

# Elo World, a framework for benchmarking weak chess engines

DR. TOM MURPHY VII PH.D.

CCS Concepts: • **Evaluation methodologies** → **Tournaments**; • **Chess** → **Being bad at it**;

Additional Key Words and Phrases: pawn, horse, bishop, castle, queen, king

## ACH Reference Format:

Dr. Tom Murphy VII Ph.D.. 2019. Elo World, a framework for benchmarking weak chess engines. 1, 1 (March 2019), 13 pages. <https://doi.org/10.1145/nnnnnnnn.nnnnnnn>

## 1 INTRODUCTION

Fiddly bits aside, it is a solved problem to maintain a numeric skill rating of players for some game (for example chess, but also sports, e-sports, probably also z-sports if that's a thing). Though it has some competition (suggesting the need for a meta-rating system to compare them), the Elo Rating System [2] is a simple and effective way to do it. This paper is concerned with Elo in chess, its original purpose.

The gist of this system is to track a single score for each player. The scores are defined such that the expected outcomes of games can be computed from the scores (for example, a player with a rating of 2400 should win 9/10 of her games against a player

with a rating of 2000). If the true outcome (of e.g. a tournament) doesn't match the expected outcome, then both player's scores are adjusted towards values that would have produced the expected result. Over time, scores thus become a more accurate reflection of players' skill, while also allowing for players to change skill level. This system is carefully described elsewhere, so we can just leave it at that.

The players need not be human, and in fact this can facilitate running many games and thereby getting arbitrarily accurate ratings.

The problem this paper addresses is that basically all chess tournaments (whether with humans or computers or both) are between players who know how to play chess, are interested in winning their games, and have some reasonable level of skill. This makes it hard to give a rating to weak players: They just lose every single game and so tend towards a rating of  $-\infty$ .<sup>1</sup> Even if other comparatively weak players existed to participate in the tournament and occasionally lose to the player under study, it may still be difficult to understand how this cohort performs in any absolute sense. (In contrast we have "the highest ever human rating was 2882," and "there are 5,323 players with ratings in 2200–2299" and "Players whose ratings drop below 1000 are listed on the next list as 'delisted.'" [1]) It may also be the case that all weak players always lose to all strong players, making it unclear just how wide the performance gap between the two sets is. The goal of this paper is to expand the dynamic range of chess ratings to span all the way from extremely weak players to extremely strong ones, while providing canonical reference points along the way.

---

Copyright © 2019 the Regents of the Wikipia Foundation. Appears in SIGBOVIK 19119 with the title inflation of the Association for Computational Heresy; *IEEEEEE!* press, Verlag-Verlag volume no. 0x40-2A. 1600 rating points.

Author's address: Dr. Tom Murphy VII Ph.D., tom7@tom7.org.

---

2019. Manuscript submitted to ACH

---

<sup>1</sup>Some organizations don't even let ratings fall beneath a certain level, for example, the lowest possible USCF rating is 100.

## 2 ELO WORLD

Elo World is a tournament with dozens of computer players. The computer players include some traditional chess engines, but also many algorithms chosen for their simplicity, as well as some designed to be competitively bad.

The fun part is the various algorithms, so let's get into that. First, some ground rules and guidelines:

- No resigning (forfeiting) or claiming a draw. The player will only be asked to make a move when there exists one, and it must choose a move in finite time. (In practice, most of the players are extremely fast, with the slowest ones using about one second of CPU time per move.)
- The player can retain state for the game, and executes moves sequentially (for either the white or black pieces), but cannot have state meaningfully span games. For example, it is not permitted to do man-in-the-middle attacks [4] or learn opponent's moves from previous rounds, or to get better at chess. The majority of players are actually completely stateless, just a function of type `position → move`.
- The player should try to "behave the same" when playing with the white or black pieces. Of course this can't be literally true, and in fact some algorithms can't be symmetric, so it's, like, a suggestion.
- Avoid game-tree search. Minimax, alpha-beta, etc. are the correct way to play chess programmatically. They are well-studied (i.e., boring) and effective, and so not well suited to our problem. A less obvious issue is that they are endlessly parameterizable, for example the search








	The player is deterministic
	Traditional approach to chess (e.g. engine)
	Vegetarian or vegetarian option available
	A canonical algorithm!
	Stateful (not including pseudorandom pool)
	Asymmetric

Fig. 1. Key

ply; this leaves us with a million things to fiddle with. In any case, several traditional chess engines are included for comparison.

### 2.1 Simple players

**random\_move.** We must begin with the most canonical of all strategies: Choosing a legal move uniformly at random. This is a lovely choice for Elo World, for several reasons: It is simple to describe. It is clearly canonical, in that anyone undertaking a similar project would come up with the same thing. It is capable of producing any sequence of moves, and thus completely spans the gamut from the worst possible player to the best. If we run the tournament long enough, it will eventually at least draw games even against a hypothetical perfect opponent, a sort of Boltzmann Brilliancy. Note that this strategy actually does keep state (the pseudorandom pool), despite the admonition above. We can see this as not really state but a simulation of an external source of "true" randomness. Most other players fall back on making random moves to break ties or when their primary strategy does not apply. 

**same\_color.** When playing as white, put pieces on white squares. Vice versa for black. This is accomplished by counting the number of white pieces on white squares *after* each possible move, and then playing one of the moves that maximizes this number. Ties are broken randomly. Like many algorithms

described this way, it tends to reach a plateau where the metric cannot be increased in a single move, and then plays randomly along this local maximum (Figure 2). This particular strategy moves one knight at most once (because they always change color when moving) unless forced; on the other hand both bishops can be safely moved anywhere when the metric is maxed out.

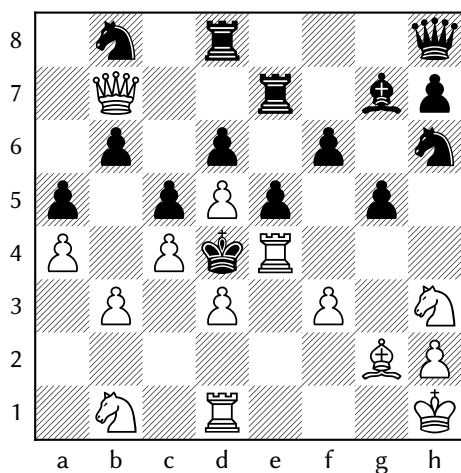


Fig. 2. `same_color` (playing as white) checkmates `same_color` (playing as black) on move 73. Note that since white opened with `Nh3`, the `h2` pawn is stuck on a black square. Moving the knight out of the way would require that move to be forced.

**opposite\_color.** Same idea, opposite parity.

**pacifist.** Avoid moves that mate the opponent, and failing that, avoid moves that check, and failing that, avoid moves that capture pieces, and failing that, capture lower value pieces. Break ties randomly. This is one of the worst strategies, drawing against players that are not ambitious about normal chess pursuits, and easily losing to simple strategies. On

the other hand, it does rarely get forced into mating its opponent by aggressive but weak players. ✓

**first\_move.** Make the lexicographically first legal move. The moves are ordered as  $\langle \text{src\_row}, \text{src\_col}, \text{dst\_row}, \text{dst\_col}, \text{promote\_to} \rangle$  for white (rank 1 is row 0) and the rows are reversed for black to make the strategy symmetric. Tends to produce draws (by repetition), because knights and rooks can often move back and forth on the first few files. (det)

**alphabetical.** Make the alphabetically first move, using standard PGN short notation (e.g. “a3” < “O-O” < “Qxg7”). White and black both try to move towards A1. (asym) (det)

**huddle.** As white, make moves that minimize the total distance between white pieces and the white king. Distance is the Chebyshev distance, which is the number of moves a king takes to move between squares. This forms a defensive wall around the king (Figure 3).

**swarm.** Like `huddle`, but instead move pieces such that they are close to opponent’s king. This is essentially an all-out attack with no planning, and manages to be one of the better-performing “simple” strategies. From the bloodbaths it creates, it even manages a few victories against strong opponents (Figure 4).

**generous.** Move so as to maximize the number of opponent moves that capture our pieces, weighting by the value of the offered piece ( $\text{♙} = 1$ ,  $\text{♘} = \text{♗} = 3$ ,  $\text{♖} = 5$ ,  $\text{♔} = 9$ ). A piece that can be captured multiple ways is counted multiple times.

**no\_i\_insist.** Like `generous`, but be overwhelmingly polite by trying to *force* the opponent to accept the gift of material. There are three tiers: Avoid at

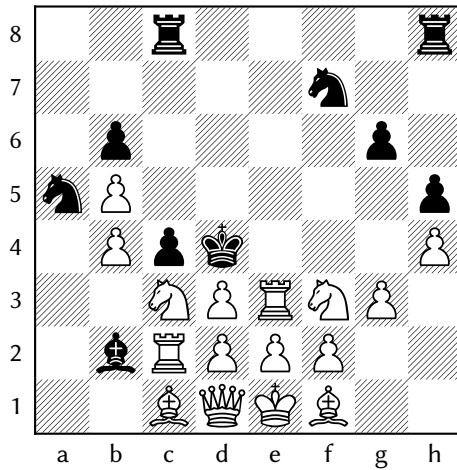


Fig. 3. huddle (white) checkmates pacifist on move 158 with substantial help. Note that white's distal pawns have advanced; these are actually the same distance from the King as they would be in their home row, since the distance metric includes diagonal moves.

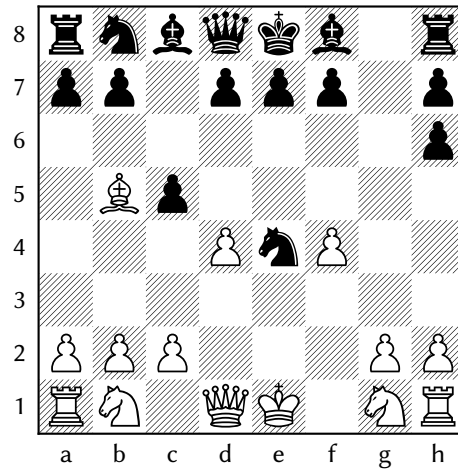


Fig. 4. swarm (white) vs. stockfish1m\_r4096 with black to move after 1. d4 Nf6 2. Bh6 gxh6 3. f4 c5 4. e4 Nxe4 5. Bb5. Black blunders 5...f6??. which must have been a random move as swarm immediately wins with 6. Qh5++.

all costs mating the opponent (and moreso checkmating). Stalemate is polite, but it is more canonical for two polite players to form a draw by repetition, from continually offering up material to one another and declining it. Next, avoid situations where the opponent can refuse our gift; among these, prefer the move where the opponent must capture the highest value piece. Finally, prefer moves where the expected value of the offered material (i.e. against random play) is highest. (This means that if there are three moves, and one captures a rook but the others capture nothing, the value is 5/3.) This strategy almost never wins, but is not among the worst players, since it often forces a draw by exchanging all its pieces.

**reverse\_starting.** This player thinks that the board is upside-down, and as white, tries to put its pieces where black pieces start. Since here we have a specific configuration in mind, we can produce a

distance metric by computing the total distance from each piece to its desired destination. Each piece uses a different distance metric; Chebyshev distance for the King, Manhattan distance for the Rook, and a pretty weird function for the Knight [3]. Trying to move the pieces into the reversed starting position often causes some conflict since the black pieces are already there, but can also end peacefully (Figure 5).

**cccp.** Prioritize moves that Checkmate, Check, Capture or Push, in that order. Push means to move pieces as deep as possible into enemy territory (without any regard for their safety). Ties are broken deterministically by the source/destination squares, so this one is technically asymmetric. [det](#) [asym](#) [За здоровье!](#)

**suicide\_king.** Take a random move that minimizes the distance between the two kings. Putting

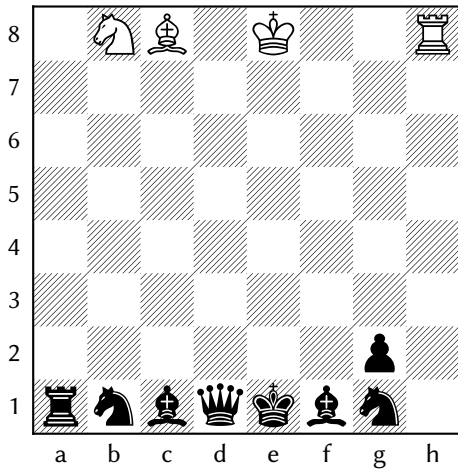


Fig. 5. reverse\_starting draws against itself by repetition, both players having happily moved their surviving pieces into the correct spots.

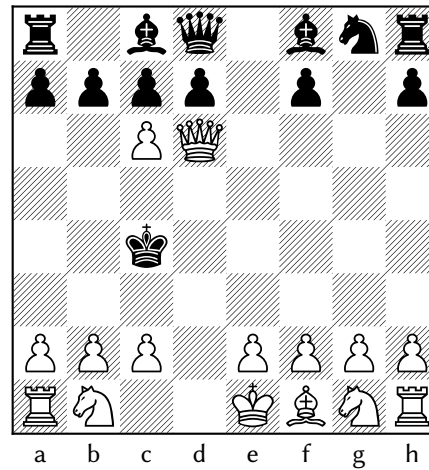


Fig. 6. suicide\_king (black) dramatically failing against cccp's slightly more principled play, after 1. d4 g5 2. Bxg5 Nc6 3. Bxe7 Kxe7 4. d5 Kd6 5. dxc6+ Kc5 6. Qd6+ Kc4. White delivers a discovered mate with 7. e4++.

one's king out in the open is very unprincipled (Figure 6), but it does produce enough pressure to win against unambitious opponents.

**sym\_mirror\_y.** As white, try to maximize symmetry when flipping vertically. Zero penalty for opposing e.g. a ♔ with a ♚, but a small penalty when the pieces are not the same type, and a larger penalty when they are not opposite colors. The starting position is already symmetric this way, so this usually has the effect of copying the opponent's moves when possible. As "copy your opponent's moves" is a common (but underspecified) strategy discovered by many children, this player is close to being canonical. However, it admits a bit too much arbitrary choice in the penalties assigned.

**sym\_mirror\_x.** As *sym\_mirror\_y*, but maximize symmetry when flipping horizontally. This does not make much chess sense, but can produce aesthetic arrangements.

**sym\_180.** As *sym\_mirror\_y*, but maximize symmetry under 180° rotation of the board (Figure 7). An emergent priority is to "castle" with the king and queen to "fix" them.

**min\_oppt\_moves.** Take a move that minimizes the number of resulting legal moves for the opponent, breaking ties randomly. This is a beautifully simple approach that generalizes many chess principles: Occupying space reduces the destination squares available to the opponent; capturing their pieces reduces the number of their pieces that they can move; pinning pieces or checking the king further reduces the legal moves; and mating the opponent is the best possible move.<sup>2</sup> Among the players in the paper, this one is Pareto efficient in terms of its simplicity and effectiveness. canon

<sup>2</sup>However note that it does not distinguish checkmate and stalemate, despite these having very different results.

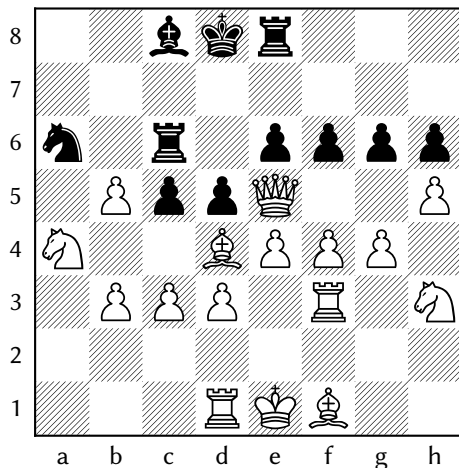


Fig. 7. sym\_180 (white) vs. pacifist after 66 moves. Note that the position is not quite rotationally symmetric, but is close given the material.

**equalizer.** Prefer to move a piece that has been moved the fewest times, and then prefer moving to a square that has been visited the fewest times. Castling counts as moving both the king and rook, and visiting both destination squares. This is the first strategy described that keeps meaningful state. (canon) (state)

## 2.2 Fate-based players

If we keep state, then we can track the location of each piece as it moves around the board (allowing us to distinguish the two rooks, or follow a pawn as it promotes). We can then use statistics on the average fates of each piece over hundreds of millions of games to guide the moves. These statistics give us, for each piece (e.g. the c2 pawn) and square, how likely it is to end the game on that square, and how likely it is to be alive or dead there when the game ends [5].

**safe.** This strategy moves pieces towards squares where they are likely to end the game alive. For this strategy and several others, simply moving to maximize this score (e.g. its sum or product over all pieces) is very boring, since the score is almost always maximized in the starting position. So this strategy actually makes each move randomly, weighted by the total score of the resulting positions. The scores are normalized (with the lowest-scoring move receiving weight 0.0 and the highest 1.0) and then sampled according to these weights. Without normalization, the play is almost identical to uniformly random, since the weights of the resulting positions tend to be very similar (dominated by the many pieces that don't move). But it's pretty arbitrary. (state)

**popular.** Like safe, but the score for a piece/square pair is the probability that the piece ends on that square, whether it lives or dies. This player likes to follow the crowd! (state)

**dangerous.** The dual of safe; the score is the probability that the piece dies on that square. Note that a king is said to die if it is checkmated or his side resigns. This player likes to live life on the edge! (state)

**rare.** The dual of dangerous; the score is one minus the probability of ending the game on that square. This player has a thirst for adventure! (state)

**survivalist.** Like the above, but the score is the ratio of the survival and death probabilities on the square. In the data set, every piece ends alive or dead in every square (except for the bishops, which can only legally occupy squares of their color) at least 1000 times, so each ratio is defined. Here, the sums of ratios have plenty of variability, and the highest ratios are not so often on the starting squares. So with this strategy, we simply do a weighted sample

from the moves according to their (non-normalized) scores. [state](#)

**fatalist.** The dual of `survivalist`; the score is the ratio of the death and survival probability on the square. This player knows that even if you win, you just have to keep playing over and over again, so you might as well get it over with! [state](#)

### 2.3 Engine-based players

Of course, people have been making more serious attempts at automating chess since before computers, and there are thousands of chess engines out there. We include a few in here to represent the high end of skill, and to make sure that weaker players are evaluated somewhat in terms of their ability to play chess proper, not just beat other weak players.

**stockfish0.** Stockfish [6] is probably the strongest open-source chess engine (or even publicly available engine); at full strength its play is estimated to be around 3500 Elo on the FIDE scale. Aside from being quite machine-dependent (it can search more moves in a given amount of time when it has a fast CPU with many cores), there are many options to fiddle with. Stockfish can use both opening books and endgame tables; neither is used here. It also has a “difficulty” setting, which is set here to 0. Stockfish is run in a separate process, and the board is reset before each move, but I am not extremely hygienic about flushing e.g. internal hash tables between moves. One consequence of this attempt at statelessness is that Stockfish sometimes walks into a draw by repetition in positions where it would be able to win, because it doesn’t know that it is repeating positions. [trad](#)

**stockfish5.** Stockfish as above, but at difficulty 5. [trad](#)

**stockfish10.** Same, at difficulty 10. [trad](#)

**stockfish15.** Same, at difficulty 15. [trad](#)

**stockfish20.** And difficulty 20, the maximum. Even at this difficulty, Stockfish produces moves basically instantaneously. [trad](#)

**stockfish1m.** As expected, the engine’s performance increases steadily as the difficulty setting increases (without apparently affecting the time to make moves). I don’t know what it’s doing with these settings. The true way to unleash chess engines is to give them a lot of CPU and memory to search. Since the tournament is run simultaneously across about 60 threads<sup>3</sup> using dozens of gigabytes of memory, and sometimes I would play *Hollow Knight* (aka ♁) while it ran, I wanted to avoid having the chess skill be dependent on the scheduling environment. So here, Stockfish is given a “node” budget of 1 million, hence 1m. It takes about one second per move when running alone, and is easily the strongest player (type) evaluated. [trad](#)

**worstfish.** On the other hand, a strong chess engine can also be used to play badly. When playing as white, for each legal move, I ask Stockfish (configured as `stockfish0`) to evaluate the resulting position from black’s perspective.<sup>4</sup> I then choose the move that produces the best position for black. This is easily the worst player evaluated, but it is not hard to imagine ways it could be worse. Indeed, a common twist on chess is to play for a loss, called *Antichess* or *Losing Chess* [9]. Recently it was even proved that white can always win (i.e. lose) [8] in this variant! However, the variant requires that you capture a

<sup>3</sup> The computer is the completely egregious AMD 2990WX “Threadripper 2,” which has 32 cores (for 64 hardware threads) and 250 Watts of power dissipation at load. The torture of this CPU was part of the impetus for the paper.

<sup>4</sup>By asking it to make a move, which also returns the evaluation of its move. The UCI protocol does not seem to offer a way to evaluate a position directly.

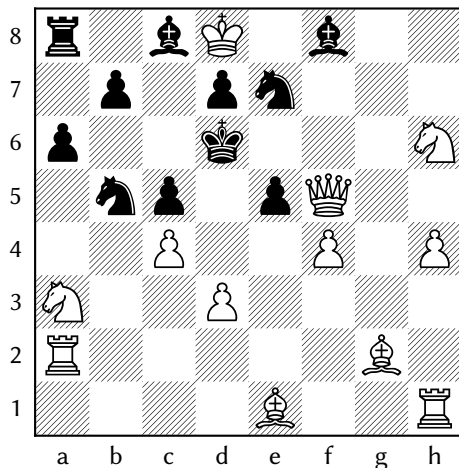


Fig. 8. suicide\_king (white) to move against worstfish after 36 moves. Since the kings are already at their minimum distance, white will make a move at random. 37. Qxe5++ wins for white immediately, but suicide\_king plays 37. Qxd7??. The only legal move is 37...Bxd7++, so worstfish must play it, and thus wins one of its only victories.

piece if you are able to, so strategies and engines that support this variant cannot be directly applied. We could use stronger settings of Stockfish, but since it already invokes Stockfish for each legal move, it is also one of the slowest players. Like Stockfish, it occasionally blunders an otherwise losing position into a draw by repetition. But most importantly, its search strategy when evaluating positions is not applying the correct logic ( $\alpha$ - $\beta$  pruning); it assumes that its opponent will choose strong moves, and that it will itself play strong moves in future plies. As a result, it sometimes allows the opponent to create a situation where worstfish is forced to checkmate its opponent (Figure 8).

**topple10k.** Topple [7] is another strong open source engine, provided to keep Stockfish on its toes. Here, its node budget is 10,000. [trad](#)

**topple1m.** Topple with a node budget of 1 million, which like stockfish1m takes about one second per move. Stockfish almost always wins, though it is not clear whether the budgets are actually comparable. [trad](#)

**chessmaster.nes\_lv1.** This is *Chessmaster* for the Nintendo Entertainment System, published in AD 1989. Moves are extracted from the game via emulation. This proved to be somewhat delicate, because the in-memory representation of the board is not that simple (it appears to be represented in two parallel arrays, perhaps using the “0x88 method”) and the game understandably goes wild if you mess it up. To play a move, I restore a saved machine state in the “board editor” mode, and then modify memory to contain the desired board. I then emulate button presses to operate the game menu and return to “playing” mode. Chessmaster repairs its internal data structures from this board, and makes a move. During this time I mash controller buttons to skip its dramatic tunes and modal messages like CHECK. Once some memory locations reach certain values, the move has been played; I can diff the before and after boards to uniquely determine the move. Since this approach uses emulation, it would normally be completely deterministic, but I deliberately stall for a random number of frames so that it can advance its internal pseudorandom state. A nice thing about this engine is that it is an earnest attempt at writing a good engine, but limited to the techniques of the 1980s, and running on hardware that was underpowered even for its day. It finishes well behind the



modern engines, but ahead of all the non-traditional strategies. [trad](#)

**chessmaster.nes\_lv2.** As above, but increasing the “Level Of Play” to “Newcomer 2.” Chessmaster has stronger levels still, but in these modes it will think for several NES minutes, which takes multiple seconds to emulate—too slow for our purposes. Still much weaker than modern engines (Figure 9). [trad](#)



Fig. 9. `stockfish1m` (2019; running on a modern machine) beats `chessmaster.nes_lv2` (1989; running on an emulated NES) thousands of times without a loss or draw.

## 2.4 Blind players

**blind\_yolo.** This player is only allowed to see the 64-bit mask of where pieces are placed on the board, but not their colors or types. It is described in *Color- and Piece-blind Chess* [4].

**blind\_kings.** As `blind_yolo`, but forcing the prediction to produce exactly one king for each side, which improves its accuracy a little.

**blind\_spycheck.** As `blind_kings`, but performing a “spy check” before each move. Here, the player tries capturing each piece of a given (predicted) color with other pieces of that same (predicted) color. If

such a move is legal, then one of the two pieces was mispredicted, so we prefer the capture over sending an incorrect position to the engine.

Each of these uses a neural network to predict the configuration of the board, and then the equivalent of `stockfish1m` to make a move for that predicted board. (If that move is illegal because the board was mispredicted, then it plays a random legal move.) These players can therefore be seen as handicaps on `stockfish1m`.

## 2.5 Interpolation methods

Even at its weakest setting, `stockfish0` crushes all of the nontraditional strategies. This is not surprising, but it creates a problem for quantifying their difference in skill (do they win one in a million games? One in  $10^{100}$ ?). Having intermediate players allows some Elo points to flow between the two tiers transitively. One nice way to construct intermediate players is by interpolation. We can interpolate between any two players,<sup>5</sup> but it is most natural to interpolate with the most canonical player, `random_move`.

**stockfish1m\_rnnn.** There are 15 players in this group, each characterized by a number *nnn*. Before each move, we generate a random 16-bit number; if the number is less than *nnn* then we play a random move and otherwise, a move as `stockfish1m`. The values of *nnn* are chosen so that we mix a power-of-two fraction of noise: `stockfish1m_r32768` blends half random with half strong moves, but we also have 1/4 random, 1/8 random, 1/16, 1/32, ..., 1/1024. These give us a nice smooth gradation at high levels of play (Figure 11). At the extremes, many games

<sup>5</sup> Even if they are stateful! The interface separates “please give me a move for the current state” and “the following move was played; please update your state,” allowing them to disagree. So we can just keep two independent states for the two players.

have no random moves, and the performance is difficult to distinguish from `stockfish1m`. Even at half random moves, `stockfish1m_r32768` consistently beats the non-traditional players. So we also dilute `stockfish` by mixing with a majority of random moves: `stockfish1m_r49152` is 3/4 random, and we also have 7/8, 15/16, 31/32, and 63/64. At this point it becomes hard to distinguish from `random_move`.

Note that playing 1/64 excellent moves and the rest randomly doesn't do much to help one's performance. On the other hand, playing one random move for 63/64 strong ones does create a significant handicap! Against strong opponents, it's easy to see how a mistake can end the game. Even against weak ones, a blunder can be fatal (Figure 10).

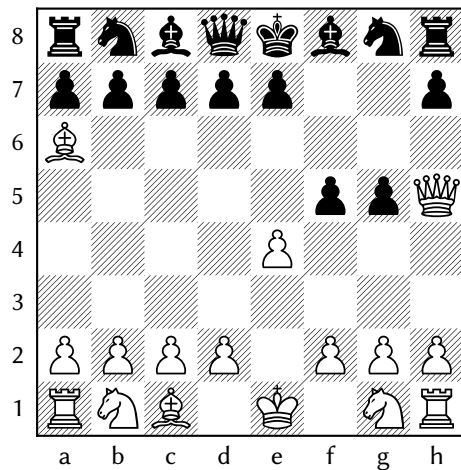


Fig. 10. A demonstration of how quickly random play can ruin a game. `stockfish1m_r32768` (black; plays half random moves and half strong moves) loses to `reverse_starting` in one of the shortest possible sequences: 1. e4 g5 2. Ba6 f5 3. Qh5++. Both of black's moves are bad (so they must be random) but white's 2. Ba6 is a blunder as well. White is just pushing pieces as far as possible; the mate is purely accidental.

### 3 RUNNING THE TOURNAMENT

Given the players, it's a simple matter to simulate a bunch of games. Since the tournament runs for tens of thousands of CPU hours, there is of course some code to allow it to checkpoint its work, to make efficient use of CPU and RAM, and to provide attractive ANSI graphics showing off its activity—but this is all straightforward.<sup>6</sup> The tournament accumulates win/loss/draw counts for each pair of players (playing as both white and black), as well as example games used to debug or create the figures in this paper.

After the tournament is done, or from any checkpoint, I can run a separate program to produce Elo (and other) rankings. Elo depends on the order in which games are played, because it is designed to take into account changes in player skill over time. However, it is easy to just make up a random order in which the games were played, because these players do not change over time.

Despite its strong reputation, I found Elo to be rather finicky. It is sensitive to the starting scores for the players, and the factor  $k$  which governs how strong of a correction to make after a game. With players of such vastly different skill, it is also very sensitive to the order in which the games are played.<sup>7</sup> For similar reasons, it is also very sensitive to imbalance in the number of games played between a particular pair of players; if some player mostly has games against weak players, this can artificially inflate its score.

<sup>6</sup> The source code can be found at [sourceforge.net/p/tom7misc/svn/HEAD/tree/trunk/chess/](https://sourceforge.net/p/tom7misc/svn/HEAD/tree/trunk/chess/)

<sup>7</sup>For example, suppose the weak player `random_move` has one win against `stockfish1m`. If this win happens early, when the two players have their initial Elo scores, then it doesn't affect anything. If it happens last, when `stockfish1m` has thousands of Elo points over `random_move`, then it produces a very strong correction.

To control for these effects, I compute the maximum number  $n$  for which we have  $n$  games played between every pair of players. I then seed each player with an initial Elo score of 1000, and sample exactly  $n$  games to play from each cell without replacement. This is so that we play a balanced number of games. I also randomize the global order of all games. Then I do this over again, with a smaller update factor  $k$ , for 20 passes. I don't know whether it's the fact that I reduce  $k$  over time, or that I end up with effectively 20 times the number of games played,<sup>8</sup> but doing this significantly reduces variance. I run the whole thing 19 times and the result is the median Elo score. I also computed the 25<sup>th</sup> and 75<sup>th</sup> percentiles, which were within 1–2% of the median, which is good. The Elo scores appear in Section 4;  $n > 400$  for this run.

Ideally, such a system would be robust against adding new players, but this is probably not the case; it is easy to see from the results (Figure 11) that there are certain matchups that favor one of the players despite its average skill. During development, I often noticed significant swings in Elo scores as players were added, especially before there were middle-tier players in the mix. One way to deal with this would be to run the Elo simulations multiple times while randomly ablating players from the tournament; we could then at least estimate the variance in the Elo scores under the removal of players, which is like adding players in reverse. I did not implement such a thing because of oppressive SIGBOVIK deadlines.

## 4 RESULTS

The main use of the Elo World tournament is to benchmark some new engine or algorithm for playing chess, particularly if it is not that good [4]. There are a few ways that we can interpret the results:

The **Elo score**, as described.

We can find a **comparable interpolated player**. This is like the Scoville scale for measuring how spicy something is: Some golden-tongued blind tasters are given the pure chili oil and asked to distinguish it from a cup of sugar water, which they of course can do. They then dilute the chili oil 1:1 with sugar water, and try again. The number of dilutions before the testers cannot reliably tell the drink from sugar water yields the score on the Scoville scale. Here, for example, we can say that `blind_spycheck` performs between `stockfish1m_r63488` and `_r61440`, so it is approximately a 93.75–96.875% diluted Stockfish.

We can compute a **Markov probability**. The tournament table can be easily thought of as a Markov transition matrix. Imagine that there is a Champion trophy, held by the player corresponding to some row. Each cell in that row contains the probability that in a game between those two players, the trophy would be passed to that player. We treat draws as choosing one of the two sides by a coin flip (or, equivalently, splitting the trophy in half); and for the games that the row player wins, the trophy is retained (probability mass assigned to a self-transition). It is easy to compute this matrix from the win/loss/draw counts in the tournament table, and it is not sensitive to imbalance like the Elo calculation is. Presented this way, we can compute the stationary distribution (if it exists, which it typically will), which basically tells

---

<sup>8</sup>To be clear, running 20 passes means that games can be reused, which is not ideal.

us that after an extremely large number of games, what is the probability that a given player holds the Champion trophy? This tends to agree with the Elo score, but there are a few places where it does not (e.g. `same_color` has a much higher `p(Champion)` score than its Elo seems to warrant).

And so finally, a table with numbers:

Player	Elo	p(Champion)
worstfish	188.54	0.00000071
pacifist	286.90	0.00000509
alphabetical	320.91	0.00000276
generous	342.71	0.00000306
popular	349.48	0.00000374
dangerous	349.60	0.00000291
safe	350.64	0.00000454
first_move	357.71	0.00000434
rare	361.39	0.00000323
no_i_insist	378.23	0.00000244
huddle	399.55	0.00000933
sym_180	429.94	0.00000331
same_color	430.90	0.00002590
opposite_color	432.11	0.00000293
sym_mirror_x	438.87	0.00000305
survivalist	439.03	0.00000681
random_move	439.21	0.00000462
fatalist	439.89	0.00000314
sym_mirror_y	442.50	0.00000716
reverse_starting	445.56	0.00000301
suicide_king	447.02	0.00000493
stockfish1m_r64512	462.82	0.00000297
stockfish1m_r63488	482.44	0.00000356
blind_yolo	488.97	0.00000488
...		

... Player	Elo	p(Champion)
blind_kings	501.98	0.00000633
equalizer	504.21	0.00000653
stockfish1m_r61440	521.45	0.00001173
swarm	534.03	0.00000623
blind_spycheck	546.91	0.00002554
cccp	553.54	0.00000544
min_oppt_moves	597.04	0.00001076
stockfish1m_r57344	600.54	0.00001112
stockfish1m_r49152	752.22	0.00000737
chessmaster.nes_lv1	776.15	0.00002055
stockfish1m_r32768	976.20	0.00003025
chessmaster.nes_lv2	989.21	0.00012094
stockfish1m_r16384	1277.73	0.00012509
stockfish0	1335.69	0.00008494
stockfish5	1644.63	0.00070207
stockfish1m_r8192	1690.15	0.00152052
topple10k	1771.65	0.00179298
stockfish10	1915.23	0.00257648
stockfish15	1952.93	0.00310436
stockfish1m_r4096	2020.25	0.00886928
stockfish20	2139.47	0.00721173
topple1m	2218.71	0.01091286
stockfish1m_r2048	2261.50	0.03132272
stockfish1m_r1024	2425.72	0.06759788
stockfish1m_r512	2521.78	0.11122236
stockfish1m_r256	2581.97	0.15364889
stockfish1m_r128	2609.45	0.17861429
stockfish1m_r64	2637.78	0.20508090
stockfish1m	2644.10	0.21523142

## 5 CONCLUSION

Shout out to the Thursd'z Institute and anonymous commenters on my blog for discussion of players and suggestions. Several ideas were suggested by multiple people, increasing my confidence that they are somehow canonical.

The author would also like to thank the anonymous referees of the Special Interest Group on Bafflingly Overdone Ventures In Chess journal for their careful reviews.

## REFERENCES

- [1] 2017. FIDE Handbook – B.. Permanent conditions. [www.fide.com/component/handbook?id=2](http://www.fide.com/component/handbook?id=2).
- [2] Arpad E Elo. 1978. *The rating of chessplayers, past and present*. Arco Pub.
- [3] Amanda M. Miller and David L. Farnsworth. 2013. Counting the Number of Squares Reachable in k Knight’s Moves. *Open Journal of Discrete Mathematics* 03, 03 (2013), 151–154. <https://doi.org/10.4236/ojdm.2013.33027>
- [4] Tom Murphy, VII. 2019. Color- and piece-blind chess. In *A Record of the Proceedings of SIGBOVIK 2019*. ACH.
- [5] Tom Murphy, VII. 2019. Survival in chessland. In *A Record of the Proceedings of SIGBOVIK 2019*. ACH.
- [6] Tord Romstad, Marco Costalba, and Joona Kiiski. 2019. Stockfish Chess. <https://stockfishchess.org/>.
- [7] Vincent Tang. 2019. Topple. <https://github.com/konsolas/ToppleChess/releases/>.
- [8] Mark Watkins. 2017. Losing Chess: 1. e3 Wins for White. *ICGA Journal* 39, 2 (2017), 123–125.
- [9] Wikipedia. [n. d.]. Losing Chess. [http://en.wikipedia.org/wiki/Losing\\_Chess](http://en.wikipedia.org/wiki/Losing_Chess).

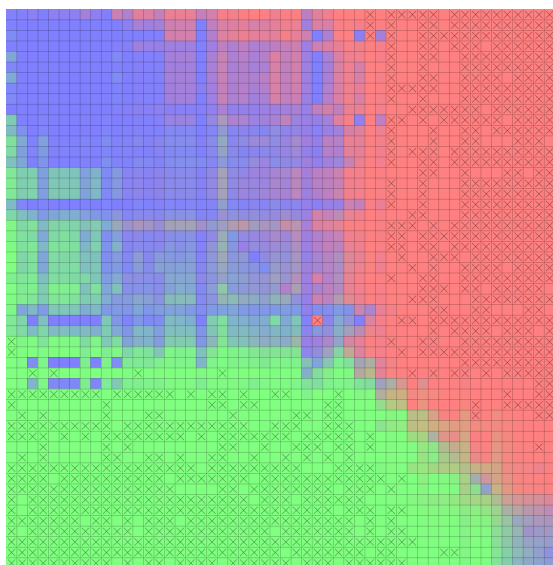


Fig. 11. The matrix of outcomes for all players. There is too much data to print, so we just have a bunch of colors, mostly because it looks rad. If you don’t have a color printer or TV it will probably not look rad. Rows indicate the player playing as white, from worst (worstfish, top) to best (stockfish1m, bottom). Columns for the player with the black pieces, in the same order from left to right. Green indicates a cell that is predominately wins (for white), red is losses, and blue is draws. The right and bottom third of the graphic are all different levels/dilutions of strong engines (Stockfish and Topple). This creates a nice smooth gradient where better engines regularly beat even slightly weaker ones, but start to produce draws at the highest levels. They consistently destroy weak engines; a cell with an  $\times$  indicates that it only contained wins or only contained losses (not even draws).

At the low end, there is a large splat of matchups that mostly produce draws against one another, but can consistently beat the weakest tier. In the top-left corner, a square of blue draws from low-ambition play; these aggressively bad players almost never win games, even when playing each other.

Microtexture outside of these broad trends comes from matchups where the player is unusually suited or weak against that particular opponent. For example, the bright red cell on the diagonal near the center is cccp vs. cccp; this deterministic strategy always wins in self-play as black.