Chess Mate-in-N Puzzle Composition by Random Generations

André Luis Bradacz[1], **Pedro Belin Castellucci**[1,2]

[1]Departamento de Informática e Estatística – Universidade Federal de Santa Catarina
Caixa Postal 5064 – 88035-972 – Florianópolis – SC – Brasil

andre.bradacz@ufsc.br

[2]Centro de Artes, Humanidades e Tecnologia – Universidade Federal de São Carlos
R. Dr. Eduardo Nielsen, 420 – 15030-070 – São José do Rio Preto – SP – Brasil

pedrobc@ufscar.com

Abstract. Introduction*:Chess mate-in-N problems (puzzles) are positions on a chessboard in which the attacking side must find a forced checkmate in exactly N moves. The automatic generation of mate-in-N problems is a computational challenge due to the size of the search space. The few approaches in the literature typically use some kind of prior knowledge (e.g. a known puzzle position or database of patterns) to generate such puzzles.* ***Objective****: We propose and investigate the generation of mate-in-N problems without prior knowledge.* ***Methodology****: We generate chess puzzles from random positions with a defined set of pieces. With a method for ensuring uniqueness of solution, we experimentally evaluate the computational effort and effectiveness of this approach.* ***Results****: The experiments indicate that the approach is not only viable but also simpler and competitive with state-of-the-art approaches.*
Keywords *Computational chess, random search, mate-in-n.*

1. Introduction

Chess problems (puzzles) of finding a forced checkmate in N moves (mate-in-N problems) are widely used to teach piece coordination, for tactical training and also for entertainment. The problem consists of finding, given a chess position, a sequence of moves for the attacking side (white) forcing a checkmate for every possible response of the defending side (black).

The automatic generation of this kind of problem is a challenging computational task due to the size of the search space. For chess positions reachable from the initial board state, this number is estimated to be in the order of 10^{37} [Gourion 2022]. For chess puzzles, this number is even larger since the positions do not have to be reachable from the starting position of a typical game.

Due to this huge search space, the few attempts to automatically generate chess problems typically rely on prior knowledge. As we discuss in Section 2, existing approaches either start from an existing puzzle and try to improve it using heuristic rules or they use a database of games and/or positions in an attempt to identify interesting patterns to build puzzles from.

In a somewhat counter-intuitive approach, we investigate the possibility of generating mate-in-N problems by creating random positions. This approach would have been infeasible in the early days of computational chess due to the reduced computational

power. However, with current personal computational resources, we show that it is viable to generate puzzles of sizes compatible with state-of-the-art methods.

To the best of our knowledge, this is the first time a random exploration of the search space of a chess game is investigated for chess puzzle generation without prior knowledge. Our contributions are twofold: (i) we provide empirical evidence that generating chess puzzles with unique solutions using random exploration of the search space is a viable option for generating mate-in-N positions, in a way that is simpler, less resource intensive and competitive with state-of-the-art methods; (ii) We show experimentally that evaluating the moves in different orders have little to no effect in the search for a puzzle.

The remainder of this paper is structured to present the related works in Section 2. Then, we properly define the problem of finding mate-in-N puzzles in Section 3. In Section 4, we present the computational method used to find such problems and verify its uniqueness. The experiments and corresponding results are discussed in Section 5. Finally, Section 6 presents some final remarks and future work.

2. Related works

Since the early estimate by [Shannon 1950] that the number of possible chess positions is bounded by 10^{43}, very little research has been done on improving this estimate. [Steinerberger 2015] estimated an upper bound on the number of positions of $2 \cdot 10^{40}$ by considering specifics about the movements of bishops and pawns. More recently, [Gourion 2022] enhanced this bound to $4 \cdot 10^{37}$ by restricting pawn positioning. To the best of our knowledge, there is no research considering the number of positions that have a forced checkmate. It is also worth noting that, in chess problem composition (e.g. mate-in-N puzzles), the positions do not have to be reachable from the starting position of the pieces. Intuitively, the number of positions with forced checkmate should be small, but we show that, due to current computational power, it is viable to use random position generation to find mate-in-N problems.

Despite the early references (e.g. [Schlosser 1988]), the automatic composition of chess puzzles is a largely overlooked theme in computational chess [Fainshtein and HaCohen-Kerner 2006]. Much more frequent are works focusing on solving puzzles [Maharaj et al. 2022, Björkqvist 2024] or playing a full game [Upasani et al. 2021, Patel et al. 2022]. Classical strategies related to chess are based on game theory and search techniques [von Neumann and Morgenstern 1944, Shannon 1950]. In this context, alpha-beta pruning is one of the key strategies for reducing the search space [Knuth and Moore 1975].

[Fainshtein and HaCohen-Kerner 2006] proposed a mate-in-two chess composer that actually starts from an original puzzle and uses a heuristic search to improve the original position. [Iqbal 2011] used statistics from a database of mate-in-three problems to compose other mate-in-three positions starting from an empty board. The problems were evaluated based on the presence of tactical features (e.g. number of pieces, pins, material sacrifice). [Iqbal 2021] proposed a method for improving chess puzzles based on heuristic rules (e.g. removing pieces, swapping attacking pieces for weaker ones, looking for shorter mates). More recently, [Feng et al. 2025] tried to leverage the power

of generative AI to build high quality puzzles. The authors claimed that using generative AI, reinforcement learning and a database of 4.4 million training datapoints, they generate high quality problems.

Similar challenges are explored for different games like Sudoku [Oliveira 2024]. [Santos 2022] mention the importance of integrating generation and validation of puzzles, especially when uniqueness of solution is of interest. [Yannakakis and Togelius 2018] discussed other challenges related to generation for games.

On the topic of playing a full game, [Gillogly 1972] proposed an alpha-beta search based chess program that tries the moves that are captures first (sorted by highest value piece) and prioritizes moves that refute a position in the subsequent tries (*killer moves*). The engine also incorporates positional analysis based on phases of the game (e.g. opening, middle game, endgame) [Akl and Newborn 1977] proposed that finding the principal continuation (an optimal sequence of moves) for both players might be used to find killer moves. Their experiments consider king-pawn endgames. [Schaeffer 1989] highlights that the move ordering is critical to the success of alpha-beta search. Different strategies have been explored for improving the search strategy, including with neural networks [Kocsis et al. 2002].

Due to the combinatorial nature of the problem and the size of the search space, many strategies for chess problems generation are based on previous knowledge, either a database of positions or the existence of a problem to be improved upon. We, on the other hand, explore the search space by generating random positions and we show experimentally that it is possible to produce mate-in-N problems with N values compatible with the literature within reasonable computational times and much less computational power than some alternatives.

3. Problem definition

A mate-in-N problem consists of a chess position in which the white pieces can force a checkmate in exactly N moves. We investigate the computational challenge of generating a chess position satisfying:

1. The existence of a sequence N moves that forces a checkmate.
2. There must not be a shorter sequence leading to a checkmate.
3. There must not be a different sequence with a different first move.

Each sequence of moves is formed by *plies*. A *ply* corresponds to a single move by one of the players while a move is two *plies* (e.g. a move by the white pieces and a response by the black pieces). Therefore, to verify a checkmate in N moves, the search depth is $2N - 1$ plies.

Formally, let $Mate(s, n)$ be the number of different sequences that lead to a checkmate from an initial state (positioning of the pieces) s in exactly n plies. We search for an initial state s such that:

- $Mate(s, 2N - 1) = 1$
- $Mate(s, k) = 0$, for all $0 \leq k < 2N - 1$

4. The adaptative search algorithm

Beyond finding a forced checkmate, we want to experiment with different move ordering. Therefore, we need a customizable solution method. For this, consider the following

three-phase method: (i) generating a candidate solution, (ii) verifying the existence of shorter mates and (iii) uniqueness verification. The candidate solution is a position that has a checkmate in N moves, but there might be a shorter mate (verified in step (ii)) and there might be another checkmate (hence step (iii)).

All three steps are based on a recursive tree search. The game tree is represented as an *AND–OR* tree such that each level is associated with the decision of a player, alternately. The tree nodes corresponding to the attacking player (white) are *OR*-nodes and the defending player is represented with *AND*-nodes. In the *OR*-nodes, we search for any valid sequence of moves leading to a checkmate, whereas in the *AND*-nodes, every play must preserve the possibility of the checkmate, thereby resulting in a forced checkmate.

The algorithm has three input parameters: the current state of the board (`board`), the number of plies left (`n`) and an indicator of whose turn is being considered (`maximizing`), in which `true` (`false`) indicates White (Black). The pseudocode is presented in Algorithm 1, which we describe in detail.

The base cases indicate a terminal state of the game, either a checkmate or a stalemate. If a checkmate is reached before the N-th move or the board corresponds to a stalemate position, the search is pruned. On the other hand, if a checkmate is reached with exactly N (`n = 0`) moves, then the search attempt was a success. After checking the base cases, the algorithm proceeds according to player's turn.

When it is White to move, the algorithm tries all the legal moves in the current position. For each move, the state of the board is updated and the search continues recursively for Black. After the search, the move is undone, in preparation for the next move. The search returns whether there exists at least one sequence leading to a checkmate from that position. If there is a checkmate, a counter of the number of different moves that lead to a valid solution is updated. The algorithm returns 0 if no mate-in-N could be found, 1 if there is a unique solution to the mate-in-N problem, and 2 if there are multiple solutions. When it is Black to move, the algorithm verifies if all possible legal moves for Black preserve the possibility of a checkmate in the required number of moves. For this, the algorithm recursively searches all available moves. If any defensive move results in preventing the mate (return 0), the search is pruned, indicating there are no checkmates available (with result 0). If, despite Black move options, the checkmate is still available, the algorithm returns 1.

Algorithm 1 Main agorithm

```
1: function MATEINN(board, n, maximizing)
2:     if CHECKMATE(board) then
3:         return 1 if n = 0, else 0
4:     if STALEMATE(board) or n ≤ 0 then
5:         return 0
6:     if maximizing then
7:         total ← 0                    ▷ Count the number of initial movements leading to a checkmate
8:         for all m ∈ LEGALMOVES(board) do
9:             APPLYMOVE(board, m)
10:            res ← MATEINN(board, n − 1, false)
11:            UNDOMOVE(board, m)
12:            if res ≥ 1 then
13:                total ← total + 1
14:            if total > 1 then
15:                return 2                                  ▷ Prune due to multiple solutions
16:        return total
17:    else
18:        for all m ∈ LEGALMOVES(board) do
19:            APPLYMOVE(board, m)
20:            res ← MATEINN(board, n − 1, true)
21:            UNDOMOVE(board, m)
22:            if res = 0 then
23:                return 0                                  ▷ There is a defense
24:        return 1
```

In phase (i) of the solution, Algorithm 1 starts from a random position and with a limited number of nodes (in the search tree to be explored). If this threshold is reached, the search restarts with a new initial position. This threshold is important to prevent spending too much time in a position that has no forced checkmate. If a candidate solution is found, the solution is submitted to the next phase. In phase (ii), Algorithm 1 starts from the candidate position and without the node threshold to verify whether a shorter checkmate exists. If there is a shorter checkmate, then the search restarts at phase (i); otherwise, the solution is verified for uniqueness. In this phase, for each valid White move, the algorithm checks if there exists checkmate in exactly N moves.

Finally, different move orders can be considered, which are only applied for the attacking side. The function `legalMoves` considers different criteria for sorting the trial moves in the search for a mate, yielding different heuristic searches. The criteria are (in alphabetical order):

1. *Capture Value Ascending* (CVA) and *Capture Value Descending (CVD*: prioritizes captures of opponent pieces with lower (respectively, higher) material value.
2. *Check First* (CF): prioritizes moves that give check to the opponent's king.
3. *Default* (DEF): *python-chess* default behavior. It scans the board from first to last square (a1, a2, ..., h8) and generates moves for each piece in the order found.
4. *Distance to King* (DK): prioritizes moves that decrease the distance between attacking pieces and the opponent's king.
5. *Progressive Pressure* (PP): combines *Check First*, *King Mobility Reduction* and number of pieces attacking the opposing king.

6. *Piece Value Ascending* (PVA) and *Piece Value Descending* (PVD): prioritizes moves involving pieces in ascending (respectively, descending) order of their material value.
7. *Reduce King Moves* (RKM): prioritizes moves that reduce the number of legal moves available to the opponent's king.

5. Computational experiments

To evaluate the different strategies, the board is initialized from a random position, sampled from a subset of the pieces. First, the two kings are randomly positioned without attacking each other. Then, the positions (squares) on the board are shuffled and assigned to each piece in the subset in a sequential manner.

To define the subset of pieces, we considered two puzzle books. From [Polgár 2006], we select the first 20 in each section: *Mate in 1* and *Mate in 3*, and from [Bajarani 2024] we select the first 20 for mate-in-four for the white pieces problems. When different puzzles featured the same subset of pieces, the following puzzle was used instead. For mate-in-two problems, we find that the set of pieces in [Polgár 2006] has little variation, therefore we considered the same set for mate-in-one and mate-in-two problems. With these datasets, we performed two experiments. Firstly, we evaluate whether different move ordering could have an impact on the computational effort of identifying the puzzles (Subsection 5.1). Secondly, we perform the main experiment of evaluating the computational effort of finding mate-in-N problems by generating random positions (Subsection 5.2). In both experiments, we repeat the run 10 times for each subset of pieces and define a time limit of 300s for each repetition.

Regarding the computational environment, the experiments were run on a desktop with an AMD Ryzen 5 5500 processor and 32GB of RAM and Windows 11 operating system. The solution was implemented in Python, using the *python-chess* library. The chess engine Stockfish 18 was used to validate the solutions.

5.1. Evaluating move ordering

This section presents the results grouped by different values of N, in the mate-in-N problem for each heuristic (move ordering). Tables 1–3 summarize the results regarding time to find a puzzle. We report average execution time (Mean, s); standard deviation (SD); average time considering only successful runs (Mean (Succ.)); standard deviation considering only successful runs (SD (Succ.)). The number of executions and successful (that could find a puzzle) are also reported when necessary. For $N = 1$ and $N = 2$, the success rate was 100% and average time was always smaller than 2 seconds for every ordering heuristics. For each case, the number of executions is 200 (10 repetitions for each of the 20 sets of pieces considered).

The mate-in-three and mate-in-four cases are increasingly more challenging (Tables 2 and 3). The average time for heuristics for $N = 3$ increases by two orders of magnitude when compared to $N = 2$. Also, the first failures appear, even though the success rate is typically around 90%. For $N = 4$ successes are rare, consistently below 20% with average execution times higher than 250 seconds.

As expected, there is no heuristic that dominates the others. For each case, Tables 1–3, different move orderings were better on average. However, looking at the standard

Table 1. Results for mate in one and mate in two moves for the different heuristic ordering regarding computational time.

Mate-in-1				
Heuristic	**Mean**	**SD**	**Mean (Succ.)**	**SD (Succ.)**
CVA	0.29	0.55	0.29	0.55
CVD	0.24	0.37	0.24	0.37
CF	0.28	0.78	0.28	0.78
DEF	0.27	0.51	0.27	0.51
DK	0.27	0.48	0.27	0.48
PP	0.23	0.33	0.23	0.33
PVA	0.27	0.46	0.27	0.46
PVD	0.25	0.41	0.25	0.41
RKM	0.30	0.61	0.30	0.61
Mate-in-2				
CVA	1.38	1.05	1.38	1.05
CVD	1.45	1.15	1.45	1.15
CF	1.25	0.95	1.25	0.95
DEF	0.97	0.70	0.97	0.70
DK	1.35	1.02	1.35	1.02
PP	1.43	1.12	1.43	1.12
PVA	1.43	1.13	1.43	1.13
PVD	1.48	1.18	1.48	1.18
RKM	1.39	1.06	1.39	1.06

deviation, we see that the computational times vary greatly. This is aligned with the randomness of the generation and the vastness of the search space.

Table 2. Results regarding success rate and computational time for mate in three.

Heuristic	Exec.	Succ.	(%)	Mean (s)	SD (s)	Mean (Succ., s)	SD (Succ., s)
Capture Value Asc	200	180	90.00	110.15	101.88	88.95	63.87
Capture Value Desc	200	182	91.00	115.06	106.33	96.68	69.45
Check First	200	176	88.00	122.65	111.79	98.35	72.58
Default	200	178	89.00	124.18	113.02	102.34	75.31
Distance to King	200	180	90.00	118.88	108.40	98.64	71.22
Progressive Pressure	200	188	94.00	108.62	98.55	96.35	68.91
Piece Value Asc	200	167	83.50	122.42	109.77	87.10	60.54
Piece Value Desc	200	180	90.00	119.52	108.90	99.35	72.01
Reduce King Moves	200	168	84.00	128.22	115.34	95.32	70.44

Table 3. Results regarding success rate and computational time for mate in four.

Heuristic	Exec.	Succ.	(%)	Mean (s)	SD	Mean (Succ.)	SD (Succ.)
Capture Value Asc	200	30	15.00	276.43	187.56	140.30	77.35
Capture Value Desc	200	29	14.50	280.78	191.22	164.89	92.44
Check First	200	33	16.50	272.42	182.11	130.62	70.88
Default	200	36	18.00	271.92	180.45	141.98	76.92
Distance to King	200	28	14.00	279.34	188.73	149.67	84.10
Progressive Pressure	200	34	17.00	274.16	183.77	146.02	80.55
Piece Value Asc	200	35	17.50	272.39	181.98	139.97	75.88
Piece Value Desc	200	33	16.50	272.37	181.65	130.32	70.44
Reduce King Moves	200	33	16.50	276.21	186.90	153.59	88.21

Table 4. Summary of the number of nodes explored in each phase of the algorithm.

Mate in #	Generation	Shorter	Uniqueness	Total
1	5613.86	0.00	34.33	5648.19
2	154472.45	50.48	1209.73	155732.65
3	2576285.38	3502.16	57314.72	2637102.26
4	14997431.62	4024.41	38418.29	15039874.32

We also consider how the computational effort is distributed between the generation, the verification of shorter mates and uniqueness. As observed in Tables 1–3, there is no relevant difference between the move orderings so Table 4 summarizes the results considering only the *default* heuristic. The generation phase clearly dominates the computational costs. For different values of N, the generation explored a number of nodes that, on average, are at least two orders of magnitude higher than the other phases. It is also clear that the number of nodes that need to be explored increases rapidly with the number of moves to forced checkmate.

5.2. Evaluating computational effort of problem generation

Since we identified that move ordering heuristics do not have a decisive effect on the computational effort, the results reported in this section only consider the *default* move ordering. We report, for $N \in \{1, 2, 3, 4\}$, the average computational time (t(s)), the number of tries (T) until a problem was found for different number of pieces for White (W) and Black (B). We also report minimum and maximum number of tries (Min. T and Max T., respectively). Results are averaged across 10 repetitions.

The number of pieces does not seem to be correlated with the computational effort of finding a problem, for $N \in \{1, 2, 3, 4\}$ (Tables 5 and 6). Neither the computational time nor the number of tries seem to be determined by the number of pieces. There are cases in which the same number of pieces resulted in very different computational times (0.8s to 0.1s) and average number of tries (3440.0 to 348.9). We observed that the actual pieces might have a stronger relation to the computational effort, although further investigation is needed.

The computational effort increases when looking for longer mate sequences. Consistently, the average computational times are higher for the mate-in-two case, for example (Table 5). However, it might happen that the number of tries is lower for the mate-in-2, since it usually takes longer to evaluate mate-in-2 positions. Also, there is great variation in computational effort among the 10 repetitions with the same number of pieces. There is a big amplitude in the number of tries. This is expected, since minor changes in piece positioning might have a big impact on the nature of the game.

Mate-in-3 problems are more challenging to be found than mate-in-2 or mate-in-2 cases (Table 6). On average (and median), a mate-in-3 problem could be found in about 90% of the cases. The average time to find these problems increased by three orders of magnitude when comparing to mate-in-2 problems, but on average, the search finished in less than two minutes.

As already presented in Table 3, $N = 4$ is beyond the capabilities of this approach, on average (and median), less than 20% of the cases found a puzzle. Thus, we do not

Table 5. Summary of the results for mate in one and mate-in-2.

		Mate in 1				Mate in 2			
W	**B**	**t (s)**	**T**	**Min. T**	**Max. T**	**t (s)**	**T**	**Min. T**	**Max. T**
2	2	0.8	3440.0	586	10643	15.5	4278.2	51	11016
2	2	0.1	348.9	51	1187	1.0	400.2	56	773
2	3	0.2	718.8	18	2875	3.7	1062.2	49	1745
3	1	0.0	170.8	73	330	1.0	203.2	3	688
3	4	0.1	903.4	23	3196	6.4	3137.2	97	11924
3	4	0.2	1166.1	223	4207	2.3	1520.7	173	4963
4	3	0.5	3484.3	376	9152	5.4	2645.7	192	7347
4	4	0.2	1836.8	8	6354	2.7	2022.5	377	7190
4	4	0.1	768.7	35	2629	2.2	1377.7	63	4739
5	3	0.0	119.5	40	265	0.6	153.9	10	624
5	5	0.4	3669.7	314	8733	4.2	2822.4	79	7253
5	5	0.2	2938.4	106	6443	1.1	2474.7	257	7708
5	6	0.0	465.5	71	1324	2.1	2254.3	63	8934
5	7	0.3	3849.0	892	10273	3.1	4409.6	560	11309
6	6	0.1	838.1	2	3453	1.8	1723.7	15	5989
7	6	0.1	589.1	43	1703	0.6	590.0	81	960
7	7	0.2	2572.4	27	6487	1.0	1778.4	53	4451
7	7	0.2	2361.7	3	8611	0.8	2090.4	429	5067
9	9	0.3	5070.3	183	16213	0.6	2855.9	202	6327
15	15	1.4	16080.0	180	49077	5.1	42954.4	9867	104915
Average		**0.3**	**2569.6**	**162.7**	**7657.8**	**3.1**	**4037.8**	**633.9**	**10696.1**
Median		**0.2**	**1501.5**	**61.0**	**5280.5**	**2.2**	**2056.5**	**80.0**	**6158.0**

Table 6. Summary of the results for mate-in-3.

W	**B**	**Succ.**	**t (s)**	**T**	**Min. T**	**Max. T**
6	6	10	48.7	420.3	66	1363
6	7	10	79.2	853.2	60	1905
6	12	8	85.2	1841.6	367	4274
7	6	9	116.4	1775.4	107	2761
7	8	8	77.7	3821.8	615	10676
7	10	9	88.6	5442.6	2336	11630
8	7	10	83.2	3116.4	358	9324
8	7	9	79.3	4460.4	930	10851
8	7	10	121.1	3187.0	15	7346
8	9	9	88.9	6407.8	163	21029
8	12	8	123.0	2846.3	154	7158
10	10	10	148.1	4812.3	671	9822
10	11	9	111.8	10927.9	89	23423
11	9	10	87.8	7662.8	999	19410
11	10	7	153.8	5664.6	1406	9355
11	11	9	63.5	6774.3	271	19825
11	12	8	125.6	12871.6	1767	31553
12	12	10	91.8	8387.2	807	25669
12	13	7	155.1	14448.6	4241	23367
12	13	8	153.1	17519.4	157	36601
Average		**8.9**	**104.1**	**6162.1**	**779.0**	**14367.1**
Median		**9.0**	**90.4**	**5127.4**	**362.5**	**10763.5**

report detailed results for mate-in-4.

We remark that our proposed strategy generates a great variety of puzzles. Less than 0.05% of the compositions were repeated. Examples of generated puzzles are shown in Figure 1. We also remark that each puzzle is a candidate for improvement using methods from [Fainshtein and HaCohen-Kerner 2006], for example.

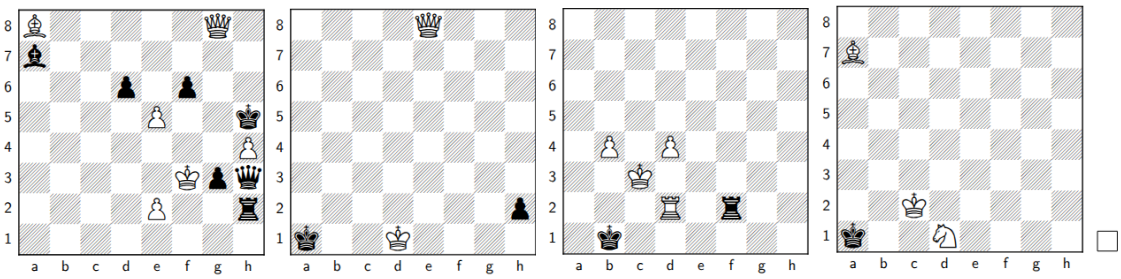

Figure 1. Examples of generated puzzles, N=1, 2, 3, 4 (from left to right)

6. Concluding remarks and future work

The automatic generation of chess mate-in-N puzzles is an interesting computational challenge due to the size of the search space. Traditionally, it has typically been explored by relying on *a priori* knowledge of existing problems and patterns. We investigate the composition of mate-in-N puzzles by generating random chess positions.

Despite the vastness of the search space, we show experimentally that exploring random positions is a viable method to build mate-in-N puzzles. Specially, we generated puzzles with N up to three, which is the same limit found in the literature for more sophisticated methods. To the best of our knowledge, this is the first time such an exploratory investigation of the search space has been performed for mate-in-N problems.

We have identified the limit of this strategy to be $N = 4$, for which the success rates drop significantly. As future work, there is the possibility of incorporating heuristics for piece positioning to enhance the success rate for the cases with $N = 3$ and $N = 4$. We also believe that this generation could be used as prior knowledge for the methods already present in the literature, eliminating the need to have large databases of games or positions.

References

Akl, S. G. and Newborn, M. M. (1977). The principal continuation and the killer heuristic. In *Proceedings of the 1977 annual conference*, pages 466–473.

Bajarani, I. (2024). *720 Exercícios de Xadrez: Mate em 4, Nível Especialista*. Independently published.

Björkqvist, S. (2024). Estimating the puzzlingness of chess puzzles. In *2024 IEEE International Conference on Big Data (BigData)*, pages 8370–8376. IEEE.

Fainshtein, F. and HaCohen-Kerner, Y. (2006). A chess composer of two-move mate problems. *ICGA Journal*, 29(1):33–40.

Feng, X., Veeriah, V., Chiam, M., Dennis, M., Pachauri, R., Tumiel, T., Barbero, F., Obando-Ceron, J., Shi, J., Singh, S., et al. (2025). Generating creative chess puzzles. *arXiv preprint arXiv:2510.23881*.

Gillogly, J. J. (1972). The technology chess program. *Artificial Intelligence*, 3:145–163.

Gourion, D. (2022). An upper bound for the number of chess diagrams without promotion. *ICGA Journal*, 44(2):44–55.

Iqbal, A. (2011). Increasing efficiency and quality in automatic composition of three-move mate problems. *International Journal of Computer Games Technology*, 2011:1–10.

Iqbal, A. (2021). A computational method of optimizing chess compositions to enhance aesthetic appeal. In *2021 2nd International Conference on Artificial Intelligence and Data Sciences (AiDAS)*, pages 1–6. IEEE.

Knuth, D. E. and Moore, R. W. (1975). An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326.

Kocsis, L., Uiterwijk, J. W., Postma, E., and van den Herik, J. (2002). The neural movemap heuristic in chess. In *International Conference on Computers and Games*, pages 154–170. Springer.

Maharaj, S., Polson, N., and Turk, A. (2022). Chess ai: competing paradigms for machine intelligence. *Entropy*, 24(4):550.

Oliveira, T. A. d. (2024). Sudoku: um estudo sobre algoritmos de geração, classificação e resolução para aplicação educacional. Master's thesis, Universidade Federal do Rio Grande do Sul (UFRGS).

Patel, M., Pandey, H., Wagh, T., Hujare, A. D., and Dangi, R. (2022). Vecma: An advance chess engine. In *2022 IEEE Pune Section International Conference (PuneCon)*, pages 1–6. IEEE.

Polgár, L. (2006). *Chess: 5334 Problems, Combinations and Games*. Black Dog & Leventhal, New York.

Santos, R. d. S. d. (2022). Geração procedural de puzzles para o desenvolvimento de jogos. Master's thesis, UFRN.

Schaeffer, J. (1989). The history heuristic and alpha-beta search enhancements in practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(11):1203–1212.

Schlosser, M. (1988). Computers and chess-problem composition. *ICGA Journal*, 11(4):151–155.

Shannon, C. E. (1950). Programming a computer for playing chess. *Philosophical Magazine*, 41(314):256–275.

Steinerberger, S. (2015). On the number of positions in chess without promotion. *International Journal of Game Theory*, 44(3):761–767.

Upasani, N., Gaikwad, A., Patel, A., Modani, N., Bijamwar, P., and Patil, S. (2021). Dev-zero: A chess engine. In *2021 International Conference on Communication information and Computing Technology (ICCICT)*, pages 1–6. IEEE.

von Neumann, J. and Morgenstern, O. (1944). *Theory of Games and Economic Behavior*. Princeton University Press.

Yannakakis, G. N. and Togelius, J. (2018). *Artificial Intelligence and Games*. Springer.