels, this does not immediately imply that it weakly solves the game for all non-losing players, as there is no guarantee the algorithm ever labels any state—and most importantly S_0—at all. We continue by proving that Algorithm 2 always labels at least S_0, and thus always provides a perfect positional strategy from this initial state.

Lemma 8. *During Algorithm 2, if an* `Expand` *call on game state S returns earlier than the bottom of the function (earlier than the implicit return), then S or S_0 has been labelled.*

Proof. Along with the bottom of the function, there are three other places where `Expand` can return: lines 19, 26 and 29.

If the function returns on line 19, then S has a move leading to a (losing) terminal state, meaning the `RetrogradeAnalysis` call must have labelled S winning.

The return on line 26 is only taken if S or S_0 are labelled, and similarly the return on line 29 is only taken if S is labelled.

Hence, if an `Expand` call on game state S returns earlier than the bottom of the function, then S or S_0 has been labelled. □

Lemma 9. *During Algorithm 2, if an* `Expand` *call on game state S ends through the bottom of the function, then at least one of the following holds: S is labelled, there is no path in the graph from S through exclusively unlabelled vertices ending in an unlabelled leaf, or S_0 is labelled.*

Proof. Assume, for the sake of contradiction, that after a certain game state S's `Expand` call S is still unlabelled, it has a path through exclusively unlabelled vertices ending in an unlabelled leaf and S_0 is not yet labelled.

By Lemma 8, each call to `Expand` that does not return through the bottom of the function either labels its associated state, or labels S_0.

Because neither S nor S_0 are labelled, S's `Expand` call must have returned through the bottom of the function. As such, it must have passed the for loop that recursively calls `Expand` on all of S's unexpanded successors.

Crucially, this means that at the end of S's `Expand` call, S's successor in the path must also have been expanded. Because the path goes through exclusively unlabelled vertices, this `Expand` call must have also returned through the bottom of the function.

This recursive relationship continues, ensuring every game state in the path has had `Expand` called upon it.

Let S_l be the *leaf* node of this path, and S_p be S_l's *predecessor* in the path.

Because `Expand` was called upon S_p, all of its valid moves have been added to the graph. If any of these moves lead to a terminal state, that state would have been labelled losing. As S_l is unlabelled, S_l is non-terminal.

In S_l's `Expand` call, all of its valid moves were added to the graph as well. As S_l is a leaf, however, it has no valid moves. By Definition 9, only terminal states can have no valid moves: contradiction.

As such, if an `Expand` call on S ends through the bottom of the function, then S is labelled, or there is no path in the graph through exclusively unlabelled vertices ending in an unlabelled leaf, or S_0 is labelled. □

Theorem 3. *Algorithm 2 weakly solves Onitama for all non-losing players, given enough computational resources.*

Proof. By Theorem 2, Algorithm 2 provides a perfect positional strategy for every labelled node. To prove that this algorithm weakly solves Onitama for all non-losing players, we thus only have to prove that it labels the initial state S_0. Assume S_0 is unlabelled.

By Lemma 2, the algorithm never strips the label from an already labelled state—once a state is labelled, it will preserve that label for the rest of the algorithm. For S_0 to be unlabelled it can thus never be assigned a label throughout the entire algorithm.

By Lemma 8, if the `Expand` call on S_0 did not return through the bottom of the function, S_0 would have been labelled. Because S_0 is unlabelled, it returned through the bottom of the function. As such, by Lemma 9, there is no path in the graph through exclusively unlabelled vertices ending in an unlabelled leaf.

Because S_0 is unlabelled, `RetrogradeAnalysis` is called after S_0's `Expand` call. Because S_0 is unlabelled, it cannot be labelled by any one of this function's `AnalyseEdge` calls.

However, $S_0 \in V_F$, because S_0 is unlabelled and has no path in $A \cap {V_U}^2$ from S_0 to a node in V_L. Therefore, S_0 will be labelled a draw in the subsequent loop over V_F.

This induces a contradiction, meaning the initial state S_0 will always be labelled by Algorithm 2. Therefore, this algorithm weakly solves Onitama for all non-losing players. □

Algorithm 2 does not necessarily weakly solve Onitama for a losing player, as this algorithm prunes all other branches out of a winning node once an edge to a losing node is found. This means that if the current player in a winning game state were to perform a suboptimal move—one that has not been analysed by the algorithm—there is no guarantee the game will end up in a game state with a defined optimal move.

For a node to be marked as either losing or a draw, all outgoing edges must have been analysed, meaning even if the current player in such a node plays suboptimally, the following game state will have still been analysed, and an optimal move will be defined.

Therefore, by Lemmas 3 and 4, Algorithm 2 only guarantees a weak solution for non-losing players: it provides a perfect positional strategy from all game states that are marked winning or a draw, regardless of what moves their opponent performs, because the optimal move out of these game states lead to either a losing or a draw node, for which all outgoing edges have been analysed.

4.2 Complexity

Algorithm 2 is a recursive algorithm. Because every `Expand` call immediately adds its subject to V_E, and `Expand` is only ever recursively called on game states not in V_E, `Expand` is at most evaluated once for each game state.

By far the most expensive line in `Expand` (excluding its own recursive call) is the call to Algorithm 1's function `RetrogradeAnalysis`, which is itself of complexity $\Theta(|A|)$[And+10]. This call works on the currently discovered graph G, and keeps track of what edges have already been labelled in prior calls. Therefore, all calls to `RetrogradeAnalysis` can at most incur an equal cost to calling `RetrogradeAnalysis` once on the full state graph.

The rest of the algorithm is mostly linearly bounded by the two loops over all outgoing edges of S, meaning these operations through the entire algorithm incur at most a cost of complexity $O(|A|)$ of the full graph—in the theoretical case that every game state has all of its edges analysed.

In total, the worst-case complexity of forward-looking retrograde analysis is thus $O(|A|) + O(|A|) = O(|A|)$—the same worst-case complexity as regular retrograde analysis. Furthermore, in the absolute worst case, Algorithm 2 will have to map out the full state space anyway, not providing any benefit to retrograde analysis.

While the worst-case complexity is just as bad as retrograde analysis, due to it analysing the in-progress graph while it is still being built up, it is often able to exit far before the entire state space is mapped out. This gives forward-looking retrograde analysis a significantly better lower bound than retrograde analysis, which requires the full state graph in all cases.

Next we look at some results forward-looking retrograde analysis provides in real-world experimentation.

4.3 Results

The algorithm has been executed on a variety of board sizes. The average amount of time these runs took are displayed in Table 8.

Comparing Table 8 with the runtimes for retrograde analysis (including graph construction) in Table 2, we see a blanket improvement in runtime for forward-looking retrograde analysis compared to retrograde analysis.

There is a particularly large difference for the 3×4 and 4×3 boards, which took retrograde analysis 4503.315 s and over two hours respectively, whereas forward-looking retrograde analysis only took 201.976 s and 250.325 s respectively—over 22 times faster for the former and at least 28 times faster for the latter.

m \ n	1	2	3	4	5
2	0.000 s	0.000 s	0.000 s	0.000 s	0.000 s
3	0.000 s	0.000 s	0.042 s	250.325 s	–
4	0.000 s	0.001 s	201.976 s	–	–
5	0.000 s	0.039 s	–	–	–

Table 8: Average runtime for forward-looking retrograde analysis on various board sizes $n \times m$ with exclusively *boar* cards. All tests without results exceeded two hours in runtime and were terminated. Averaged over 100 runs.

We also quantified the number of distinct game states forward-looking retrograde analysis visited during its execution across different board sizes. These results are displayed in Table 9.

Comparing Table 9 with the total number of reachable game states from the initial state displayed in Table 6, we immediately notice a few characteristics.

Column $n = 1$ is completely identical between the two. This is because with a single-tile-wide board, both players always have only one move they can perform: move their only pawn forward. This means forward-looking retrograde analysis has no choice but to visit every reachable game state, as that is the only possible path through the game.

The difference seems greatest in row $m = 2$, as for these cases forward-looking retrograde analysis could simply observe that the starting player can in their first move capture the master of their opponent by moving one of their pawns forward, after which the algorithm can immediately exit with that optimal move, whereas retrograde analysis would still have to analyse all other possible moves to completion.

m \ n	1	2	3	4	5
2	2	3	4	5	6
3	3	17	4068	161 138	–
4	4	260	144 354	–	–
5	5	3480	–	–	–

Table 9: Number of game states analysed by forward-looking retrograde analysis on various board sizes $n \times m$ with exclusively *boar* cards. All tests without results exceeded two hours in runtime and were terminated.

The difference in the number of game states visited between retrograde analysis and forward-looking retrograde analysis also seems to decrease as the height m of the board increases. This likely has to do with the height of the board determining how far apart the opposing pawns start, meaning taller boards allow for more configurations before any pawns get captured. As such, a lot more game states need to be considered before the algorithm stumbles upon a winning strategy, and can guarantee its correctness.

The growth in runtime and the number of states that need to be analysed is still rapid, making it unlikely for this algorithm in its current state to find a weak solution for all non-losing players for Onitama on a 5×5 board with a reasonable amount of computational resources. We therefore have to look towards other methods to decrease the number of game states that would need to be analysed, which could in turn decrease the runtime of the algorithm.

5 State symmetries

Many game states, although technically distinct, are functionally identical when it comes to analysing their strategies. Identifying these *symmetries* could cut down on the number of game states that would need to be analysed in finding a perfect positional strategy, potentially allowing for more complicated games to be solved.

In this section we discuss one such useful symmetry, as well as its performance when applied on forward-looking retrograde analysis.

5.1 Player-swapping

Take the colours of the players. While vitally important for keeping track of which pawns and cards belong to which player, no rule in the game assigns importance to the specific colours themselves—merely to the (in)equality of colour across different game elements.

Let $S = (\pi, B, C_s, C_R, C_B) \in X$ be a game state, and define a function $\rho\colon T^{n\times m} \to T^{n\times m}$ that rotates a board halfway and swaps the colours: $\rho(B)_{xy} = \overline{B_{(n+1)-x,(m+1)-y}}$ for all $B \in T^{n\times m}, (x, y) \in [1..n] \times [1..m]$. Then for determining a perfect positional strategy, S is equivalent to $\overline{S} := (\overline{\pi}, \rho(B), C_s, C_B, C_R)$, i.e., game state S with the players swapped.

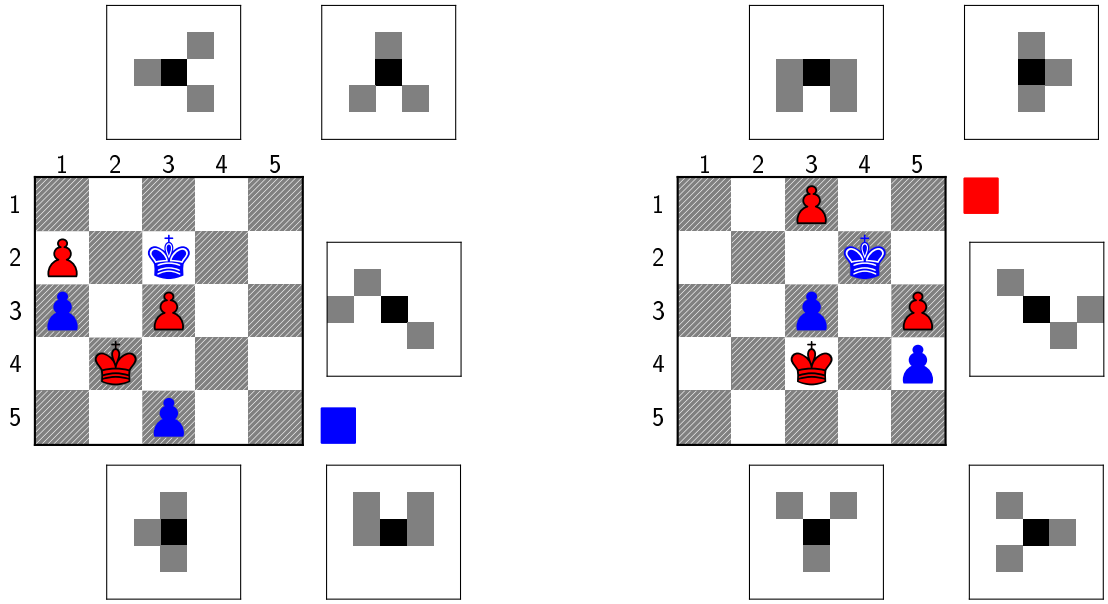

Figure 10: Two player-swapped boards. Note that the board effectively rotates halfway, with the colours of the pawns swapping. All the cards (including the set-aside card) rotate accordingly, as they have to swap to their new owner's (or, in case of the set-aside card, owner-to-be's) perspective.

First note that as the two temples are located on tiles $\left(\lfloor\frac{n+1}{2}\rfloor, 1\right)$ and $\left(\lceil\frac{n+1}{2}\rceil, m\right)$, after player-swapping these locations become respectively $\left(n+1-\lfloor\frac{n+1}{2}\rfloor, m+1-1\right) = \left(\lceil\frac{n+1}{2}\rceil, m\right)$ and $\left(\lfloor\frac{n+1}{2}\rfloor, 1\right)$; the temples effectively swap colours.

Also note that $(\rho \circ \rho(B))_{xy} = \overline{\overline{B_{(n+1)-[(n+1)-x],(m+1)-[(m+1)-y]}}} = \overline{\overline{B_{xy}}} = B_{xy}$, meaning $\rho \circ \rho = \mathrm{id}$. Furthermore, this means that $\overline{\overline{S}} = (\overline{\overline{\pi}}, \rho \circ \rho(B), C_s, C_R, C_B) = (\pi, B, C_s, C_R, C_B) = S$; player-swapping is an involution.

We wish to prove player-swapping is a symmetry on game states when it comes to finding perfect positional strategies. In doing so, we need to prove all the moves between these game states have a symmetric equivalent as well.

Lemma 10. *For every game state $S \in X$, $(S, S') \in A$ if and only if $(\overline{S}, \overline{S'}) \in A$.*

Proof. Let $S = (\pi, B, C_s, C_R, C_B)$ and $m = (o, d, C) \in M_S$. As m is valid, $C \in C_\pi$, $d - o \in C$, $B_o \in \{m_\pi, s_\pi\}$ and $B_d \notin \{m_\pi, s_\pi\}$. We show that for $\overline{S} = (\overline{\pi}, \rho(B), C_s, \overline{C_R} := C_B, \overline{C_B} := C_R)$, move $\overline{m} := ((n+1, m+1) - o, (n+1, m+1) - d, C) \in M_{\overline{S}}$.

First note that $C \in \overline{C_{\overline{\pi}}} = C_\pi$. By definition, $\rho(B)_{(n+1,m+1)-o} = \overline{B_{(n+1,m+1)-((n+1,m+1)-o)}} = \overline{B_o} \in \{m_{\overline{\pi}}, s_{\overline{\pi}}\}$ and similarly $\rho(B)_{(n+1,m+1)-d} \notin \{m_{\overline{\pi}}, s_{\overline{\pi}}\}$. This means that $\overline{m}$ is a valid move out of $\overline{S}$: $\overline{m} \in M_{\overline{S}}$.

Next we show that $\overline{m}(\overline{S}) = \overline{m(S)}$.

Applying a move flips the current player, and so does player-swapping. Hence the current player in both $\overline{m}(\overline{S})$ and $\overline{m(S)}$ is equal to $\overline{\overline{\pi}} = \pi$.

Player-swapping swaps the two players' hands, and applying a move swaps the used card in the current player's hand with the one that is set aside. $\overline{\pi}$'s hand in $m(S)$ is $C_{\overline{\pi}}$, meaning this is π's hand instead in $\overline{m(S)}$. π's hand in $m(S)$ is $(C_\pi \setminus \{C\}) \cup \{C_s\}$, meaning this is $\overline{\pi}$'s hand in $\overline{m(S)}$. Player-swapping does not alter the set-aside card, so the set-aside card in both $m(S)$ and $\overline{m(S)}$ is C.

In $\overline{S}$, $\overline{\pi}$ is the current player, meaning through $\overline{m}$ the hand belonging to π stays invariant. Therefore π's hand in $\overline{m}(\overline{S})$ is π's hand in $\overline{S}$, which is $\overline{\pi}$'s hand in S: $C_{\overline{\pi}}$. In $\overline{S}$, the current player $\overline{\pi}$'s hand is C_π, meaning $\overline{\pi}$'s hand in $\overline{m}(\overline{S})$ is $(C_\pi \setminus \{C\}) \cup \{C_s\}$. Player-swapping does not alter the set-aside card, so the set-aside card in $\overline{S}$ is still C_s. In $\overline{m}(\overline{S})$ this changes to the used card C. The set-aside card and both players' hands are thus equal across $\overline{m(S)}$ and $\overline{m}(\overline{S})$.

For the board in $m(S)$ we have that the tiles at the coordinates $t \neq o, d$ are equal to B_t. Player-swapping rotates the board halfway and flips the colours, so for game state $\overline{m(S)}$ we have that all the tiles at the coordinates $t \neq (n+1, m+1) - o$ or $(n+1, m+1) - d$ are equal to $\overline{B_{(n+1,m+1)-t}}$.

Player-swapping first means that in state $\overline{S}$ all tiles at the coordinates $t \neq (n+1, m+1) - o$ or $(n+1, m+1) - d$ are equal to $\overline{B_{(n+1,m+1)-t}}$. Move $\overline{m}$, having origin $(n+1, m+1) - o$ and destination $(n+1, m+1) - d$, changes nothing across these tiles, meaning these tiles in the board of $\overline{m}(\overline{S})$ are also equal to $\overline{B_{(n+1,m+1)-t}}$.

In the board of $m(S)$, tile o is emptied to $\varnothing$. After player-swapping this means that in the board of $\overline{m(S)}$ tile $(n+1, m+1) - o$ is empty instead. For $\overline{m}$ the origin is $(n+1, m+1) - o$, meaning tile $(n+1, m+1) - o$ is also empty in the board of $\overline{m}(\overline{S})$.

Finally, in the board of $m(S)$, the destination tile d is set to B_o. After player-swapping, this means tile $(n+1, m+1) - d$ in the board of state $\overline{m(S)}$ is equal to $\overline{B_o}$. For $\overline{m}$, the origin is $(n+1, m+1) - o$, which in the board of $\overline{S}$ is equal to $\overline{B_o}$. Therefore, tile $(n+1, m+1) - d$ in the board of state $\overline{m}(\overline{S})$ is also equal to $\overline{B_o}$.

The current player, the set-aside card, both coloured hands and every tile of the board are equal between $\overline{m}(\overline{S})$ and $\overline{m(S)}$: $\overline{m}(\overline{S}) = \overline{m(S)}$ for all $S \in X$ and all $m \in M_S$.

Let $(S, S') \in A$. Then there exists an $m \in M_S$ such that $m(S) = S'$. For $\overline{S}$ there is then the valid player-swapped move $\overline{m}$, for which $\overline{m}(\overline{S})$ is thus well defined. As we have just proven, $\overline{m}(\overline{S}) = \overline{m(S)} = \overline{S'}$, meaning there is a valid move from $\overline{S}$ to $\overline{S'}$. By definition of the state graph, $(\overline{S}, \overline{S'}) \in A$.

Due to player-swapping being an involution, the same holds from $\overline{S}$ to $\overline{\overline{S}} = S$, meaning for every game state $S \in X$, $(S, S') \in A$ if and only if $(\overline{S}, \overline{S'}) \in A$. □

In order for this symmetry to be useful in finding perfect positional strategies, we need to prove a move $m \in M_S$ is optimal for a game state S if and only if $\overline{m}$ is optimal for $\overline{S}$. For this, we need to prove the quality of S assessed by a perfect positional strategy remains invariant after player-swapping to $\overline{S}$.

Lemma 11. *For every perfect positional strategy x and all game states $S \in X$, $Q_x(S) = Q_x(\overline{S})$.*

Proof. By definition of Q_x, x provides a non-empty collection $\mathcal{P}$ of paths starting in S where all of S's current player's moves are given by x, all of which result in an outcome of at least $Q_x(S)$.

By Lemma 10, $\mathcal{P}$ can be associated with a collection of equivalent player-swapped paths $\overline{\mathcal{P}}$ starting in $\overline{S}$. $S \in \mathcal{T}$ if and only if $\overline{S} \in \mathcal{T}$, which provides the following properties:

- if $Q_x(S) = Win$, all paths in $\mathcal{P}$ end in a terminal state after an odd number of moves;
- if $Q_x(S) = Draw$, these paths either terminate in an odd number of moves (in case the opponent plays suboptimally) or do not terminate at all;

- if $Q_x(S) = \mathit{Lose}$, regardless of what the current player in S does, there exists a path that ends in a terminal state in an even number of moves.

The paths in $\mathcal{P}$ explore all possible moves the opponent in S could take. By Lemma 10, as player-swapping cannot add or remove moves, this means $\overline{\mathcal{P}}$ explores all possible moves the opponent in $\overline{S}$ could take. As $\overline{\mathcal{P}}$ is non-empty, the paths in $\overline{\mathcal{P}}$ then define part of a perfect positional strategy guaranteeing the current player in $\overline{S}$ can secure themselves an outcome of at least $Q_x(S)$ regardless of what the opponent in $\overline{S}$ does: $Q_x(\overline{S})$ is at least as good as $Q_x(S)$.

As player-swapping is an involution, this proof also holds from $\overline{S}$ to $\overline{\overline{S}} = S$, meaning $Q_x(S)$ is at least as good as $Q_x(\overline{S})$, ultimately proving $Q_x(S) = Q_x(\overline{S})$. □

Theorem 4. *For determining a perfect positional strategy, game state $S \in X$ is equivalent to $\overline{S}$.*

Proof. By Lemma 10, a move $m \in M_S$ exists if and only if there is an equivalent player-swapped move $\overline{m} \in M_{\overline{S}}$ that leads to $\overline{m(S)}$.

By Lemma 11, $Q_x(S) = Q_x(\overline{S})$, and also $Q_x(m(S)) = Q_x(\overline{m(S)}) = Q_x(\overline{m}(\overline{S}))$ for all $m \in M_S$, i.e., the quality of game state S and that of all of its successors are equal to their player-swapped counterparts. Therefore, the optimality of all moves in M_S remains invariant under player-swapping.

This means that for any game state S, all local information provided by perfect positional strategies (its optimal and redundant moves, as well as the state's quality) can directly be applied to its player-swapped state $\overline{S}$, and vice versa. For determining a perfect positional strategy, game state $S \in X$ is equivalent to $\overline{S}$. □

5.2 Results

Some tests were performed to observe to what degree this symmetry aids in reducing the number of visited game states, and how much that affects the runtime of the algorithm. The average runtime and number of analysed game states across a variety of board sizes for forward-looking retrograde analysis can be found in Tables 10 and 11.

m \ n	1	2	3	4	5
2	0.000 s	0.000 s	0.000 s	0.000 s	0.000 s
3	0.000 s	0.000 s	0.020 s	38.992 s	–
4	0.000 s	0.001 s	76.396 s	–	–
5	0.000 s	0.031 s	–	–	–

Table 10: Average runtime for forward-looking retrograde analysis on various board sizes $n \times m$ with exclusively *boar* cards, making use of player-swapping symmetries. All tests without results exceeded two hours in runtime and were terminated. Averaged over 100 runs.

m \ n	1	2	3	4	5
2	2	3	4	5	6
3	3	17	2285	72 657	–
4	4	254	94 412	–	–
5	5	2815	–	–	–

Table 11: Number of game states analysed by forward-looking retrograde analysis on various board sizes $n \times m$ with exclusively *boar* cards, making use of player-swapping symmetries. All tests without results exceeded two hours in runtime and were terminated.

Comparing these values with the ones in Tables 8 and 9, we once again see a blanket improvement in both runtime and number of game states analysed with the symmetries applied.

We observe that column $n = 1$ and row $m = 2$ are identical across the two tables. For $n = 1$, this is for the same reason as laid out in Section 4.3: there is only one path that can be taken on a board of width 1. For $m = 2$ the game ends immediately after the first move, not giving the opponent any opportunity to perform a move. As such, player-swapping has no effect, because the symmetry swaps the current player as well, which is of no importance if the game ends in a single move. In both cases, the player-swapping symmetry thus does not provide any utility, because no player-swapped equivalent to a previously analysed state will ever be analysed.

For all board sizes the number of game states visited when using symmetries is at most as much as are visited without using symmetries, with their difference increasing as the size of the board increases. This likely has to do with the fact that a larger state space overall increases the chances of coming across a symmetric state, whereas small state spaces indicate simple games that can end before such a symmetric state can be realised.

In accordance with this trend, we also see the difference in runtime increase as the size of the board increases, with the runtime using symmetries never being longer than the one without symmetries. This time decrease can be quite significant, with the runtime for the 4×3 board decreasing by roughly 84.42 % when making use of symmetries. The number of game states analysed in the process decreases to merely 45.09 % of the number analysed without this symmetry applied.

The larger decrease in runtime compared to the decrease in the number of states analysed can be explained by how the state graph that is worked with is often smaller when using symmetries. With a smaller state graph, some repeated operations in the algorithm whose runtimes depend on the size of the graph can then be performed quicker. For example, within the implementation of Algorithm 1, the set of vertices in the state graph is iterated over in order to construct the subset of unlabelled, expanded vertices. With a smaller overall state graph, this operation, and in turn each call to `RetrogradeAnalysis`, will resolve more quickly.

Furthermore, the testing application uses a `std::unordered_map` and a `std::unordered_set` as underlying data structures for the state graph and the unlabelled edges in retrograde analysis respectively. These data structures have faster lookups if there are fewer collisions within these structures, which naturally occurs less often if there are fewer vertices in these structures in the first place.

Finally, if a symmetric state is found, it could at that point already be labelled due to its symmetric equivalent having been labelled before. As such, any expanding state that leads to this symmetric state could potentially immediately be labelled without having to expand any subsequent states that would normally have been expanded if no symmetries were used. Therefore, making use of symmetries can induce a greater decrease in the number of states analysed than the decrease of the full state space this symmetry provides.

Player-swapping thus has a meaningful impact on the runtime of forward-looking retrograde analysis. Regardless, as player-swapping always cuts the full search space in half at best, this symmetry does not reduce the complexity of the algorithm, meaning a 5×5 solution still seems to be out of reach with this approach.

6 Conclusion and further research

6.1 Conclusion

In this thesis we applied retrograde analysis to the board game Onitama. We found that Onitama is too complex of a game for retrograde analysis to strongly solve, having to analyse all game states reachable from the initial state, with the number of board states bounded from below and above respectively by

$$\Omega([m-2]^{n-1}) \quad \text{and} \quad \mathcal{O}\left(\frac{n^2(nm)!}{([n-1]!)^2(n[m-2])!}\right)$$

in relation to the game's board size $n \times m$. As the number of reachable game states is at most a constant factor larger than the number of reachable board states, these complexity values can also be applied to the size of the game's state graph itself.

By experimentation, most of retrograde analysis' runtime is spent building up the state graph. A forward-looking derivative of retrograde analysis was discussed, which can analyse a state graph as it is still being constructed. This allows the algorithm to exit the moment it finds a weak solution from the initial state for all non-losing players, without having to construct and analyse the full state graph.

Forward-looking retrograde analysis was formally proven to provide weak solutions to the game for all players that cannot lose under optimal play from the initial state. Forward-looking retrograde analysis was shown to provide significantly faster results than retrograde analysis, despite having equivalent worst-case complexity. On a 3×4 board with exclusively *boar* cards, forward-looking retrograde analysis found a solution 95.51 % faster than retrograde-analysis.

In addition, a symmetry was discussed which could be applied on the game states with regard to finding perfect positional strategies, reducing the state space even further. This symmetry was also shown to provide significant time improvements when applied to forward-looking retrograde analysis.

Despite all these advancements, the game (using only *boar* cards) was only able to be strongly solved by retrograde analysis up to a 3×4 board, with forward-looking retrograde analysis only being able to improve this by finding a weak solution for all non-losing players up to a 4×3 board. All larger board sizes took over two hours on both algorithms. Forward-looking retrograde analysis making use of symmetries was able to solve the 3×4 board roughly 98.30 % faster than retrograde analysis could while not making use of symmetries. Solving a 4×3 board using forward-looking retrograde analysis was shown to take roughly 84.42 % less time if symmetries were used, compared to the same algorithm not making use of symmetries.

6.2 Further research

During our studies we were not able to fully solve Onitama with a 5×5 board in a reasonable amount of time. There are several approaches that could still be investigated to improve upon our results, and potentially bring a solution for the 5×5 board within reach:

- One could apply a heuristic upon each move during forward-looking retrograde analysis, to improve the chances of stumbling across the perfect positional strategy, allowing it to exit with even fewer states analysed.
- The state graph generation is by far the most resource-intensive computation for retrograde analysis. Graph exploration lends itself well to parallelisation[HOO11]. Therefore, a well-performing multithreaded graph-generating algorithm could be used in conjunction with retrograde analysis to theoretically improve runtime.
- The codebase used for testing is written in a way that prioritises readability, extendability and modifiability, as this thesis mostly concerned itself with theory as opposed to practical results. In some cases, these prioritisations degrade the program's performance. Lengths could be taken to rewrite the codebase in a more performant manner, or run the program on more powerful hardware for a longer amount of time.

Apart from these improvements that could be made on (forward-looking) retrograde analysis in context of Onitama, the game itself also has a number of unanswered questions that promise an interesting study:

- One could more closely research what game states are reachable given a specific set of cards, and improve the upper and lower bounds found in this thesis.
- More symmetries could be discovered and analysed to further reduce the state space of the game.
- This game is very complex, and does not have any trivial metric that could be applied on heuristic strategies for finding a good, but not necessarily optimal, move to be performed. Research could be done into how game states can be scored against one another to improve heuristic strategies.
- Research on the perfect positional strategies themselves could be performed to distil (parts of) a strategy that can be used by people in real-world play.
- At the moment of the publication of this thesis there are three expansion sets to the game: *Sensei's Path*, *Way of the Wind*, and *Light & Shadow*[OniEx]. While *Sensei's Path* merely adds sixteen new cards to the game, *Way of the Wind* introduces a new pawn that can be moved by

both players, and *Light & Shadow* adds elements of imperfect information to the game. These expansions invite many new questions to study and gameplay mechanics to analyse.

Finally, forward-looking retrograde analysis could also be studied in the context of other chess-like games, comparing its performance to retrograde analysis in those scenarios.

References

[And+10] Daniel Andersson et al. 'Deterministic Graphical Games Revisited'. In: *Journal of Logic and Computation* 22.2 (Feb. 2010), pp. 165–178. ISSN: 0955-792X. DOI: https://doi.org/10.1093/logcom/exq001.

[Arn21] Robert Arntzenius. 'Agents for the Strategy Game Onitama'. Bachelor's Thesis. Leiden University, 2021. URL: https://theses.liacs.nl/pdf/2020-2021-ArntzeniusR.pdf.

[Boe24] Teun Boekholt. 'How does using Theory of Mind in an Agent-Based Model Influence the Results of the Two-Player Board Game of Onitama'. Bachelor's Thesis. University of Groningen, 2024. URL: https://fse.studenttheses.ub.rug.nl/33601/1/BachelorThesisTeunBoekholt.pdf.

[Ewe02] Christian Ewerhart. 'Backward Induction and the Game-Theoretic Analysis of Chess'. In: *Games and economic behavior.* 39.2 (2002), pp. 206–214. ISSN: 0899-8256. DOI: https://doi.org/10.1006/game.2001.0900.

[HOO11] Sungpack Hong, Tayo Oguntebi and Kunle Olukotun. 'Efficient Parallel Graph Exploration on Multi-Core CPU and GPU'. In: *2011 International Conference on Parallel Architectures and Compilation Techniques.* 2011, pp. 78–88. DOI: 10.1109/PACT.2011.14.

[ISR02] Hiroyuki Iida, Makoto Sakuta and Jeff Rollason. 'Computer shogi'. In: *Artificial intelligence* 134.1 (2002), pp. 121–144. ISSN: 0004-3702. DOI: https://doi.org/10.1016/S0004-3702(01)00157-6.

[OniEx] *Onitama.* Arcane Wonders. URL: https://www.arcanewonders.com/product/onitama/#eael-advance-tabs-7eea470 (visited on 25/05/2025).

[Sat] Shimpei Sato. *Onitama Rulebook.* Arcane Wonders. URL: https://www.arcanewonders.com/wp-content/uploads/2021/05/Onitama-Rulebook.pdf (visited on 12/02/2025).

[Sch+07] Jonathan Schaeffer et al. 'Checkers Is Solved'. In: *Science* 317 (Oct. 2007), pp. 1518–1522. DOI: 10.1126/science.1144079.

[Wer04] Erik van der Werf. 'AI techniques for the game of Go'. PhD thesis. Maastricht University, 2004. ISBN: 90 5278 445 0. URL: https://project.dke.maastrichtuniversity.nl/games/files/phd/Van%20der%20Werf_thesis.pdf.

Glossary

board A *board* B is an $n \times m$ grid of tiles: $B \in T^{n \times m}$, with $n \geq 1$, $m \geq 2$. n and m denote the width and height of the board respectively. For any board B, B_{xy} denotes the tile in the x-th column from the left and the y-th row from the top, where the *red* player is always seated at the top.

capture When a pawn moves onto a tile holding a pawn of its opposing colour, the opposing-coloured pawn is *captured*, and taken out of play.

card A *card* C depicts a set of offsets from the origin. A card permits the player to move any of their pawns in accordance with any of these offsets. $C \subset [-2..2]^2$. Each card is also associated with a colour: $\chi \colon \mathcal{P}([-2..2]^2) \to \{r, b\}$. The colour of the set-aside card in the initial state dictates which player's turn it is first.

game state A *game state* $S = (\pi, B, C_s, C_R, C_B)$ is a 5-tuple consisting of the current player π (either red or blue), a board $B \in T^{n \times m}$ and five cards ($\subset [-2..2]^2$), in the order of the set-aside card C_s, the multiset of the two cards $C_R \coloneqq \{C_{r1}, C_{r2}\}$ belonging to the red player, and the multiset of the two cards $C_B \coloneqq \{C_{b1}, C_{b2}\}$ belonging to the blue player. X denotes the set of all game states reachable from the initial state.

master A type of pawn, depicted as either m_r or m_b for respectively a red and a blue master. Each player starts with one of them, and if this piece is captured or reaches the temple arch of the opponent, the game ends.

move A *move* $m \coloneqq (o, d, C) \in \mathcal{M} \coloneqq ([1..n] \times [1..m])^2 \times \mathcal{P}([-2..2]^2)$ holds an *origin* o and *destination* d, as well as a *card* C such that $d - o \in C$.

pawn A *pawn* p is either a master or a student, pertaining to a particular player π. $p \in P \coloneqq \{m_r, m_b, s_r, s_b\}$. $\overline{p}$ denotes the same pawn type (master or student) of the other player (red $\leftrightarrow$ blue).

player A *player* π is either *red* (r) or *blue* (b). $\overline{\pi}$ denotes π's opponent.

state graph Let $G = (V, A)$ be a directed graph, with the nodes $V = X$ consisting of all reachable game states from the initial state, and the edges being between two states reachable via single valid move: $A \coloneqq \{(S, S') \in V^2 \mid \exists m \in M_S \text{ s.t. } m(S) = S'\}$. We call G the *state graph* for the game Onitama.

student A type of pawn, depicted as either s_r or s_b for respectively a red and a blue student. In an $n \times m$ game, each player starts with $n - 1$ students.

temple arch $t_r \coloneqq (\lfloor \frac{n+1}{2} \rfloor, 1)$ and $t_b \coloneqq (\lceil \frac{n+1}{2} \rceil, m)$ are the *temple arches* of respectively the red and blue players. These are the starting positions of the masters. If a master reaches the temple arch of opposing colour, the game ends.

terminal state A game state is called *terminal* if and only if at least one of the following holds:

- $\exists m \in \{m_r, m_b\} : B_{xy} \neq m \, \forall (x, y) \in [1..n] \times [1..m]$ (a master got captured and got taken off of the board);
- $B_{t_r} = m_b$ or $B_{t_b} = m_r$ (a master reached the temple arch of the opponent).

$\mathcal{T} \subset X$ denotes the set of all terminal states that are reachable from the initial state.

tile A tile t represents a position on a board and is either empty ($\varnothing$), or a pawn ($\in P$): $t \in T \coloneqq P \cup \{\varnothing\}$.

valid move A move $m = (o, d, C)$ is considered *valid* for a specific game state $S = (\pi, B, C_s, C_R, C_B)$ if and only if $C \in C_\pi$, $B_o \in \{m_\pi, s_\pi\}$ and $B_d \notin \{m_\pi, s_\pi\}$, i.e., if and only if the move is defined by one of the current player's cards, and it moves one of the current player's pawns to either an empty tile, or one occupied by an opponent's pawn.

Appendix

Data collection and source code

All data in this thesis was collected using a custom-made simulation of Onitama, which is publicly accessible under the MIT License via GitHub: https://github.com/PringlesGang/Onitama/. Version 1.1.1 of the software was used for all data collection, which can be downloaded here: https://github.com/PringlesGang/Onitama/releases/tag/v1.1.1.

All tests were performed in release mode on an AMD Ryzen 5 2600 CPU with 16 GB of DDR4 SDRAM. Tests were terminated and deemed to take 'too long' if they took more than two hours to process.

Reproduce data

The data collected throughout this thesis can be reproduced using the aforementioned program with the following commands. In all these commands, the variables `$Width` and `$Height` refer to the width n and height m of the game board respectively.

- The data in Tables 2, 3, 6 and 7 were collected by executing the program with the following arguments:

```
experiment stategraph retrograde-analysis 0 game --size $Width
    $Height --cards boar --disable-symmetries
```

- The data in Tables 8 and 9 were collected by executing the program with the following arguments:

```
experiment stategraph forward-retrograde-analysis game --size
    $Width $Height --cards boar --disable-symmetries
```

- The data in Tables 10 and 11 were collected by executing the program with the following arguments:

```
experiment stategraph forward-retrograde-analysis game --size
    $Width $Height --cards boar
```

Table 1 was filled using data corroborated by all three of these experiments, with only the 4×3 datapoint not being able to be corroborated by retrograde analysis, as its runtime exceeded two hours.