A Practical Algorithm for Chess Unwinnability

Miguel Ambrona
Independent Researcher, Madrid, Spain

Abstract

The FIDE Laws of Chess establish that if a player runs out of time during a game, they lose unless there exists no sequence of legal moves that ends in a checkmate by their opponent, in which case the game is drawn. The problem of determining whether or not a given chess position is *unwinnable* for a certain player has been considered intractable by the community and, consequently, chess servers do not apply the above rule rigorously, thus unfairly classifying many games.

We propose, to the best of our knowledge, the first algorithm for *chess unwinnability* that is sound, complete and efficient for practical use. We also develop a prototype implementation and evaluate it over the entire Lichess Database (containing more than 3 billion games), successfully identifying *all* unfairly classified games in the database.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms; Software and its engineering → Software libraries and repositories

Keywords and phrases chess, helpmate, unwinnability, timeout, dead position

Digital Object Identifier 10.4230/LIPIcs.FUN.2022.2

Related Version *Full Version*: `https://chasolver.org/FUN22-full.pdf`

Supplementary Material `https://github.com/miguel-ambrona/D3-Chess`

Acknowledgements Special thanks to Pooya Farshim, for very fruitful discussions and all his feedback; Elena Gutiérrez, for all her help and comments; Antonio Nappa, for providing the hardware; the Lichess Team; the Stockfish Team; and many others: `https://chasolver.org/acks.html`. I would also like to express my sincere gratitude to Andrew Buchanan and Andrey Frolkin, for sharing with me and letting me include two original compositions, and for all the feedback. Finally, I would like to thank Maarten Loeffler, for having found a gap in an earlier version of the proof of Lemma 8, and for all his comments. I am also very thankful to the other anonymous reviewers of FUN 2022, for their valuable time and careful reading of this manuscript.

1 Introduction

Chess clocks have been used since 1883 [14] and are an essential tool in (tournament) chess to enforce game termination. They introduce a reliable upper-bound on the duration of games, ensuring that players will not excessively delay the match. This is crucial for designing and respecting tournament schedules.

A chess clock consists of two adjacent and entangled (countdown) timers that can never run simultaneously. Each player is responsible for one of the timers and must complete the game before their timer gets down to zero. Otherwise the player would "flag", i.e., lose on time. During the game, the player with the turn must press the clock's button after making a move (this is not necessary in online chess). This action, which concludes the player's turn, will stop their timer and resume their opponent's timer, who now has the turn and must proceed analogously.

There exists a wide variety of *time controls* that specify the initial allotted time and (optionally) a time bonus after every prescribed number of moves, ranging from several hours (or even days) to just 15 seconds to complete the entire game [1]. What is common to all time controls is that running out of time leads to a defeat. But not always! The clock is

just a tool to guarantee that the game will finish, but actual game position is given a higher priority than the clock state. For example, if your last move has checkmated your opponent, you win, even if your timer got down to zero while executing it [7, Article 5.1.1]. Or if you just have the king (because all your other pieces were captured), you cannot win anymore, not even on time! This folklore rule is a particular case of the following more general rule described in Article 6.9 of the FIDE Laws of Chess [7].

> *... if a player does not complete the prescribed number of moves in the allotted time, the game is lost by that player. However, the game is drawn if the position is such that the opponent cannot checkmate the player's king by any possible series of legal moves.*

1.1 The problem

Thus, in order to rigorously apply Article 6.9, one must be able to tell whether or not a given position can be won by the player who still has time on their clock.

▶ Remark 1. A position being *winnable* does not mean that a certain player can force a victory. Rather, it refers to the existence of a sequence of legal moves that ends in a checkmate by the player. Such sequence, which typically contains a poor choice of moves, is sometimes referred to as a *helpmate* [22].

Deciding whether or not a position is unwinnable, i.e. whether a helpmate does not exist, is usually relatively simple for a human. For example, it is not very hard to realize that no player can deliver checkmate in Position 1 (it is a so-called *dead position*), since the pawn wall is blocked and the bishops are not useful to make any progress. According to the FIDE Laws of Chess [7, Article 5.2.2] the game is finished as soon as the position becomes dead. No further moves are permitted and would be considered illegal.[1]

However, other such positions can be more involved. For example, it is not so easy to understand/prove why Position 2 is also dead (White to move).[2] Interestingly, if in Position 2 the pawn on a4 were on a5, the position would be winnable for White. Indeed, the following is a possible helpmate sequence in that case:

1 ♔a6 ♔e8 2 ♗b7 ♔d8 3 ♗c8 ♔e8 4 ♗d7+ ♔d8 5 ♗e8 ♔c8 6 ♗f7 ♔d8 7 ♗g8 ♔e8 8 ♔b7 ♔d8 9 g5 ♔e8 10 ♔c8 a4 11 ♗f7#

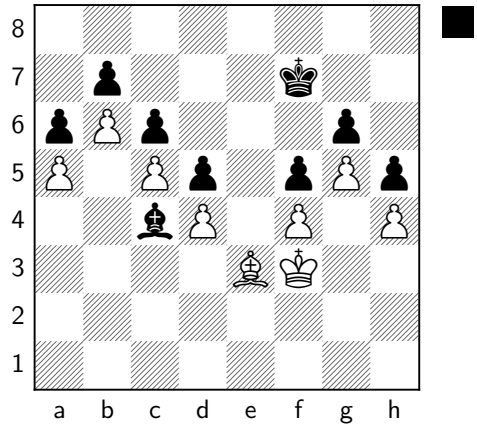
Note how the pawn being on a5 gave Black an additional tempo on move 10; without it, Black would have been in stalemate. This position, devised by the prominent chess composer Andrew Buchanan[3], evidences the hardness of deciding unwinnability, as very subtle changes in the position can alter the result.

---

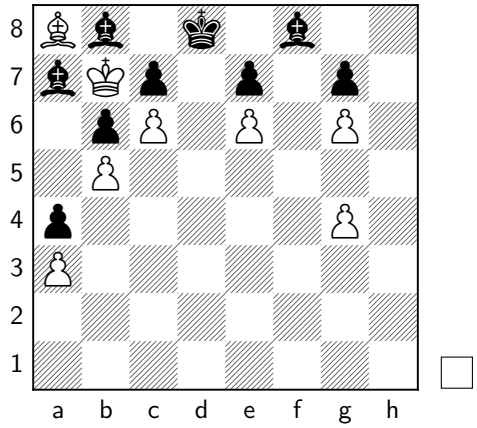
[1] This rule was introduced on July 1, 1997, and gave birth to a completely new genre of chess compositions called *dead reckoning* [3, 4, 20]. See Appendix C for two original compositions of this kind.

[2] The player to move is indicated by either a white or a black box adjacent to the diagram.

[3] The original composition asks: *What was the last move?* (In Position 2). Since it is White to move, the last move must have been either ... ♔e8–d8 or ... a5–a4, but only one of these moves comes from a position that is not dead. It's instructive to guess which one is the case here!



Position 1 Lichess game `tLUsotyi`.



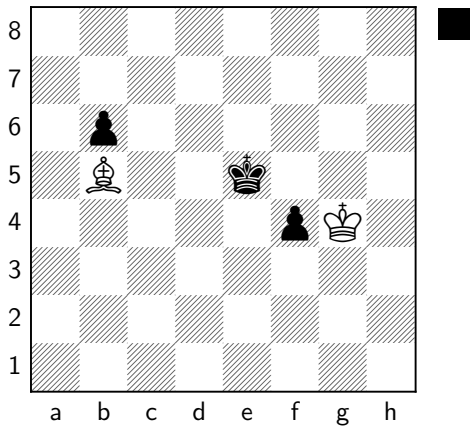
Position 2 A. Buchanan, *StrateGems 2002*.

1.2 Related work

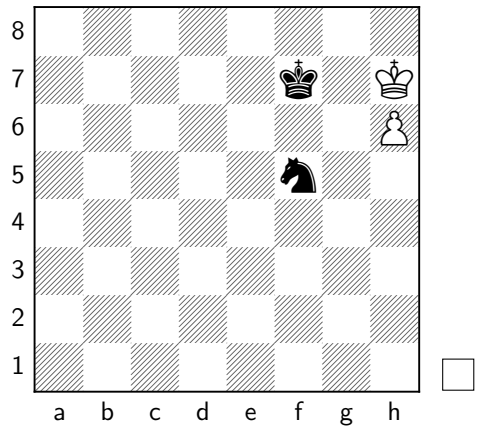
Online chess. Chess servers are an important point of reference to understand what the current state-of-the-art with respect to chess unwinnability is. It turns out that given the apparent complexity of deciding unwinnability, chess servers only analyze whether the intended winner has sufficient material to checkmate. Indeed, the three most popular chess servers adjudicate timeouts to date as follows.

- Chess.com, currently the Internet's biggest online chess server, declares a position as drawn if, after a timeout, the player with time on the clock has *insufficient material* [5, 16, 17]. Namely, if they have (i) a lonely king, (ii) a king and a bishop, (iii) a king and a knight, (iv) a king and two knights. This decision is supported by the claim that they do not follow the FIDE Laws of Chess for adjudicating timeouts and instead follow the USCF rules, which specify that the game is drawn (in case of insufficient material) if there is no *forced mate* by the intended winner.
- Lichess.org, one of the most popular chess websites in the world while remaining 100% free/libre and open-source, focuses on positions without pawns. Unlike other servers, *Lichess never declares a position as unwinnable when it is indeed winnable.* (Although it fails to identify all unwinnable positions.) In particular, Lichess correctly classifies all insufficient material positions that do not contain pawns. For example, KQ vs KB (king and queen vs king and bishop) positions are (correctly) declared as unwinnable for the player holding the bishop (see Lemma 6).
- Chess24 seems to proceed as Chess.com does. As an example, see Position 3. It corresponds to a Chess24 game that, according to FIDE rules, was unfairly classified as a draw after Black ran out of time. White has only a bishop, but there is still a chance for White to checkmate Black (of course not by force) if Black promoted to a knight and trapped themself in a corner!

We do not advocate following the FIDE Laws of Chess over other choices. Every chess server should have the right to choose their own rules, especially for situations that occur infrequently and do not significantly impact the experience of chess players. However, we believe that the approach followed by Lichess is more satisfactory: *in case unwinnability cannot be determined by the system's logic, declare the position as winnable.* Otherwise, very unfair situations may arise. For example, using only the above-mentioned material rules



Position 3 Black ran out of time. White can still *helpmate* with under promotion to a knight. (Chess24 game `VO1NB3MGSYqXv3BhVsCBkA` [8].)



Position 4 White ran out of time. Black can *force a victory* with just a knight. (Lichess game `HaTT3dsU`.)

in Position 4, after White ran out of time, would lead to classifying the game as drawn. However, not only can Black helpmate in that position, but Black can also force a victory, for example, with the following forced sequence.[4]

1 ♔h8 ♘h4 2 ♔h7 ♘g2 3 ♔h8 ♘f4 4 ♔h7 ♘e6 5 ♔h8 ♘f8 6 h7 ♘g6#

An absurd situation arises: In Position 4, White is completely lost. However, White can still draw the game by letting their time run out (if they are playing on a server that adjudicates timeouts by following simple rules based on the amount of material) [9].

Existing tools for unwinnability. Labelle [12] performed a computer search over a database of 2M+ *over the board* games (from 1998 to 2011), searching for dead drawn positions by forced insufficient material or forced stalemate where players continued making moves (illegal moves, according to [7, Article 5.2.2]). This demonstrates that dead positions occur in professional chess and can be easily missed by chess arbiters.

Varmose [21] implemented an algorithm that identifies blocked positions that only involve bishops and pawns, which only misses some corner cases. This is a non-trivial step towards solving unwinnability, but the tool cannot identify all unwinnable positions.

Other tools for solving helpmate problems, such as the analyzer by Paliulionis [18] or the solver by Dugovic [6] can potentially identify any unwinnable position as they perform an exhaustive search over the tree of moves. Nevertheless, such tools can hardly be utilized to decide unwinnability, they would incur a prohibitive cost for most positions [15].

This state of affairs leaves the problem of automatically checking whether or not a position can be won by a given player inadequately addressed. Given the intractability of a simple brute force approach, we ask:

How can we rule out all sequences of legal moves without actually exploring them all?

[4] The trick is to always watch the `g6` square when the white king is on `h8`, preventing White from stalemating themself by pushing the pawn. This has to be done while maneuvering the knight to `f8` from where it controls `g6` as before as well as `h7`, thus forcing White to push the pawn.

On the complexity of chess unwinnability. Chess unwinnability for an appropriate generalization of chess over an $n \times n$ board has been studied by Brunner, Demaine, Hendrickson, and Wellman [2]. The authors prove, via a reduction from a one-player game called Subway Shuffle [10], that chess unwinnability is PSPACE-complete.

Their generalization of chess does not impose any restriction on the length of games. A natural alternative generalization, motivated by the 75-moves rule[5], would be to impose a polynomial bound on the round complexity of the game. In that case, it would follow from the results of Brunner et al. that chess unwinnability is coNP-complete under such generalization. These intractability results, however, do not apply to the (constant) case $n = 8$, our goal in this paper.

1.3 Our contributions

We pursue the study of chess unwinnability and establish several results that, together, form an algorithm which is sound, complete and computationally practical.

Static analysis. Our main contribution is a mechanism for statically determining that a position is unwinnable without explicitly exploring game variations (Section 3), which we believe, can be of independent interest and applicable to other board games. This algorithm is particularly effective on *blocked* positions, e.g. Position 1, where players have access to limited and disjoint regions of the board and can make no progress. Our static analysis is performed in two steps: (i) identifying what pieces can move and all the squares that each can potentially reach; (ii) based on the previous information and based on the number of pieces that can check and constrain the movement of the intended loser's king, our analysis may conclude that checkmate is impossible.

The problem associated to (i), that we coin the *mobility problem*, is arguably the most challenging part of the analysis. In Section 3.1, we provide an algorithm (Figure 7) that over-approximates the true solution to the mobility problem on the given position (Corollary 9). (Informally, the solution provided by our algorithm is always greater than or equal to the actual solution.) We then show (Lemma 11) that our routine for addressing our second step (ii), described in Figure 8, is sound when given as input the true solution to the mobility problem. (It is never wrong when its output is "unwinnable".) Furthermore, we argue that this routine is monotone (Lemma 10), which allows us to conclude that it is also sound when given an over-approximation to the actual mobility solution (Theorem 12). Consequently, the composition of our two routines leads to a (static) unwinnability algorithm which is **sound** (but not complete).

Search of variations. We propose an algorithm for exploring variations, enhanced with a transposition table and selected heuristics for deciding what moves to explore further (Section 2.2, Figure 5). This building block is combined with our static analysis to build an algorithm for chess unwinnability (Section 4, Figure 9) that is sound and complete.

Implementation. We implement our algorithm and evaluate it over the entire database of Lichess standard rated games [13], successfully identifying all the games that were unfairly classified after a timeout (Section 5). These results evidence that the algorithms developed in this work are suitable for practical use and could be adopted by real-world chess servers.

[5] Establishing that a game is drawn if 75 full moves are completed without captures or pawn movements.

2 Preliminaries

We assume the reader to be familiar with the basic rules of chess, including piece movement and game completion. Nevertheless, we establish a formal notation, which facilitates a rigorous description of our main algorithms and results.

Let $\mathcal{S} \coloneqq \{\mathsf{a1}, \mathsf{b1}, \ldots, \mathsf{h8}\}$ be the set of all 64 *squares* on a chessboard. Let the set of *piece types* be $\mathcal{T} \coloneqq \{\mathsf{K}, \mathsf{Q}, \mathsf{R}, \mathsf{B}, \mathsf{N}, \mathsf{P}\}$. A *position* pos is a collection of pieces, i.e., triples $(t, c, s) \in \mathcal{T} \times \{\mathsf{w}, \mathsf{b}\} \times \mathcal{S}$ where t is a piece type, c is the piece color and s is a square. Given a piece $P \coloneqq (t, c, s)$ we define $P.\mathsf{type} = t$, $P.\mathsf{side} = c$ and $P.\mathsf{sq} = s$.

A position is *valid* if for every distinct $P, P' \in \mathsf{pos}$, it holds that $P.\mathsf{sq} \neq P'.\mathsf{sq}$. A position is said to be *legal* if it can be reached from the initial position of a chess game by a sequence of legal moves. We define the *king-distance* between two squares as the number of moves that it takes for a king to go from one to the other over an empty board. We define *knight-distance* analogously. We say that two squares are *adjacent* if they are at king-distance 1. A rank is a row of 8 squares, whereas a file is a column of 8 squares. Unless specified otherwise, we measure the depth of a variation (a sequence of moves) in halfmoves or *plies*.

For every square $s \in \mathcal{S}$, we denote by $\boldsymbol{+}(s)$ the set of squares that share a border with s (that is, adjacent squares of a different color) and by $\boldsymbol{\times}(s)$ the squares that are diagonally adjacent to s (i.e., adjacent squares of the same color). We define $\boldsymbol{*}(s) \coloneqq \boldsymbol{+}(s) \cup \boldsymbol{\times}(s)$. We also denote by $\boldsymbol{\mathsf{N}}(s)$ the set of squares that are at knight-distance 1 from s. The empty or singleton set containing the adjacent square to s from which a white (respectively, black) pawn could move to s in one non-capturing move is denoted by $\boldsymbol{\mathsf{P}}(s)$ (respectively, $\boldsymbol{\mathsf{p}}(s)$). Finally, we denote by $\boldsymbol{\mathsf{P}}(s)$ (respectively, $\boldsymbol{\mathsf{p}}(s)$) the set of squares from which a white (respectively, black) pawn attacks s.

▶ **Definition 2** (Unwinnability). *We say that a position is* unwinnable *for a given player if there does not exists a sequence of legal moves that ends in a checkmate by the player.*

Our goal is to build an algorithm for solving chess unwinnability, which given a position and an intended winner, after a finite number of steps always outputs a binary value in $\{\text{Unwinnable}, \text{Winnable}\}$, indicating unwinnability. Ideally, when the position is declared as Winnable, we would like our algorithm to also provide a helpmate sequence, i.e., a witness of non-unwinnability. We require our algorithm to be sound.

▶ **Definition 3** (Soundness). *We say an algorithm for solving chess unwinnability is* sound *if it is never wrong when its output is* Unwinnable.

Another desirable property is completeness, meaning that the algorithm will be able to identify all unwinnable positions.

▶ **Definition 4** (Completeness). *We say an algorithm for solving chess unwinnability is* complete *if it is never wrong when its output is* Winnable.

Since chess is finite, it is not hard to implement a chess unwinnability algorithm that is both sound and complete, by performing an exhaustive search over the tree of variations. The challenge here, however, is to achieve efficiency while preserving soundness and completeness. Note that an exhaustive search would terminate relatively quickly in most positions, since it just needs to find a helpmate sequence for the intended winner. However, in positions where no checkmate is possible, the whole tree of variations would need to be exhausted before unwinnability could be concluded.

Our starting point will be a routine that performs an exhaustive search, enhanced with a transposition table and with heuristics for selecting what moves to explore first (see Section 2.2). We will first establish some preliminary results that ensure unwinnability in positions that contain no pawns.

Find-Helpmate$_c$(pos, depth, maxDepth): Global variables: table, cnt, nodesBound

Inputs: position, depth (int), maxDepth (int)
Output: bool (*true* if a checkmate sequence was found, *false* otherwise)

1: **if** the intended winner is checkmating their opponent in pos **then return** *true*
2: **if** the intended winner has just the king **or** the position is unwinnable according to Lemma 5 or Lemma 6 **or** the position is stalemate **or** the intended winner is receiving checkmate in the position **then return** *false*
3: increase cnt and set $d :=$ maxDepth - depth
4: **if** cnt $>$ nodesBound $\vee\ d < 0$ **then return** *false* ▷ The search limits are exceeded
5: **if** $(\mathsf{pos}, D) \in$ **table** with $D \geq d$ **then return** *false* ▷ pos was already analyzed
6: store (pos, d) in table
7: **for every** legal move m in pos **do**:
8: let inc = match Score(pos, m) with Normal $\to 0$ | Reward $\to 1$ | Punish $\to -2$
9: **if** Find-Helpmate$_c$(pos.move(m), depth $+ 1$, maxDepth $+$ inc) **then return** *true*
10: **return** *false* ▷ No mate was found after exploring every legal move

Figure 5 Find-Helpmate$_c$ routine, returns *true* if a checkmate sequence for player $c \in \{\mathsf{w}, \mathsf{b}\}$, the intended winner, is found or *false* otherwise. The base call should be done on depth $= 0$, cnt $= 0$, and an empty table. The value of maxDepth and nodesBound can be chosen to set the limits of the search. See the full version of this paper for details about the Score routine.

2.1 Preliminary results

We say a position is *pawn-free* if it does not contain pawns of any color. We refer to the full version of this paper for a formal a proof of the following lemmas.

▶ **Lemma 5.** *A pawn-free position is unwinnable for a player with just a knight if their opponent does not have knights, bishops or rooks.*

▶ **Lemma 6.** *A pawn-free position is unwinnable for a player with just bishops of one square color if their opponent does not have knights or bishops of the opposite square color.*

2.2 Search of variations

We propose a dedicated search of variations, enhanced with a transposition table and heuristics that reward some variations, which will be explored further before others (described in Figure 5). This routine, called Find-Helpmate$_c$, constitutes an important building block of our main algorithm for chess unwinnability, described in Section 4, where it is combined with our static analysis (see Section 3).

Find-Helpmate$_c$ is a recursive algorithm that outputs *true* only when it has found a checkmate position for the intended winner. Otherwise, the algorithm will output *false* based on several criteria:

- The game is over, but the intended winner did not checkmate their opponent.
- The conditions of Lemma 5 or Lemma 6 apply.
- The position was found in the transposition table (a table storing all positions that have been explored so far), so it is not necessary to repeat the search that starts from it.
- All legal moves have been explored without having found a checkmate.
- The search limits were reached.

If the final output of Find-Helpmate$_c$ on the given position is *false*, and the search limits were not reached in any of its recursive calls, the position is truly unwinnable.

The search limits include a maximum depth for the variations being explored and a limit on the total number of explored positions. Before exploring the position after a legal move, we determine with our Score heuristic (described in the full version of this paper) whether the maximum depth limit will be increased (rewarded), decreased (punished) or remain the same in the analysis of the variation associated with the move.

We refer to Section 4 for details of how Find-Helpmate$_c$ is integrated into our chess unwinnability algorithm via iterative deepening [11].

3 Static algorithm

The search of variations provided by Find-Helpmate$_c$ (Figure 5), which is enhanced with a transposition table and our heuristics for selecting what moves to explore first, will potentially terminate on any position, correctly classifying it with respect to unwinnability if the search limits were sufficiently large. However, in blocked positions, the search space can become prohibitively large. For example, in Position 1 the search would need to iterate over and store the (more than) 80K positions that can arise from that board configuration before deciding that the position is unwinnable. This would greatly exceed the maximum computation time that we should dedicate to a single position if we want our algorithm to be competitive and suitable for its integration in real-world chess servers, which usually handle tens of games terminating every second.

In order to offload unnecessary computations from our main routine in blocked positions, we design a mechanism that allows us to conclude that certain positions are unwinnable without explicitly exploring all variations. Our algorithm is divided into two phases:

(i) Identifying what pieces can move and all the squares that each can potentially reach.

(ii) Identifying the *king's region*, defined as the set of all squares that can be reached by the intended loser's king, as well as identifying all the intended winner's pieces that can move inside the king's region, the so-called *intruders*. Based on the number of intruders and their piece type, our algorithm may conclude that checkmate is impossible.

▶ Remark 7. This algorithm does not need to be complete, since it is backed by our main search routine. In fact, as evidenced by [2] or by the example in Position 2, deciding unwinnability without exploring variations may be an impossible task. Nevertheless, we require that our static algorithm be **sound** in the sense that it can be fully trusted when it classifies a position as *unwinnable*.

Performing step (i) is arguably the most challenging part of the analysis. For every $P \in \mathsf{pos}$ and every $s \in \mathcal{S}$, we need to decide whether or not piece P, currently on square $P.\mathsf{sq}$, can potentially go to square s after a sequence of legal moves. Define $M^*_{P\to s}$ as 1 if the above displacement is possible and 0 otherwise. Our *mobility algorithm* will try to approximate the correct value of $M^*_{P\to s}$ in the given position for every piece and every target square.

3.1 Mobility algorithm

We consider binary variables $M_{P\to s} \in \{0, 1\}$ for every $P \in \mathsf{pos}$ and every $s \in \mathcal{S}$, encoding the output of our mobility algorithm. Since the function associated to solving step (ii) is monotone (see Lemma 10), any over-approximation of the actual solution is acceptable for the static algorithm to be sound. Namely, we allow for solutions $M_{P\to s}$ that satisfy $M_{P\to s} \geq M^*_{P\to s}$, coined *admissible* solutions. Intuitively, this is possible because wrongly

concluding that a piece can move more than what it really can is not harmful (in the sense that it may lead to the conclusion that the position is winnable when it is actually unwinnable, but not vice versa). However, we hope for an approximation that is as close as possible to the actual solution. (Observe that a degenerate output of $M_{P\to s} = 1$ for all P and s is admissible, but not useful, because step (ii) would simply return "possibly winnable".)

We also define additional variables representing square reachability and clearance. This is useful to model pawn captures and king movements more accurately. More concretely, we consider the following binary variables: $M_{P\to s}$ for every $P \in \mathsf{pos}$ and every $s \in \mathcal{S}$; C_P for every $P \in \mathsf{pos}$; and R^c_s for every $s \in \mathcal{S}$ and every $c \in \{\mathsf{w}, \mathsf{b}\}$, defined as follows:

- $M_{P\to s}$ indicates if piece P, currently on $P.\mathsf{sq}$, can eventually *move* to square s.
- R^c_s indicates if square s can eventually be *reached* by a *non-king* piece of color c (or if it is currently occupied by such a piece).
- C_P indicates if piece P can be *cleared* from its current square (by moving or being captured).

Given a piece P and a square s, we define the P-predecessors of s, denoted by $\mathsf{pred}_P(s)$, as the squares that are at king-distance 1 from s (except for knight predecessors, which are at king-distance 2), from which a piece of type $P.\mathsf{type}$ can reach s in one *non-capture* move over an empty board. More concretely,

$$\mathsf{pred}_P(s) = \begin{cases} \blacksquare(s) & \text{if } P = (\♙, \mathsf{w}, \_) \\ \blacksquare(s) & \text{if } P = (\♙, \mathsf{b}, \_) \\ \blacksquare(s) & \text{if } P.\mathsf{type} = \♘ \\ \blacksquare(s) & \text{if } P.\mathsf{type} = \♗ \\ \blacksquare(s) & \text{if } P.\mathsf{type} = \♖ \\ \blacksquare(s) & \text{if } P.\mathsf{type} \in \{\♕, \♔\} \end{cases} \qquad \mathsf{pred\text{-}capt}_P(s) = \begin{cases} \blacksquare(s) & \text{if } P = (\♙, \mathsf{w}, \_) \\ \blacksquare(s) & \text{if } P = (\♙, \mathsf{b}, \_) \\ \mathsf{pred}_P(s) & \text{otherwise} \end{cases}$$

$$\mathsf{prom}(P) = \begin{cases} \{\mathsf{a8}, \ldots, \mathsf{h8}\} & \text{if } P = (\♙, \mathsf{w}, \_) \\ \{\mathsf{a1}, \ldots, \mathsf{h1}\} & \text{if } P = (\♙, \mathsf{b}, \_) \\ \emptyset & \text{otherwise} \end{cases}$$

We define $\mathsf{pos.attackers}(s)$ as the set $\{P \in \mathsf{pos} : P.\mathsf{sq} \in \mathsf{pred\text{-}capt}_P(s)\}$. The mobility algorithm from Figure 7 greedily activates the mobility (reachability and clearance) variables as soon as it is possible (i.e. not forbidden by the logic of the implications from Figure 6). For example, if P is a knight and there is some piece P' of the same color as P, currently on $\mathsf{h8}$, $M_{P\to\mathsf{h8}}$ can be set to true as soon as ($M_{P\to\mathsf{g6}} = 1$ or $M_{P\to\mathsf{f7}} = 1$) and $C_{P'} = 1$. But it cannot be set to true otherwise: for the knight to reach $\mathsf{h8}$, it must first reach a direct predecessor of $\mathsf{h8}$ and the ally piece on $\mathsf{h8}$ must first be cleared.

▶ Lemma 8 (Mobility Soundness). *Let pos be a position where no player has* castling rights *and* en passant *is not possible. Let* $M \leftarrow$ *Mobility*(*pos*) *(see Figure 7). If* $M_{P\to s} = 0$ *for some piece* $P \in$ *pos and some square* s*, then there is no sequence of legal moves after which piece* P*, starting from pos, can reach square* s*.*

Proof. The result follows from the fact that all implications from Figure 6 are sound, in the sense that they are all satisfied by the true solution to the mobility problem.

We will prove a more general result, involving the extra variables for reachability and clearance. Let $(\{M_{P\to s}\}_{P,s}, \{C_P\}_P, \{R^c_s\}_{s,c})$, for $P \in \mathsf{pos}$, $s \in \mathcal{S}$, $c \in \{\mathsf{w}, \mathsf{b}\}$, be the final state of the execution of Mobility(pos). We will argue that, for all $n \in \mathbb{N}$:

(a) If there exists a sequence of n legal halfmoves (plies) after which piece $P \in \mathsf{pos}$ ends at square s, then $M_{P\to s} = 1$.

(b) If there exists a sequence of n legal halfmoves after which a square s is occupied by a non-king piece of color c, then $R^c_s = 1$.

(c) If there exists a sequence of n legal halfmoves after which a piece $P \in \mathsf{pos}$ is cleared from its current square $P.\mathsf{sq}$, then $C_P = 1$.

Move. If a piece can move to square s, it must pass first by a predecessor of s:

$$\forall P \in \mathsf{pos} : P.\mathsf{type} \neq \♙, \quad \forall s \in \mathcal{S} \setminus \{P.\mathsf{sq}\}. \qquad M_{P\to s} \implies \bigvee_{u \in \mathsf{pred}_P(s)} M_{P\to u}$$

Pawn move. For pawn captures we require that the capturing square can be reached by a (non-king) opponent piece. Pawns that promote may go everywhere:

$$\forall P \in \mathsf{pos} : P.\mathsf{type} = \♙, \quad \forall s \in \mathcal{S} \setminus \{P.\mathsf{sq}\}. \qquad M_{P\to s} \implies \bigvee_{u \in \mathsf{pred}_P(s) \cup \mathsf{prom}(P)} M_{P\to u} \vee \bigvee_{u \in \mathsf{pred\text{-}capt}_P(s)} \left(M_{P\to u} \wedge R_s^{\neg P.\mathsf{side}}\right)$$

Clearance. A piece can be cleared from its square by moving or being captured:

$$\forall P \in \mathsf{pos}. \qquad C_P \implies \bigvee_{s \in \mathcal{S}\setminus\{P.\mathsf{sq}\}} M_{P\to s} \quad \vee \bigvee_{\substack{P' \in \mathsf{pos} \\ P'.\mathsf{side} \neq P.\mathsf{side}}} M_{P'\to P.\mathsf{sq}}$$

Reachability. A square can be reached if some piece can move to it or it is occupied:

$$\forall s \in \mathcal{S}, \quad \forall c \in \{\mathsf{w}, \mathsf{b}\}. \qquad R_s^c \implies \bigvee_{\substack{P \in \mathsf{pos} \\ P.\mathsf{side}=c \,\wedge\, P.\mathsf{type}\neq \♔}} M_{P\to s}$$

King attackers. Direct opponent attackers must be cleared before a king can move:

$$\forall P \in \mathsf{pos} : P.\mathsf{type} = \♔, \quad \forall s \in \mathcal{S} \setminus \{P.\mathsf{sq}\}. \qquad M_{P\to s} \implies \bigwedge_{\substack{P' \in \mathsf{pos.attackers}(s) \\ P'.\mathsf{side} \neq P.\mathsf{side}}} C_{P'}$$

Not self-capture. A piece must be cleared from a square before other of the same color can move to it:

$$\forall P, P' \in \mathsf{pos}, P \neq P' \text{ with } P.\mathsf{side} = P'.\mathsf{side}. \qquad M_{P\to P'.\mathsf{sq}} \implies C_{P'}$$

Figure 6 Logical implications for the static algorithm.

We proceed by induction on n. Assume $n = 0$, it is easy to see that (a) will hold, because the Mobility algorithm sets $M_{P\to P.\mathsf{sq}}$ to 1 in step (2), for every piece P. (Note that the algorithm never changes the value of a variable to 0 after it has been set to 1.) To see that (b) holds, note that the *reachability* rule from Figure 6 will set $R_{P.\mathsf{sq}}^{P.\mathsf{side}}$ to 1 for all $P \in \mathsf{pos}$, but those are exactly the squares that can be reached in no halfmoves by pieces of the corresponding color. Finally, (c) holds trivially, because there is no sequence of 0 halfmoves that allows a piece to be cleared from their current square.

Now, assume that the result is true for sequences of halfmoves of length n. We will argue that it must also be true for sequences of $n+1$ halfmoves. We start with (a). Consider a piece $P \in \mathsf{pos}$ and a square s such that $M_{P\to s} = 0$. We distinguish four cases depending on the piece type of P:

- If $P.\mathsf{type} = \♘$, then there are at most two rules in Figure 6 of the form $M_{P\to s} \implies f$, the *move* rule, with $f_1 = \bigvee_{u \in \mathsf{pred}_P(s)} M_{P\to u}$; and possibly the *not self-capture* rule, with $f_2 = C_{P'}$ (if there was a piece P' of the same color as P originally at s). Because variable

Mobility(pos):

Inputs: a position
Output: mobility solution $\{M_{P\to s}\}_{P\in\mathsf{pos},s\in\mathcal{S}}$

1: set $M_{P\to s} := 0$, $C_P := 0$, $R^c_s := 0$ for all $P \in \mathsf{pos}$, $s \in \mathcal{S}$ and $c \in \{\mathsf{w},\mathsf{b}\}$ and let $\vec{X}$ be the state containing all these variables
2: set $M_{P\to P.\mathsf{sq}} := 1$, for all $P \in \mathsf{pos}$ ▷ Every piece can "move" to its current square
3: **for every** variable V in $\vec{X}$ that is still set to 0 **do**
4: **if** for every rule from Figure 6 of the form "$V \Rightarrow f$", formula f evaluates to *true* on the current state of variables $\vec{X}$ **then** set V to 1 in $\vec{X}$
5: repeat steps 3 and 4 until no new variables are set to 1
6: **return** $\{M_{P\to s}\}_{P\in\mathsf{pos},s\in\mathcal{S}}$

Figure 7 Mobility algorithm.

$M_{P\to s}$ was not set to 1 in any iteration of steps 3-4 from Figure 7, clause f_1 (or clause f_2 when applicable) must evaluate to false on the final state. Applying the induction hypothesis, this means that in n halfmoves P did not have time to reach any of the P-predecessors of s, or that there was a piece P' of the same color as P initially on square s that did not have time to be cleared from s. Therefore, it is impossible for piece P to reach s in one more halfmove, as desired.

- If $P.\mathsf{type} \in \{♗,♖,♕\}$, the sliding pieces, we will argue that $M_{P\to s} = 0$ implies that for every sliding direction starting at s in which P can potentially move, it must hold that every square t in the direction (counting from s) satisfies $M_{P\to t} = 0$, until there is (possibly) a square t^* such that $C_{P'} = 0$ with $P'.\mathsf{sq} = t^*$ and $P'.\mathsf{side} = P.\mathsf{side}$ for some $P' \neq P$. If we can show that, the induction hypothesis gives us that piece P cannot reach in n halfmoves any square in the relevant sliding directions of s, unless there is a piece of the same color as P between the square and s, which cannot be cleared in n halfmoves. In that case we can safely conclude that P cannot reach s in n+1 halfmoves, as desired.
  The above claim can be proved by induction on the sliding direction. Let t be the predecessor of s in a certain direction. As before, if $M_{P\to t}$ was not set to 1 in any iteration of steps 3-4 from Figure 7, we know that either $M_{P\to u} = 0$ for all P-predecessors of t (and in particular for the next square in the direction!) or $C_{P'} = 0$ for some piece of the same color as P with $P'.\mathsf{sq} = t$. If the first case is true, we can continue the induction on the direction from the next square. If the second case is true, we can stop the induction since we have already found an ally blocker in the direction, as desired.
- If $P.\mathsf{type} = ♔$, we can proceed as before but this time there is one extra rule that comes into play, the *king attackers* rule. Again, by applying the induction hypothesis, we can conclude that in n halfmoves it was impossible for P to reach a P-predecessor of s, or that there was an ally piece on s that did not have time to be cleared, or that there was at least an enemy directly attacking s that did not have time to be cleared.[6] Consequently, we conclude that it is impossible for king P to reach s in one extra move.
- Finally, if $P.\mathsf{type} = ♙$ a similar reasoning, now involving rules *pawn move* and *not self-capture*, applies. By the induction hypothesis we can conclude that in just n halfmoves, either (i) it was impossible for P to reach a P-predecessor of s[7] and it was impossible for

[6] Direct enemies cannot be blocked, so the only way they stop attacking s is by moving or being captured.
[7] For double pawn pushes we need to argue as for sliding pieces over the jumped square, but a similar technique applies.

pawn P to reach a promoting square and it was impossible for P to reach a P-capture-predecessor of s while at the same time having a (non-king) enemy piece reaching s; or (ii) it was impossible for an ally initially at s to be cleared from s in n halfmoves. This makes it impossible for pawn P to reach s in one extra move.

To see (b), let s be an arbitrary square, let $c \in \{\mathsf{w}, \mathsf{b}\}$ and assume that $R_s^c = 0$. Note that variable R_s^c only appears in one rule from Figure 7, the *reachability* rule. Since R_s^c has not been activated, we conclude that $M_{P \to s} = 0$ for all $P \in \mathsf{pos}$ such that $P.\mathsf{side} = c$ and $P.\mathsf{type} \neq \mathbb{K}$. As we have shown above, this means none of the non-king pieces of color c could have reached s in a sequence of $n{+}1$ halfmoves, as desired.

Finally, to see (c), let P be any piece in the position and assume that $C_P = 0$. Note that C_P only appears in the *clearance* rule. Because C_P has not been activated, we can conclude that $M_{P \to s} = 0$ for all $s \in \mathcal{S} \setminus \{P.\mathsf{sq}\}$ and that $M_{P' \to P.\mathsf{sq}} = 0$ for all $P' \in \mathsf{pos}$ such that $P'.\mathsf{side} \neq P.\mathsf{side}$. As we have shown above, this means that there is no sequence of $n{+}1$ legal halfmoves after which piece P could have left square $P.\mathsf{sq}$ and it is also impossible (in $n{+}1$ halfmoves) that any enemy piece could have reached $P.\mathsf{sq}$, capturing P. We can conclude that piece P must still be at its initial square in pos after $n{+}1$ halfmoves as desired. ◀

The following is an immediate consequence of Lemma 8. Observe that, whenever $M_{P \to s} = 0$ for some $P \in \mathsf{pos}$, $s \in \mathcal{S}$, the lemma guarantees that there is no sequence of legal moves that allows P to reach s, so the true solution to the mobility problem will also satisfy $M^*_{P \to s} = 0$.

▶ Corollary 9. *Let* pos *be a position where no player has* castling rights *and* en passant *is not possible. Let M^* be the true solution to the mobility problem of* pos *and let $M \leftarrow$ Mobility(pos).*

$$M^*_{P \to s} \leq M_{P \to s} \quad \forall P \in \mathit{pos}, s \in \mathcal{S} \ .$$

3.2 Declaring unwinnability from a mobility problem solution

The second step of our static algorithm is described in Figure 8. It is based on the idea that a position is unwinnable if there is no good candidate mating square. Namely, if for every square in the board, (i) either the square cannot be reached by the intended loser's king, or (ii) the square cannot be attacked by the intended winner, or (iii) its adjacent squares (escaping squares for the intended loser's king) cannot all be blocked by defender pieces or covered/attacked by the intended winner pieces at the same time.

▶ Lemma 10. *The function induced by algorithm* Unwinnable^{SS} *from Figure 8 is monotone in the following sense. For every* pos*, $c \in \{\mathsf{w}, \mathsf{b}\}$ and any two mobility solutions M, M':*

$$\forall P \in \mathit{pos}, s \in \mathcal{S}.\ M_{P \to s} \leq M'_{P \to s} \implies \mathit{Unwinnable}^{SS}(\mathit{pos}, c, M) \geq^{8} \mathit{Unwinnable}^{SS}(\mathit{pos}, c, M') \,.$$

We refer to the full version of this paper for a formal proof.

▶ Lemma 11. *Let* pos *be a position and let $c \in \{\mathsf{w}, \mathsf{b}\}$. Let M^* be the true solution to the mobility problem on* pos*. If* Unwinnable$^{SS}(\mathit{pos}, c, M^*)$ *returns* true*, the position is indeed unwinnable for player c.*

[8] By convention, *true* > *false*.

$\mathsf{Unwinnable}^{\mathsf{SS}}(\mathsf{pos}, c, \{M_{P\to s}\}_{p\in\mathsf{pos}, s\in\mathcal{S}})$:

Inputs: position, intended winner, solution to the mobility problem
Output: bool (*true* if position is declared unwinnable, *false* otherwise)

1: **if** *en passant* is possible **or** a player has *castling rights* in pos **then return** *false*
2: for every piece $P \in \mathsf{pos}$, define $\mathsf{region}(P) \coloneqq \{s \in \mathcal{S} \mid M_{P\to s} = 1\}$
3: let K_c (resp. $K_{\neg c}$) be the intended winner's king (resp. intended loser's king)
4: set $\mathsf{intruders} \coloneqq \{P \in \mathsf{pos} \mid P.\mathsf{side} = c \wedge \mathsf{region}(P) \cap \mathsf{region}(K_{\neg c}) \neq \emptyset\}$
5: **if** $\exists P \in \mathsf{intruders}$ with $P.\mathsf{type} \neq$ ♗ **then return** *false*
6: **if** $\exists P, P' \in \mathsf{intruders}$ with $\mathsf{color}(P.\mathsf{sq}) \neq \mathsf{color}(P'.\mathsf{sq})$ **then return** *false*
7: for $P \in \mathsf{pos}$, define $\mathsf{att\text{-}region}(P) \coloneqq \{s \in \mathcal{S} \mid \mathsf{pred\text{-}capt}_P(s) \cap \mathsf{region}(P) \neq \emptyset\}$
8: for $s \in \mathcal{S}$, let $\mathsf{blockers}(s) \coloneqq \{P \in \mathsf{pos} \mid P.\mathsf{side} \neq c \wedge \mathsf{region}(P) \cap$ ✜$(s) \neq \emptyset\}$[a]
9: for $s \in \mathcal{S}$, define $\mathsf{assistants}(s) \coloneqq \{P \in \mathsf{pos} \mid P.\mathsf{side} = c \wedge \mathsf{att\text{-}region}(P) \cap$ ✜$(s) \neq \emptyset\}$
10: **if** $\exists s \in \mathsf{region}(K_{\neg c})$ such that $|\mathsf{blockers}(s)| + |\mathsf{assistants}(s)| \geq |$✜$(s)|$ **and** $\exists P \in \mathsf{pos}$ satisfying $s \in \mathsf{att\text{-}region}(P) \wedge P.\mathsf{side} = c$ **then return** *false*
11: **return** *true* ▷ The position must be unwinnable

[a] We could design a more complete check that looks at all neighbours of s, but the condition on step 10 would be significantly more involved (to ensure monotonicity).

Figure 8 Statically unwinnable algorithm, which may conclude that a position is unwinnable for an intended winner based on an admissible solution to the mobility problem.

Proof. Since $\mathsf{Unwinnable}^{\mathsf{SS}}$ did not return any value in step 5 or step 6, the set of pieces that can check the intended loser's king is empty or formed entirely by same-colored bishops. Furthermore, every square s in the board is such that: (i) the intended loser's king cannot reach it, or (ii) the square cannot be attacked by the intended winner, or (iii) its adjacent squares cannot not all be blocked by defenders or covered by attackers at the same time. Therefore, the intended loser will always have at least a legal move when they are in check. ◀

Since the mobility algorithm (Figure 7) always provides admissible solutions, since the static check (Figure 8) is sound on the true solution of the mobility problem, and because the static check is a (decreasing) monotone function, the composition of the mobility algorithm with the static check constitutes an algorithm for chess unwinnability that is **sound**.

▶ Theorem 12. *Let pos be a position and let* $c \in \{w, b\}$. *If* $\mathsf{Unwinnable}^{\mathsf{SS}}(\mathsf{pos}, c, \mathsf{Mobility}(\mathsf{pos}))$ *outputs* true, *then the position is unwinnable for player* c.

Proof. Let $M \leftarrow \mathsf{Mobility}(\mathsf{pos})$ and let M^* be the true solution to the mobility problem on pos. If $\mathsf{Unwinnable}^{\mathsf{SS}}(\mathsf{pos}, c, M) = \mathit{true}$, then *en passant* is not possible and no player has *castling rights* in pos (see step 1 of Figure 8), so we can apply Corollary 9 and conclude that $M^* \leq M$. We can now apply Lemma 10 and get:

$$\mathit{true} = \mathsf{Unwinnable}^{\mathsf{SS}}(\mathsf{pos}, c, M) \leq \mathsf{Unwinnable}^{\mathsf{SS}}(\mathsf{pos}, c, M^*) \ .$$

Hence we must have $\mathsf{Unwinnable}^{\mathsf{SS}}(\mathsf{pos}, c, M^*) = \mathit{true}$. By virtue of Lemma 11, pos must be unwinnable for player c, as desired. ◀

$\mathsf{Unwinnable}^{\mathsf{full}}(\mathsf{pos}, c)$:

Inputs: position, intended winner
Output: Unwinnable or Winnable (definite solution to the chess unwinnability problem)

1: **if** *true* $\leftarrow \mathsf{Unwinnable}^{\mathsf{SS}}(\mathsf{pos}, c, \mathsf{Mobility}(\mathsf{pos}))$ **then return** Unwinnable
2: **for every** $d \in \mathbb{N}$ **do** ▷ Iterative deepening
3: set $b_d \leftarrow \mathsf{Find\text{-}Helpmate}_c(\mathsf{pos}, 0, \mathsf{maxDepth} = d)$ (global $\mathsf{nodesBound} = \mathsf{bound}(d)$)
4: **if** $b_d = \mathit{true}$ **then return** Winnable
5: **else if** the search was not interrupted (in step 4 of Figure 5) **then**
6: **return** Unwinnable

Figure 9 Main routine for deciding chess unwinnability. It is based on our static algorithm (Figure 8) and our search routine (Figure 5) integrated via iterative deepening. Function bound must be increasing on d for the algorithm to be complete. The transposition table used by $\mathsf{Find\text{-}Helpmate}_c$ should be initialized to empty at the beginning, but it can be shared between different calls to $\mathsf{Find\text{-}Helpmate}_c$ in step 3. On the other hand, the global counter cnt should be initialized to 0 on every base call to $\mathsf{Find\text{-}Helpmate}_c$ in step 3.

4 Unwinnability algorithms

We present our main routine for solving chess unwinnability in Figure 9. Our algorithm consists of a search of variations (Figure 5), preceded by a static analysis (Figure 8) on the given position. Such analysis will prevent the search routine from exploring large trees of variations exhaustively, whenever it concludes that the given position is unwinnable via our alternative and much lighter mechanism described in Section 3. Our main routine achieves:

- *Soundness*: Given that it combines a search of variations with our static algorithm for identifying blocked positions, it is sound by virtue of Theorem 12.
- *Completeness*: This is due to the fact that the search over variations is exhaustive for a sufficiently large maxDepth limit, and this in turn will be eventually reached during the iterative deepening loop (step 2 of Figure 9).
- *Efficiency*: As evidenced by the experimental results from Section 5, our algorithm is practical. We identified all the unfairly classified games from the Lichess Database [13].

4.1 An alternative quicker version of our algorithm

We propose a significantly more efficient chess unwinnability algorithm, inspired by the fact that most (if not all) unwinnable positions can be classified in the following two categories:

- *Imminently terminating* positions, where the tree of variations is very small. This is usually due to the existence of forced lines, which never end on a checkmate by the intended winner (all variations end in either stalemate, checkmate by the intended loser, or insufficient mating material for the intended winner.) An example is Position 15.
- *Blocked* positions, where players can maneuver over limited and disjoint regions of the board, what prevents their interaction (and thus any possible checkmate). (See Position 1.)

The quick version of our algorithm is described in Figure 10. It performs a depth-first search over the tree of variations and stops if a certain (small) depth D is reached, with the hope that it will be sufficient to exhaustively explore the tree of variations of imminently

$\mathsf{Unwinnable}^{\mathsf{quick}}(\mathsf{pos}, c)$:

Inputs: position, intended winner
Output: Unwinnable, Winnable, or PossiblyWinnable

1: advance the position as long as there is only one legal move
2: perform a depth-first search over the tree of variations of pos and interrupt the search if (i) checkmate is found for player c or (ii) depth D is reached
3: **if** checkmate was found on the previous search **then return** Winnable
4: **else if** the search was not interrupted **then return** Unwinnable
5: **else if** the position only contains pieces of type ♙, ♗, ♔ **and** there are no *semi-open files* in the position **then**
6: **if** *true* $\leftarrow \mathsf{Unwinnable}^{\mathsf{SS}}(\mathsf{pos}, c, \mathsf{Mobility}(\mathsf{pos}))$ **then return** Unwinnable
7: **return** PossiblyWinnable ▷ Unwinnability could not be determined

Figure 10 Quick routine for analyzing unwinnability.

terminating positions. Note that this search will be almost instantaneous in most positions, because it is interrupted as soon as depth D can be reached.

After that, and if the previous search did not conclude unwinnability, our quick algorithm simply performs a call to our static routine (Figure 8). But this is done only if the position is such that there exists no *semi-open files* (files with pawns of only one color) and the only existing pieces are kings, pawns and/or bishops. This heuristic is supported by the fact that positions that do not satisfy these properties will very likely be non-blocked.

Our quick algorithm is extremely light, requiring only a few microseconds on average per position. It is also sound, but not complete. However, as we detail in Section 5, with an (empirically chosen) depth bound of $D = 9$, all unfairly classified games from the Lichess Database except three were correctly identified by $\mathsf{Unwinnable}^{\mathsf{quick}}$.

5 Experimental results

We have implemented all the algorithms described in this work and evaluated their performance in real-world games from the Lichess Database [13]. Our source code is written in C++ and leverages the code of the open-source chess engine Stockfish [19] for move generation and chess-related functions. Our implementation is publicly available as open-source and can be found on this link: `https://github.com/miguel-ambrona/D3-Chess`.

The Lichess Database of standard rated games includes 3,099,534,127 games up to date. We have applied our algorithm from Figure 9 to the final position of all games that ended in a victory by timeout. In total, 981,467,875 games (31% of all games) were analyzed in about 88 hours of CPU time (323 μs per position on average). All experiments were performed on a 3.5GHz Intel-Core i9-9900X CPU with 32GB of RAM, running Ubuntu 20.04 LTS.

Our analysis led to identifying a total of 84,100 games that were unfairly classified. Namely, games that were lost by the player who ran out of time, but their opponent could not have checkmated them by any possible sequence of legal moves. We refer to Appendix A for some remarkable positions of unfairly classified games; the remaining can be found on the following link (where our tool can also be tried interactively without installation): `https://chasolver.org`.

Table 1 Performance of the Full and Quick versions of our algorithm when applied to all games from January 2022 that ended in a victory by timeout. (A total of 32,599,280 games.)

Full Algorithm (Figure 9)	vs	Quick Algorithm (Figure 10)
2700	average # positions per second	200,000+
370 μs	average time per position	4.96 μs
1270 μs	standard deviation	9.06 μs
141 ms	maximum time per position	586 μs
2462 (100%)	unwinnable positions identified	2462 (100%)
3 h 21 min	total execution time	2 min 42 s

Our analysis identified *all* the unfairly classified games in the database. In all other games, the tool provided a checkmate sequence for the player who did not run out of time.

5.1 Comparison between the full and quick routines

Here we perform a comparison between our full algorithm, described in Figure 9 (we use $\mathsf{bound}(d) = 10,000$)[9], and our quicker version, described in Figure 10 (we use $D = 9$).

The latter is designed to be significantly faster, but it is not complete. Note that the quick version may terminate without having found a help mating sequence, declaring the position as "probably winnable". Consequently, the quick version may fail to find all unwinnable positions. In fact, out of the exactly 84,100 games that were unfairly classified (identified with the full version of our tool), the quick version can identify 84,097 of them, missing only 3 positions in the entire database. These three positions are: `FKr42ZRT` (Position 12), `bKHPqNEw` (Position 13) and `f6c1lu7R` (Position 14).

In Table 1, we present a comparison of the performance of the two versions of our tool when analyzing all Lichess games from January 2022 that ended in a victory after a timeout. We also present in Figure 11 the execution times of this analysis (for the quick algorithm).

5.2 Conclusions and future work

We believe that our algorithms are suitable for practical use and in particular chess servers (and chess software) can leverage them to accurately classify games after a timeout, following Article 6.9 of the FIDE Laws of Chess [7]. Furthermore, given the results of Figure 11, chess servers could also apply our tools after every single move during games, to terminate games as soon as a dead position is achieved, correctly applying [7, Article 5.2.2].

Although our tool successfully solved all positions from the Lichess Database, we note that there exist artificial positions that are not efficiently captured by our logic and the tool could take a long time to analyze. Indeed, Position 12 with several additional black dark-squared bishops is an example.

A very interesting direction for future work would be to equip our mobility algorithm with extra rules (see Figure 6), that increase its scope. That way, the quick algorithm could potentially identify all unfairly classified games from the Lichess Database. Out of the three positions that our quick version cannot currently handle, Position 13 could be solved by setting $D = 14$ (instead of $D = 9$ as in the experiments above; see Figure 10), but that would

[9] Note that bound should technically be an increasing function on d for the algorithm to be complete. In practice this is not necessary and a constant amount of 10K nodes per iteration seems empirically good.

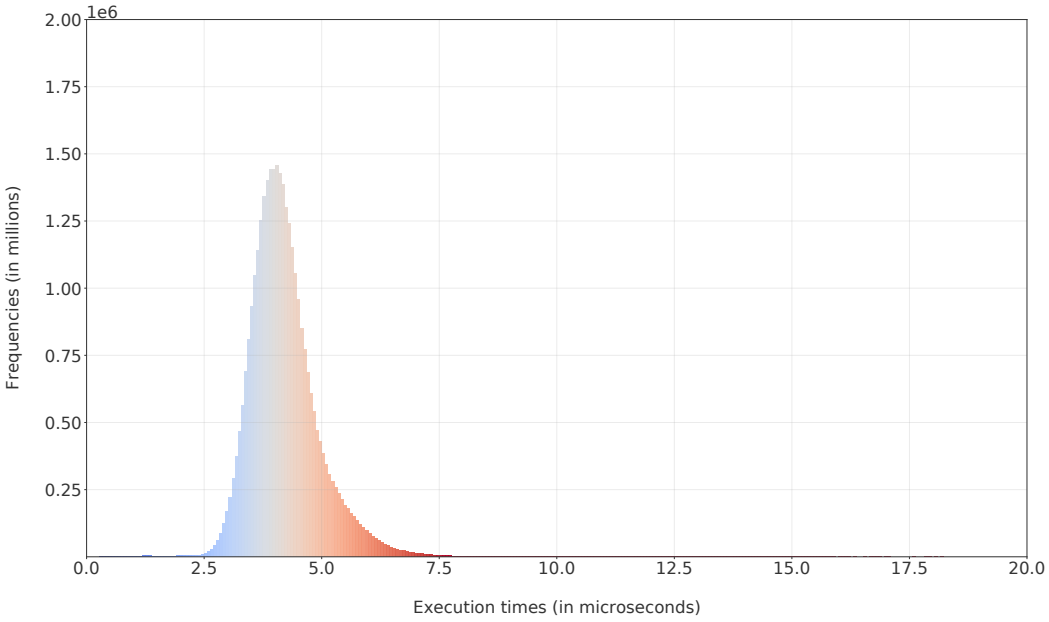


Figure 11 Quick analysis of 32,599,280 Lichess positions from January 2022.

significantly affect its performance. Positions 12 and 14, which look surprisingly similar, are harder to address. Our mobility algorithm would need to "understand" that although the pawn on g2 (of both positions) can be captured, that would leave White with no legal moves.

Finally, it would be very interesting to explore whether the ideas presented in this paper can be applied to other games that require similar analyses.

References

1 Lichess Blog. Announcing instant chess!, 2017. URL: https://lichess.org/blog/WN7V-jAAAAdH8ITR/announcing-instant-chess.

2 Josh Brunner, Erik D. Demaine, Dylan Hendrickson, and Julian Wellman. Complexity of retrograde and helpmate chess problems: Even cooperative chess is hard, 2020. arXiv:2010.09271.

3 Andrew Buchanan. Dead reckoning, 2001. URL: http://anselan.com/tutorial.html.

4 Andrew Buchanan. Dead reckoning: Castling & en passant. *StrateGems. U.S. Chess Problem Magazine*, 2001. URL: http://anselan.com/DRSGtext.html.

5 Chess.com. Game 13251713497. URL: https://www.chess.com/game/live/13251713497.

6 Daniel Dugovic. Helpmate solver, 2020. URL: https://github.com/ddugovic/Stockfish/blob/master/src/types.h#L159.

7 International Chess Federation. FIDE Laws of Chess Handbook, 2018. URL: https://handbook.fide.com/chapter/E012018.

8 Chess24. Community Feedback. Time ran out but insufficient material to mate - sufficient material, actually!, 2020. URL: https://chess24.com/en/community/feedback/time-ran-out-but-insufficient-material-to-mate---sufficient-material--actually.

9 Chess.com. Forums. Is the best move to let your time run out?, 2021. URL: https://www.chess.com/forum/view/endgames/is-the-best-move-to-let-your-time-run-out.

10 Robert A. Hearn. Games, puzzles, and computation. PhD thesis, Massachusetts Institute of Technology, 2006.

11 Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985. doi:10.1016/0004-3702(85)90084-0.

12 François Labelle. Illegal moves by grandmasters, 2011. URL: https://wismuth.com/chess/illegal-moves.html.

13 Lichess. Open database, 2022. URL: https://database.lichess.org/.

14 J.I. Minchin, J. Zukertort, and W. Steinitz. *London International Chess Tournament 1883*. ISHI Press, 2012. URL: https://books.google.es/books?id=QEqyNAEACAAJ.

15 Ornicar/lila. Issue number 6804. Detect all positions without a legal sequence of moves to checkmate, 2020. URL: https://github.com/ornicar/lila/issues/6804#issuecomment-724002709.

16 Chess.com. The Rules of Chess. My opponent ran out of time. Why was it a draw? URL: https://support.chess.com/article/268-my-opponent-ran-out-of-time-why-was-it-a-draw.

17 Chess.com. The Rules of Chess. What does 'insufficient mating material' mean? URL: https://support.chess.com/article/128-what-does-insufficient-mating-material-mean.

18 Viktoras Paliulionis. Helpmate analyzer. URL: http://helpman.komtera.lt.

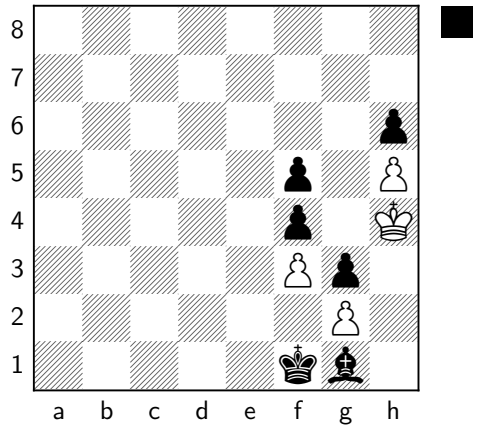
19 Stockfish. Open source chess engine, 2022. URL: https://stockfishchess.org/.

20 Ronald Turnbull. Dead reckoning: a new discovery in problem chess. *The Problemist*, pages 140–141, July 2001. URL: http://anselan.com/DRtPturnbull.html.

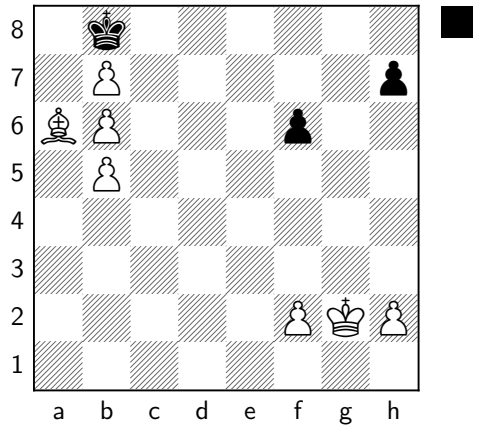
21 Jakob Varmose. Deadposition2. URL: https://github.com/jakobvarmose/deadposition2.

22 Wikipedia. Helpmate, 2021. URL: https://en.wikipedia.org/wiki/Helpmate.

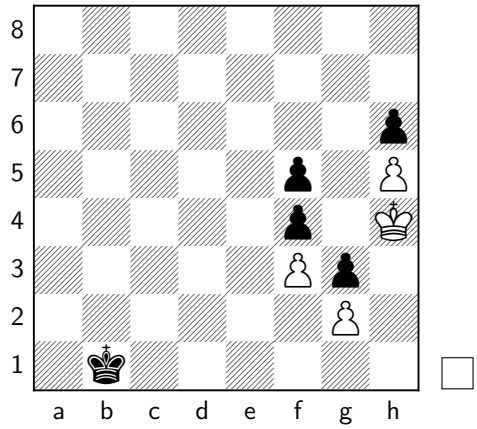
A Positions from unfairly classified Lichess games



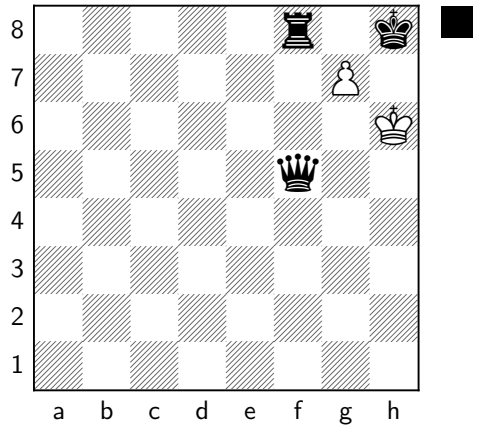
Position 12 Lichess game `FKr42ZRT`.



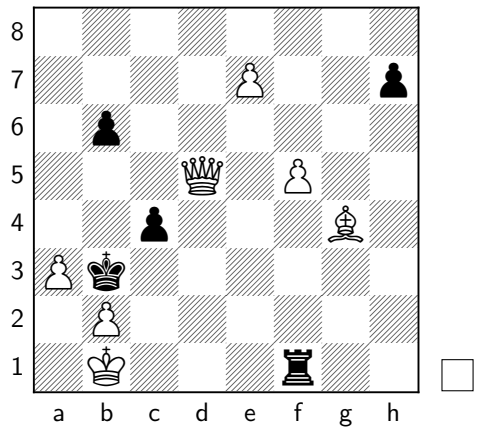
Position 13 Lichess game `bKHPqNEw`.



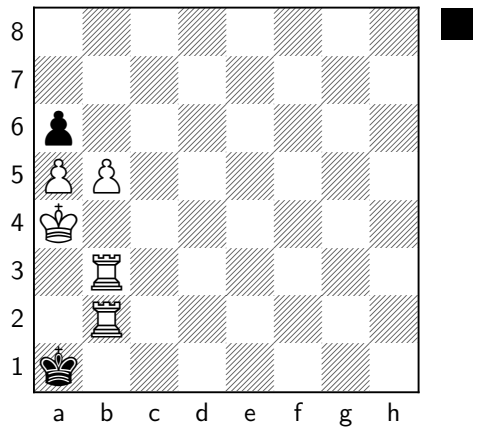
Position 14 Lichess game `f6c1lu7R`.



Position 15 Lichess game `OawUhnkq`.

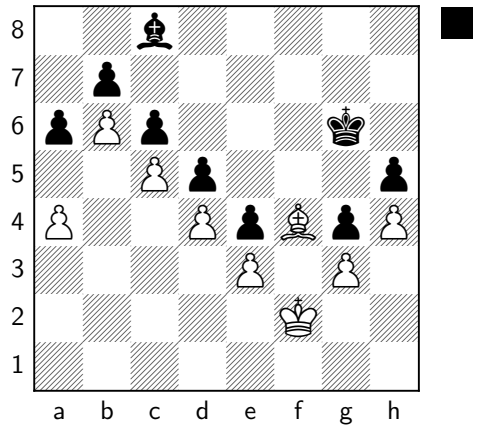


Position 16 Lichess game `ZNBhS4pz`.

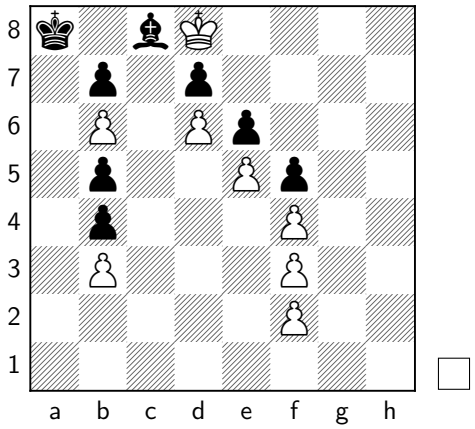


Position 17 Lichess game `3y8e8sCm`.

B Puzzles: Are the following positions dead?



Position 18 *Is this a dead position?*
Lichess game `QRvIMh3z`.

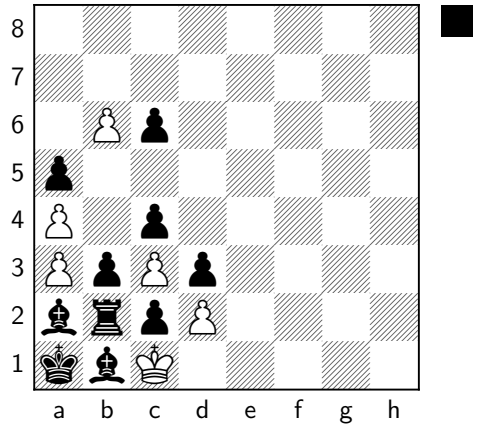


Position 19 *Is this a dead position?*
Miguel Ambrona (Spain).

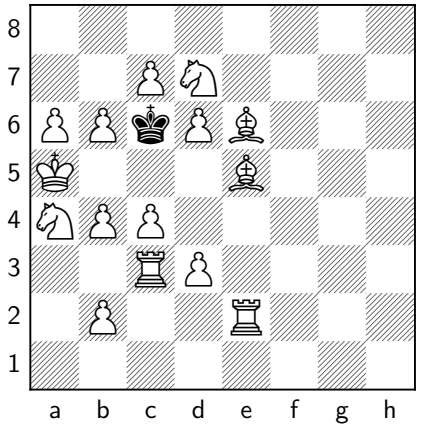
C Original compositions based on dead reckoning

We present two original compositions, by Andrew Buchanan and Andrey Frolkin, that the authors kindly offered to be included in this article. (Solutions available in the full version.)

In both problems the objective is to determine what the last move was. Solving them requires a clever retrograde analysis based on the fact that, by virtue of the FIDE Laws of Chess [7, Article 5.2.2], a game is finished as soon as a dead position is reached and no more moves are permitted. This genre of chess compositions is known as *dead reckoning* and can lead to unique motifs that cannot be enforced otherwise.



Position 20 *It is Black's turn. Last move?*
A. Buchanan (Singapore).
Original.



Position 21 *Whose turn is it? Last move?*
A. Frolkin (Ukraine) & A. Buchanan (Singapore). Original.