Synthesizing Chess Tactics from Player Games

Abhijeet Krishnan, Chris Martens

North Carolina State University
Venture IV, 1730 Varsity Dr,
Raleigh, NC 27606
akrish13@ncsu.edu, martens@csc.ncsu.edu

Abstract

Competitive games admit a wide variety of player strategies and emergent, domain-specific concepts that are not obvious from an examination of their rules. Expert agents trained on these games demonstrate many useful strategies, but these are difficult for human players to understand and adopt. Algorithmically revealing these strategies could help players develop a better model for making decisions that lead to victories. This paper presents a method for the automatic discovery of player-oriented strategies for chess. We present a formal model for chess strategies, inspired by documented chess tactics, that uses first-order logic clauses for representation. Our system uses inductive logic programming to learn human-interpretable strategies for playing chess in the form of our tactic model. Given minimal background knowledge and training data drawn from real games, our system is able to learn tactics that generalize to a large number of positions. We show that these tactics cover a large number of real-world positions and produce moves that outperform a random player.

Introduction

Recent advancements in reinforcement learning (RL) have produced agents capable of competing with and even outperforming the best human experts at various games like chess (Silver et al. 2018), Go (Silver et al. 2016), Shogi (Li et al. 2020), Mahjong (Silver et al. 2018), *StarCraft II* (Vinyals et al. 2019) and *Dota 2* (Berner et al. 2019). These agents do not simply take advantage of faster reaction times and calculation abilities, but are actually employing new, better strategies that lead to more victories (DeepMind 2019). Borrowing from Jeanette Wing's definition of computational thinking (Wing 2008), these agents appear to have better *abstractions* than human experts for the games they're trained to play.

Despite the existence of such agents in various competitive games, we still see human competition continue to thrive, with these agents leading to new ways of thinking and a re-evaluation of long-held beliefs about the game (Nelson 2019). These discoveries have, so far, involved manual or engine-assisted analysis of the games played by the agents (Sadler and Regan 2019; Zhou 2018). If these agents could explain their strategies and decision-making to human players, we posit that it would help improve their (the human players') play.

Such chess-playing agents (chess engines) are used extensively in game analysis (Smith 2004; Tukmakov 2020) and tournament preparation (Andrei 2021). Expert chess players utilize engine move suggestions and evaluations to analyze new lines to play (PTI 2016). Most current engines use a neural network-based model with many thousands of parameters trained using deep reinforcement learning (DRL) in conjunction with a search algorithm to produce game moves. Examples include Monte Carlo Tree Search in *AlphaZero* (Silver et al. 2016), Predictor+Upper Confidence Bound tree search in *Leela Chess Zero* (Pascutto, Gian-Carlo and Linscott, Gary 2019) or alpha-beta pruning in *Stockfish 14* (Romstad, Costalba, and Kiiski 2021). However, this is very different to how human chess players employ pattern-recognition to produce moves (de Groot 1946; Connors, Burns, and Campitelli 2011). Existing solutions to explaining chess moves use extensive domain knowledge (DecodeChess 2022) or do not adequately explain individual moves.

In this work, we present a model for a human-readable chess strategy, inspired by documented chess tactics, for playing chess. We also present a system that integrates a modified inductive logic programming (ILP) engine with novel metrics for learning these tactics. Our system can extrapolate chess tactics from positions drawn from play traces of human vs. human games. It requires no more domain knowledge than the base rules of chess. We show that the learned tactics generalize well to real-world chess positions and produce moves that outperform a random player. We discuss some limitations of our approach and conclude with directions for future work.

Related Work

Strategy Synthesis A number of works attempt to learn rule-based agents using evolutionary approaches to play role-playing games like *Neverwinter Nights* (Spronck, Sprinkhuizen-Kuyper, and Postma 2004), board games like Checkers and Reversi (Benbassat and Sipper 2011), cooperative games like Hanabi (Canaan et al. 2018), platformers like Mario (de Freitas, de Souza, and Bernardino 2018), and real-time strategy games like μRTS (Mariño et al. 2021). Partially-applicable strategies for puzzle games have been

learned using constraint satisfaction (Butler, Torlak, and Popović 2017). Our model for chess tactics is learned using ILP, and incorporates domain knowledge of the concept of a tactic in order to improve interpretability.

Chess Pattern Learning Chess has been called the *drosophilia*[1] of artificial intelligence (McCarthy 1990). It has been a mainstay of AI research from the invention of the digital computer (Claude 1950) to the neural network revolution (Silver et al. 2018). Given the depth of experimentation with AI techniques for chess, it is not surprising that the idea of using patterns to guide a computer to play chess is not new. Patterns have been used to suggest moves and guide playing strategies in middle-game positions (Berliner 1975; Pitrat 1977; Wilkins 1979) and endgames (Huberman 1968; Bramer 1977; Bratko 1982). Levinson and Snyder (1991) used weighted patterns in their Morph system as an evaluation function to guide playing strategy. Very recent work has attempted to directly probe neural network engines to test for the presence of human concepts (McGrath et al. 2021). Morales (1992) developed the PAL system to learn first-order patterns in chess using ILP. We build upon this work by taking advantage of modern chess engines to serve as the reference evaluation function to select learned patterns instead of hand-crafted heuristics.

Background

Chess Tactics

Tactics in chess are maneuvers that take advantage of short-term opportunities (Seirawan 2005, Chapter 1). They are a move or series of moves that bring about an immediate advantage for the player. They are identified by *motifs*—positional patterns that indicate whether a tactic might exist in the given position. An example of a tactic is the *fork*, where a piece simultaneously attacks two opponent pieces at once (see Figure 1).

Inductive Logic Programming

Inductive logic programming (ILP) is a form of symbolic machine learning where the goal is to induce a hypothesis (a set of logical rules) that generalises given training examples. It can learn human-readable hypotheses from smaller amounts of data than neural network-based models (Cropper and Dumančić 2020).

An ILP problem is specified by three sets of Horn clauses—

- B, the background knowledge,
- E^+, the set of positive examples of the concept, and
- E^-, the set of negative examples of the concept.

The ILP problem is to induce a hypothesis $H \in \mathcal{H}$ (an appropriately chosen hypothesis space) that, in combination with the background knowledge, entails all the positive examples and none of the negative examples. Formally, this

[1]fruit fly; easily bred and thus extensively used in genetics research

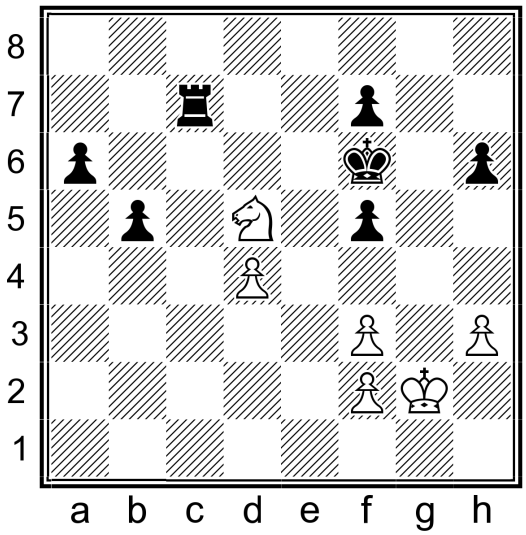

Figure 1: An example of a fork from a Lichess puzzle[2]. The white knight on d5 simultaneously checks the opponent king on f6 and attacks the rook on c7.

can be written as -

$$\forall e \in E^+, H \cup B \models e \text{ (i.e., } H \text{ is } \textit{complete)}$$

$$\forall e \in E^-, H \cup B \not\models e \text{ (i.e., } H \text{ is } \textit{consistent)}$$

To make the ILP problem more concrete, we provide a toy example.

E^+ and E^- contain positive and negative examples of the target `knight_move` relation respectively. B contains background knowledge i.e., clauses which might be useful in inducing a hypothesis for `knight_move`.

$$E^+ = \left\{ \begin{array}{l} \texttt{knight\_move(d4,c6).} \\ \texttt{knight\_move(d4,e6).} \\ \texttt{knight\_move(d4,b5).} \\ \texttt{knight\_move(d4,f5).} \end{array} \right\}$$

$$E^- = \left\{ \begin{array}{l} \texttt{knight\_move(d4,d5).} \\ \texttt{knight\_move(d4,b6).} \\ \texttt{knight\_move(d4,e1).} \\ \texttt{knight\_move(d4,h7).} \end{array} \right\}$$

$$B = \left\{ \begin{array}{l} \texttt{l\_move(d4,c6).} \\ \texttt{l\_move(d4,e6).} \\ \texttt{l\_move(d4,b5).} \\ \texttt{l\_move(d4,f5).} \end{array} \right\}$$

From this information, we could induce a hypothesis for `knight_move` as -

```
knight_move(From,To) :- l_move(From,To).
```

Popper

Popper is an ILP system that implements an approach called *learning from failures* (Cropper and Morel 2021). It operates

[2]https://lichess.org/training/4pEpj

in three stages: generate, test and constrain. Given an ILP problem, it generates a candidate solution, tests it against examples in the training set, and formulates constraints based on the outcome of failed examples to cut down the solution space. These three stages are repeated until a solution is found. Popper also allows for specifying hypothesis biases to further constrain the solution space based on domain knowledge of the problem being solved.

Popper uses two general types of constraints - *generalization* and *specialization*. Assume an ILP problem $\langle E^+, E^-, B\rangle$ and a generated hypothesis H. If H entails a negative example, then H is too general and so we can prune generalisations of it. Similarly, if H does not entail a positive example, it is too specific, and we can prune specialisations of it. The notions of the generalisation and specialisation of a hypothesis are defined in terms of *clausal subsumption*, and we refer readers to the original paper for formal definitions of these terms. Popper supports providing a *language bias* to influence the hypothesis space.

Methodology

In this section, we formalize the learning problem of *strategy synthesis* for games, and describe the problem of learning chess tactics as an instance of strategy synthesis for chess. We describe our formal model for chess tactics using first-order logic, and describe our method for learning them using ILP. Finally, we describe the metrics that we use to measure a tactic's utility as part of our learning method.

Strategy Synthesis

Let us be given a game environment $\mathcal{G}$ modeled as a finite, episodic Markov decision process $\langle \mathcal{S}, \mathcal{A}(s), \mathcal{P}, \mathcal{R}, \gamma\rangle$, where —

- $\mathcal{S}$ is the finite set of legal game states reachable from the start state s_0,
- $\mathcal{A}(s)$ is the set of legal actions that can be taken in a state $s \in \mathcal{S}$,
- $\mathcal{P}$ is the state transition function that models the game's dynamics,
- $\mathcal{R}$ is the reward function describing the game's win conditions, and
- γ is a discount factor between $[0, 1]$

We define a *strategy* ς (sigma) for $\mathcal{G}$ at timestep t as a probability distribution over the available actions in a state, for a *subset* of states in the state space. The states for which ς is defined are termed as the states for which the strategy is *applicable*, and is given by the *applicability function* $\mathrm{A}_\varsigma : \mathcal{S} \to \{0, 1\}$.

$$\varsigma(a|s) \doteq \mathbb{P}[A_t = a|S_t = s], \forall s \in \mathrm{A}_\varsigma, a \in \mathcal{A}(s) \quad (1)$$

The definition of a strategy in Equation 1 borrows from the notion of a *policy* as used in RL (Sutton and Barto 2018) as well as game-theoretic notions of a strategy (Boros et al. 2012). Intuitively, a strategy describes a pattern or feature of a game state that comes up often in regular game play and that tends to influence the actions taken in states with that feature.

```
tactic(Position, From, To) ←
        feature_1(···),
        feature_2(···),
        ⋮
        feature_n(···)
```

Figure 2: Our tactic model expressed in Prolog pseudocode. Every `feature_i` clause is a rule defined in the predicate vocabulary.

Strategy Synthesis for Chess

Our strategy model for chess uses the concept of a *chess tactic*. Formally, we define our strategy model for chess (hereafter referred to as a tactic) as a first-order logic rule expressed in Prolog using a particular predicate vocabulary. As seen in Figure 2, the rule head of the tactic consists of the variables `Position`, `From` and `To`. `Position` describes the input state, also expressed in first-order logic using our predicate vocabulary. `From` and `To` describe the output action, namely, the move which begins from the square `From` and ends on the square `To`. This replicates the long algebraic notation for moves used in the Universal Chess Interface (UCI) protocol, an open communication protocol allowing chess engines to interact with user interfaces (Kahlen 2004).

If the tactic is provided as a query to the Prolog interpreter with a grounded instance of `Position` and non-ground move variables, it will attempt to *unify* the latter with ground moves (Sterling and Shapiro 1994). The variable binding(s) (i.e., moves) it finds are treated as the action distribution defined on the state, with every found move being equi-probable, and every other move accorded a probability of 0.

Our usage of first-order logic to model chess tactics is motivated by the following reasons —

1. Chess tactics are an important concept that human players use to think about chess (Szabo 1984) and are useful in chess education (Gobet and Jansen 2006).
2. First-order logic has been extensively used to model chess patterns (ref. Related Work).
3. Logic rules are commonly acknowledged to be interpretable and have a long history of research (Zhang et al. 2021).

ILP for Tactic Learning

Given our tactic model being a first-order logic rule, we model the problem of learning it as an ILP problem. We use training examples drawn from real games played by humans online. Each training example is a $\langle \text{position}, \text{move}\rangle$ tuple, where position is the board state converted to first-order logic using a hand-engineered predicate vocabulary, and move is the move made in that position, also represented in first-order logic. We define this predicate vocabulary in the background knowledge, along with predicates representing the board state and relationships between squares. Our choice of predicates is motivated by the ability to use them to express tactics from chess literature, like the pin or the

fork. We also introduce a foreign predicate implemented externally to represent a legal move. See Figure 3 for an expression of the *fork* tactic from chess literature in this model. Our background knowledge design borrows from that of the PAL system (Morales 1992). We refer readers to the Appendix for the complete list of predicates in our background knowledge.

Given this formulation of the tactic-learning problem as an ILP problem, we select Popper as the ILP system to learn tactics with. Popper searches for the hypothesis that maximises the F1 score when evaluated against the examples. However, we wish to learn *multiple* tactics that might not cover the entire example set. Our proposed system supports learning multiple tactics. We do so by modifying the *generate* and *constrain* stages of the Popper ILP system in the following ways —

- *generate*: modified to only generate tactics that produce legal moves
- *constrain*: prevent further *specializations* of a tactic that does not match any position in the training set from being generated

```
fork(Position,From,To) ←
    make_move(From, To, Position, NewPosition),
    attacks(To, Square1, NewPosition),
    attacks(To, Square2, NewPosition),
    different_pos(Square1, Square2).
```

Figure 3: An interpretation of the *fork* tactic from the chess literature using our predicate vocabulary. The first `attacks` clause states that the piece at To attacks the opposing piece at Square1 in the current position.

Metrics

The tactics learned by our system are merely guaranteed to match with at least one position in the training set. We need additional signals that tell us how generally applicable a tactic might be, and how useful the moves it produces are. To do so, we introduce two metrics — *coverage* and *divergence*.

Coverage A tactic ς's coverage for a set of positions P is calculated as —

$$P_{\mathrm{A}} \doteq P \cap \mathrm{A}_{\varsigma} \tag{2}$$

$$\text{Coverage}(\varsigma, P) \doteq \frac{|P_{\mathrm{A}}|}{|P|} \tag{3}$$

Coverage is the fraction of positions in a given set to which the tactic is *applicable*. If P is representative of positions encountered in games, coverage is a useful measure of how likely it is that the tactic can be used to make moves in games. A low coverage value indicates that the tactic is *situational*, whereas a high coverage value indicates that the tactic is general.

Divergence To measure the quality of moves suggested by a tactic, we extend a metric previously used to analyse world chess champions (Guid and Bratko 2006, 2011; Romero 2019). A chess engine E usually provides a position evaluation function $v_E(s)$. From this, we can obtain a move evaluation function $q_E(s, a)$ as follows —

$$q_E(s,a) = \sum_{s',r} \mathcal{P}(s', r|s, a)[r + v_E(s')] \tag{4}$$

$$= v_E(s'), s' \text{ is non-terminal} \tag{5}$$

Equation 5 follows from 4 since rewards in chess are 0 for non-terminal states, $\gamma = 1$, and chess rules are deterministic. A chess engine could evaluate a position to be a 'Mate in X' rather than a numerical score. In this case, we assign an arbitrary large value to the evaluation.

Given two moves a_1, a_2 made in a position s, we can calculate their difference $d_E(s, a_1, a_2)$ as —

$$d_E(s, a_1, a_2) \doteq \mid q_E(s, a_1) - q_E(s, a_2) \mid \tag{6}$$

We can now define the *divergence* of a tactic from a set of examples P as the average difference between the moves suggested by the tactic and the ground truth move —

$$\text{Divergence}_E(\varsigma(\cdot), P) \doteq$$

$$\frac{1}{|P_{\mathrm{A}}|} \sum_{(s,a_1)\in P_{\mathrm{A}}} \sum_{a_2 \in \mathcal{A}(s)} \varsigma(a_2|s) d_E(s, a_1, a_2) \tag{7}$$

Divergence of a tactic from the ground truth is low and close to 0 when its suggestions are similar in engine evaluation to the ground truth moves, and takes on large values when it differs significantly. Divergence is a useful metric to measure how closely a tactic approximates a reference policy.

Evaluation

In this section, we describe the procedure used to evaluate the tactics produced by our learning system. We describe our testing and training datasets, how we generate the tactics used for evaluation, and the metrics we use to measure the performance of the learned tactics.

Dataset

To source our $\langle$position, move$\rangle$ training examples, we use a collection of games played by human players on the Internet chess server Lichess[3]. Specifically, we use the January 2013 archive of standard (played with regular chess rules) rated (users stand to gain or lose rating points based on the outcome) games played on the website (lichess.org 2021). The archive consists of 121,332 games, with players of ELO rating 1601 ± 289 (new players start at 1500 (Lichess 2021)). To generate N examples, we randomly sample N games and select a single random position from that game, along with the move made in that position. We select positions beginning from move 12 (Guid and Bratko 2006; Romero

[3]https://lichess.org/

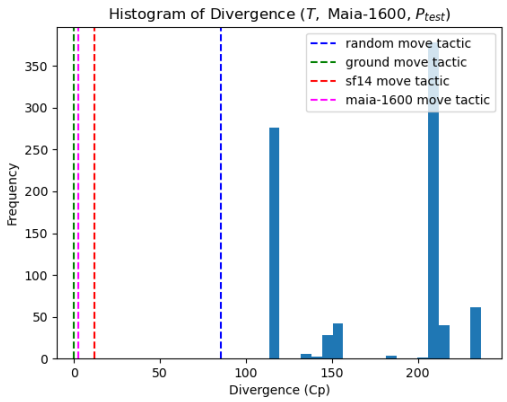

Figure 4: Divergence histogram for T evaluated using Maia-1600

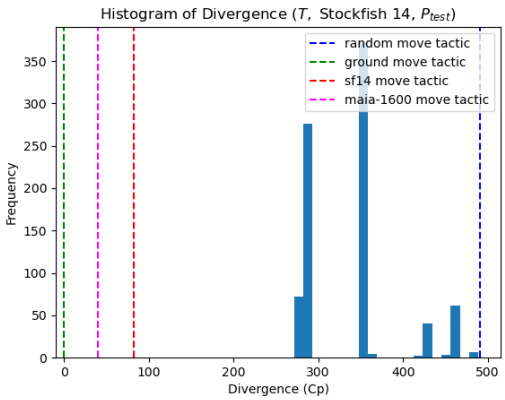

Figure 5: Divergence histogram for T evaluated using Stockfish 14

2019), and exclude games which did not end normally or by time forfeit. Using this procedure, we generated a dataset of 100 examples which were split into training/validation in a 90:10 ratio. We found that using more training examples did not produce new tactics since the hypothesis space was exhausted. For testing, we use the February 2013 archive of 123,961 games and ELO rating 1595 ± 298 to generate 10 testing examples.

Training

Using the bias settings provided by Popper, we limit the size of the learnable hypotheses to a maximum of one clause, five variables and five body literals. Empirically, we find that this strikes a good balance between learning time and quality of learned tactics. We run our proposed method until no more solutions are found. We obtain a list T of 837 tactics.

Performance Metrics

To measure the performance of our tactics on the test data, we use the metrics of *accuracy*, the percentage of moves suggested by a tactic that matches the move in the test data, and *coverage* and *divergence*, which are defined in Equations 3 and 7 respectively. To provide the evaluation function for calculating divergence, we use the engines Maia-1600 and Stockfish 14. The Maia-1600 engine has been trained on games played by players of ELO rating between $[1600 - 1699]$ and has been shown to resemble human moves (McIlroy-Young et al. 2020). Stockfish 14 is the winner of the TCEC 2020 Championship (Haworth and Hernandez 2021) and is an extremely strong engine. We limit the search depth of both engines to 1-ply to be comparable to our tactic model. We report divergence using the unit of *centipawns* (Cp), a widely accepted unit of measure used in chess as a measure of advantage, and defined as $1/100$ of a pawn (Guid and Bratko 2017).

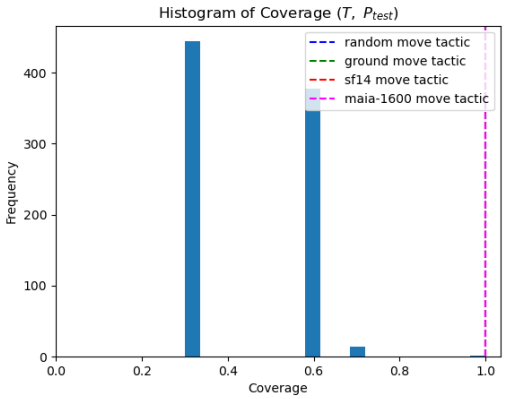

Figure 6: Coverage histogram for T

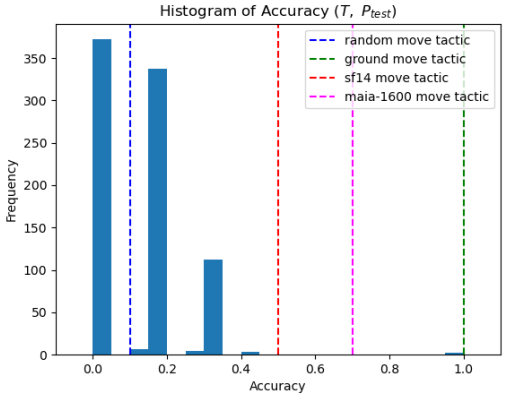

Figure 7: Accuracy histogram for T

Results and Analysis

Based on the data obtained, we report the results for each of accuracy, coverage and divergence as a histogram with

20 buckets. Since we do not have a reference set of existing tactics expressed in our predicate vocabulary to compare against, we choose to compare against the following baseline tactics —

- **random move tactic**: makes a random legal move in a given position, and is applicable to all positions
- **ground move tactic**: replicates the move made in the ground truth example, and is applicable to all positions
- **engine move tactic**: makes the best move suggested by an engine, and is applicable to all positions. We use two engines - Stockfish 14 and Maia-1600.

From the coverage values in Figure 6, we see that all the tactics learned by our system cover 30% - 60% of the test set. We conclude that the tactics learned by our system are moderately likely to be applicable to positions that arise in real games.

From the accuracy histogram in Figure 7, we see the tactics learned by our system are only marginally more accurate at predicting moves in the test set than the random baseline. Overall, the highest accuracy of a learned tactic in T is 42% compared to the 10% accuracy of the random baseline.

From the divergence histogram for T calculated using Maia-1600 (Figure 4), we see that Maia-1600 evaluates the learned tactics as having *greater* divergence from ground that the random baseline. However, from Figure 5, we see that the stronger Stockfish 14 evaluates the same tactics as having *lower* divergence from ground. We can conclude that the learned tactics are better approximating Stockfish 14's policy as compared to Maia-1600's policy.

Qualitative Analysis

We present three tactics learned by our system to further analyze qualitatively.

```
f(Position,From,To) ←
    legal_move(From, To, Position),
    attacks(From, Square1, Position),
    behind(To, Square1, Square2, Position),
    different_pos(Square1, Square2).
```

Figure 8: A learned tactic with low divergence from ground as evaluated by Stockfish 14.

Lowest Divergence The tactic in Figure 8 was chosen from those having the least divergence from ground as measured by Stockfish 14. It can be interpreted as follows - if one of your pieces is attacking your opponent's piece (`attacks`(From, Square1, Position)), move it instead to a square which puts it in line with two of your opponent's pieces (`behind`(To, Square1, Square2, Position)). To the author's knowledge, this does not represent an existing tactical idea in chess.

Highest Accuracy The tactic in Figure 9 was chosen from that having the highest accuracy. It can be interpreted as follows - if one of your pieces is attacking your opponent's piece (`attacks`(From, Square1, Position)), move it instead to different square. This somewhat resembles the idea of a tactical retreat in chess. It can be thought of as a more general tactic than that in 8.

```
f(Position,From,To) ←
    legal_move(From, To, Position),
    attacks(From, Square1, Position),
    different_pos(From, To),
    different_pos(From, Square1).
```

Figure 9: A learned tactic with the highest accuracy.

```
f(Position,From,To) ←
    legal_move(From, To, Position),
    behind(From, To, Square1, Position),
    behind(From, Square2, Square3, Position),
    behind(From, Square3, Square2, Position),
    behind(From, Square2, To, Position).
```

Figure 10: A learned tactic with meaningless variable reshuffling.

Variable Reshuffling The tactic in Figure 10 represents a possibility in the tactic hypothesis space that is merely a permutation of the variables in the rule. It is difficult to interpret this in terms of chess, and is very likely meaningless. It points to opportunities to restrict the hypothesis space using language biases to prevent such tactics from being generated.

Discussion and Future Work

Since our system uses ILP as its learning method, it inherits many benefits and challenges associated with ILP. Our learning system requires carefully selected language biases in the form of background knowledge in order to learn efficiently (Cropper and Dumančić 2020). Designing predicates suitable to game-like domains will make searching for better tactics more efficient. The learned tactics are also interpretable (Muggleton et al. 2018), and can be added back into the background knowledge to allow lifelong learning (Cropper and Tourret 2020). However, work needs to be done in this area in the context of games.

Our tactics are able to beat a random baseline in producing moves similar to a beginner player. However, they still have high divergence from player moves, have low accuracy and do not suggest the same moves, indicating that they do not yet adequately represent a human beginner player. Improving the learning algorithm to penalize tactics which do not match the ground truth moves could lead to better results.

Given that our tactics serve as an interpretable model of human players, they could also be used to interpret chess engines, and serve as a general technique for explainable AI. However, further work is required to learn tactics of playing strength comparable to top engines.

Conclusion

We have presented the problem of learning chess tactics in the form of first-order logic rules from examples. We have

presented a learning method that uses ILP to learn these tactics. We have evaluated this system and shown that the tactics learned cover a large portion of the test set. Using the metrics of divergence, we showed that they can approximate a human beginner better than a random baseline. We discussed limitations of our method and concluded that while it is promising, further work is required to learn tactics of acceptable playing strength.

References

Andrei, M. 2021. A supercomputer helped set up the World Chess Championship game. Accessed: 2021-10-27.

Benbassat, A.; and Sipper, M. 2011. Evolving board-game players with genetic programming. In *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*, 739–742.

Berliner, H. J. 1975. A representation and some mechanisms for a problem solving chess program. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE.

Berner, C.; Brockman, G.; Chan, B.; Cheung, V.; Dębiak, P.; Dennison, C.; Farhi, D.; Fischer, Q.; Hashme, S.; Hesse, C.; et al. 2019. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*.

Boros, E.; Elbassioni, K.; Gurvich, V.; and Makino, K. 2012. On Nash equilibria and improvement cycles in pure positional strategies for Chess-like and Backgammon-like n-person games. *Discrete Mathematics*, 312(4): 772–788.

Bramer, M. A. 1977. *Representation of Knowledge for Chess Endgames Towards a Self-Improving System*. Ph.D. thesis, Open University (United Kingdom).

Bratko, I. 1982. Knowledge-based problem-solving in AL3. *Machine intelligence*, 10: 73–100.

Butler, E.; Torlak, E.; and Popović, Z. 2017. Synthesizing interpretable strategies for solving puzzle games. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*, 1–10.

Canaan, R.; Shen, H.; Torrado, R.; Togelius, J.; Nealen, A.; and Menzel, S. 2018. Evolving agents for the hanabi 2018 cig competition. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, 1–8. IEEE.

Claude, E. S. 1950. Programming a Computer for Playing Chess. *Philosophical Magazine, Ser*, 7(41): 314.

Connors, M. H.; Burns, B. D.; and Campitelli, G. 2011. Expertise in complex decision making: the role of search in chess 70 years after de Groot. *Cognitive science*, 35(8): 1567–1579.

Cropper, A.; and Dumančić, S. 2020. Inductive logic programming at 30: a new introduction. *arXiv preprint arXiv:2008.07912*.

Cropper, A.; and Morel, R. 2021. Learning programs by learning from failures. *Machine Learning*, 110(4): 801–856.

Cropper, A.; and Tourret, S. 2020. Logical reduction of metarules. *Machine Learning*, 109(7): 1323–1369.

de Freitas, J. M.; de Souza, F. R.; and Bernardino, H. S. 2018. Evolving Controllers for Mario AI Using Grammar-based Genetic Programming. In *2018 IEEE Congress on Evolutionary Computation (CEC)*, 1–8. IEEE.

de Groot, A. D. 1946. *Het denken van den schaker: een experimenteel-psychologische studie*. Noord-Hollandsche Uitgevers Maatschappij Amsterdam.

DecodeChess. 2022. Chess Analysis, Powered by AI. DecodeChess. https://decodechess.com/ (Accessed: 2022-05-27).

DeepMind. 2019. AlphaStar: Mastering the real-time strategy game StarCraft II. Accessed: 2022-09-21.

Gobet, F.; and Jansen, P. J. 2006. Training in chess: A scientific approach. *Education and chess*.

Guid, M.; and Bratko, I. 2006. Computer analysis of world chess champions. *ICGA journal*, 29(2): 65–73.

Guid, M.; and Bratko, I. 2011. Using heuristic-search based engines for estimating human skill at chess. *ICGA journal*, 34(2): 71–81.

Guid, M.; and Bratko, I. 2017. Influence of search depth on position evaluation. In *Advances in computer games*, 115–126. Springer.

Haworth, G.; and Hernandez, N. 2021. The 20th Top Chess Engine Championship, TCEC20. *J. Int. Comput. Games Assoc.*, 43(1): 62–73.

Huberman, B. J. 1968. *A program to play chess end games*. 65. Department of Computer Science, Stanford University.

Kahlen, S.-M. 2004. UCI protocol. Accessed: 2022-09-21.

Levinson, R.; and Snyder, R. 1991. Adaptive pattern-oriented chess. In *Machine Learning Proceedings 1991*, 85–89. Elsevier.

Li, J.; Koyamada, S.; Ye, Q.; Liu, G.; Wang, C.; Yang, R.; Zhao, L.; Qin, T.; Liu, T.-Y.; and Hon, H.-W. 2020. Suphx: Mastering mahjong with deep reinforcement learning. *arXiv preprint arXiv:2003.13590*.

Lichess. 2021. Chess rating systems. Accessed: 2022-09-23.

lichess.org. 2021. lichess.org open database. https://database.lichess.org/. Accessed: 2021-10-27.

Mariño, J. R.; Moraes, R. O.; Oliveira, T. C.; Toledo, C.; and Lelis, L. H. 2021. Programmatic Strategies for Real-Time Strategy Games. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, 381–389.

McCarthy, J. 1990. Chess as the Drosophila of AI. In *Computers, chess, and cognition*, 227–237. Springer.

McGrath, T.; Kapishnikov, A.; Tomašev, N.; Pearce, A.; Hassabis, D.; Kim, B.; Paquet, U.; and Kramnik, V. 2021. Acquisition of Chess Knowledge in AlphaZero. *arXiv:2111.09259 [cs, stat]*. ArXiv: 2111.09259.

McIlroy-Young, R.; Sen, S.; Kleinberg, J.; and Anderson, A. 2020. Aligning Superhuman AI with Human Behavior: Chess as a Model System. In *Proceedings of the 25th ACM SIGKDD international conference on Knowledge discovery and data mining*.

Morales, E. 1992. *First order induction of patterns in Chess*. Ph.D. thesis, PhD thesis, The Turing Institute-University of Strathclyde.

Muggleton, S. H.; Schmid, U.; Zeller, C.; Tamaddoni-Nezhad, A.; and Besold, T. 2018. Ultra-strong machine learning: comprehensibility of programs learned with ILP. *Machine Learning*, 107(7): 1119–1140.

Nelson, P. H. 2019. When Magnus met AlphaZero. *New In Chess*, 2019(8): 2–10.

Pascutto, Gian-Carlo and Linscott, Gary. 2019. Leela Chess Zero (v0.21.0).

Pitrat, J. 1977. A chess combination program which uses plans. *Artificial Intelligence*, 8(3): 275–321.

PTI. 2016. World Chess Championship: Role of the 'seconds'.

Romero, O. 2019. Computer analysis of world chess championship players. *ICSEA 2019*, 212.

Romstad, T.; Costalba, M.; and Kiiski, J. 2021. Stockfish 14.

Sadler, M.; and Regan, N. 2019. Game Changer. *AlphaZero's Groundbreaking Chess Strategies and the Promise of AI. Alkmaar. The Netherlands. New in Chess*.

Seirawan, Y. 2005. *Winning chess tactics*. Everyman Chess.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; Van Den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; et al. 2016. Mastering the game of Go with deep neural networks and tree search. *nature*, 529(7587): 484–489.

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; et al. 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419): 1140–1144.

Smith, R. 2004. *Modern Chess Analysis*. Gambit. ISBN 9781904600084.

Spronck, P.; Sprinkhuizen-Kuyper, I.; and Postma, E. 2004. Online adaptation of game opponent AI with dynamic scripting. *International Journal of Intelligent Games and Simulation*, 3(1): 45–53.

Sterling, L.; and Shapiro, E. 1994. *The Art of Prolog: Advanced Programming Techniques*, 87–90. MIT Press, 2nd edition.

Sutton, R. S.; and Barto, A. G. 2018. *Reinforcement learning: An introduction*. MIT press.

Szabo, A. 1984. *Computer chess tactics and strategy*. Ph.D. thesis, University of British Columbia.

Tukmakov, V. 2020. *Modern Chess Formula - The Powerful Impact of Engines*. Thinkers Publishing. ISBN 9789492510815.

Vinyals, O.; Babuschkin, I.; Czarnecki, W. M.; Mathieu, M.; Dudzik, A.; Chung, J.; Choi, D. H.; Powell, R.; Ewalds, T.; Georgiev, P.; et al. 2019. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, 575(7782): 350–354.

Wilkins, D. E. 1979. *Using patterns and plans to solve problems and control search*. Stanford University.

Wing, J. M. 2008. Computational thinking and thinking about computing. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 366(1881): 3717–3725.

Zhang, Y.; Tiňo, P.; Leonardis, A.; and Tang, K. 2021. A survey on neural network interpretability. *IEEE Transactions on Emerging Topics in Computational Intelligence*.

Zhou, Y. 2018. Rethinking Opening Strategy: AlphaGo's Impact on Pro Play. *CreateSpace*, 1(36): 212.

Appendix

Background Knowledge Definitions

This appendix has the background knowledge definitions used by our system.

- `legal_move(From,To,Pos)`: Defines legal moves of chess pieces. Dynamically added to the knowledge base via an external script.
- `square(Rank,File)`: Defines valid rank and file coordinates of chess squares.
- `to_coords(Name,Rank,File)`: Converts a square *Name* to the corresponding rank and file coordinates.
- `sq(Name)`: Defines valid square names.
- `sameRow(X1,Y1,X2,Y2)`: The squares (X_1, Y_1) and (X_2, Y_2) are on the same row.
- `sameCol(X1,Y1,X2,Y2)`: The squares (X_1, Y_1) and (X_2, Y_2) are on the same column.
- `side(Side)`: Defines valid side names.
- `other_side(Side, OtherSide)`: The sides *Side* and *OtherSide* are opposites.
- `piece(Piece)`: Defines valid piece names.
- `sliding_piece(Piece)`: The piece *Piece* moves in a straight line.
- `contents(Side,Piece,X,Y)`: The piece of type *Piece* and colour *Side* is located at the board coordinates (X, Y).
- `move(FromX,FromY,ToX,ToY)`: Defines the move of the piece located at (FromX, FromY) to (ToX, ToY).
- `position(Pos)`: Defines a valid position.
- `turn(Side,Pos)`: Asserts that it is *Side*'s turn in the position *Pos*.
- `kingside_castle(Side, Pos)`: Asserts that *Side* can castle king-side in the position *Pos*.
- `queenside_castle(Side, Pos)`: Asserts that *Side* can castle queen-side in the position *Pos*.
- `attacks(From,To,Pos)`: The piece at square *From* attacks the opposing piece at *To* in the position *Pos*.
- `different_pos(S1, S2)`: The squares S_1 and S_2 are different.
- `piece_at(S,Pos,Side,Piece)`: Asserts that the piece *Piece* of colour *Side* exists at square S in the position *Pos*.

- `behind(Front,Middle,Back,Pos)`: The piece on the square *Front* is in line with the the pieces on squares *Middle* and *Back*.
- `make_move(From,To,Pos,NewPos)`: Performs the actual movement of a piece changing the state description.