Open University of Cyprus

Faculty of Pure and Applied Sciences

Master's Degree *Cognitive Systems*

Master's Thesis

Application of the Machine Coaching Paradigm on Chess Coaching

Vasileios T. Markos

Supervisor
Loizos Michael

December 2020

Open University of Cyprus

Faculty of Pure and Applied Sciences

Master's Degree *Cognitive Systems*

Master's Thesis

Application of the Machine Coaching Paradigm on Chess Coaching

Vasileios T. Markos

Supervisor
Loizos Michael

This master's thesis has been submitted as part of the requirements to obtain a master's degree
in Cognitive Systems
by the Faculty of Pure and Applied Sciences
of the Open University of Cyprus.

December 2020

BLANK PAGE

Abstract

In the past two decades computer chess has overcome human capabilities and efficiency in all aspects of the game. This impressive achievement has been possible, especially during the last decade, due to state-of-the-art Deep Learning methodologies that have been developed. However, since such methods perform like black-boxes, prohibiting any notion of interpretability by human users in the first place, it would be meaningful to explore the possibility of designing an explainable and cognitively efficient chess bot. In this thesis we present an efficient explainable interaction protocol accompanied by a corresponding user interface for computer chess. Moreover, we also present useful feedback from chess experts – professional players as well as chess coaches.

Contents

Chapter 1
Introduction

In this chapter, we will present the problem discussed in this thesis, the methodology we adopted as well as the motivation behind our choices. The structure of this chapter is as follows: (i) in section 1.1 we make a short introduction to the field of automated chess and briefly describe the problem this thesis addresses; (ii) in section 1.2 we delve into more details about the problem itself as well as the motivation behind the selected methodology and; (iii) in section 1.3 we provide a brief outline of the rest of this thesis.

1.1 Setting the problem

During the past decades, Artificial Intelligence (AI) agents have been utilised in several domains, from automated music suggestion – e.g. in several video/music streaming platforms such as youtube, spotify and so on – to self-driving vehicles and smart devices which provide everyday micromanagement suggestions. Regardless of the extent of success of such agents in each domain of application, the level of penetration into most people's everyday routines is astounding, given the short time interval in which AI has become more popular.

Even if AI applications are so widespread in everyday tasks, there were not so many cases in AI's relatively short history that have captured the public's attention to such an extent as G. Kasparov's game series against IBM's Deep and Deep(er) Blue in 1996 and 1997 respectively. Chess had long been considered "the touchstone of intellect", as per the words of Goethe, which led many to believe that designing a machine capable of winning chess against humans would have many philosophical implications for human thought as well as the game itself (Hsu, et al., 1990: 44). Given this highly intellectual – and, hence, attributed only to humans – nature of the game, even Deep Blue's 2-4 defeat by G. Kasparov in 1996 was an alarming result for human primacy in chess: Deep Blue had managed to take a win (in game 1) and two draws (in games 3 and 4) against a running world champion and by many considered one of the best chess players in history. Given its performance, the first win of a chess playing engine against a human seemed closer than ever before. Indeed, in 1997 Deep(er) Blue managed to score a 3½ – 2½ victory against G. Kasparov by winning games 1 and 6 and drawing games 3, 4 and 5, completing a remarkable milestone in AI's history[1] (Campbell, et al., 2002: 57-59).

Given the highly symbolic value of chess, as described by Goethe, Kasparov's 1997 defeat against IBM's Deep(er) Blue is also important for reasons beyond the game itself. Attempts to design Chess machines and engines have been recorded since the middle of the 18th century (Standage, 2002: 18-23) and they reflected, more or less each period's state of the art technologies. As a result, Deep(er) Blue's victory signaled the beginning of a new era for artificial intelligence; one in which AI applications would be capable of supporting and, in many cases, substituting humans, carrying out typically human tasks, oftentimes in a more efficient way.

Indicative of this new era and the re-established relationship between humanity and its own technological devices are contemporary machine learning approaches and their applications in various domains, a technological trend also reflected in chess playing engines. Nowadays, most of the state of the art approaches in computer chess rely on deep neural network methodologies – e.g. AlphaZero (Silver, et al., 2017: 1-3) – and/or alpha-beta pruning or similar methodologies, possibly accompanied by some dedicated hardware – e.g. the Stockfish chess engine (Romstad, Costalba, Kiiski, 2008). As recent experience has indicated, such methodologies are overwhelmingly efficient in terms of outperforming human players. For instance, AlphaZero was capable of scoring a 64-36 victory (28 wins, 72 draws and 0 loses) against Stockfish 7 in 2017 (Silver, et al., 2017: 4-5), while Stockfish – being denied access to its opening books and end-game tables – had recently defeated H. Nakamura[2] by a 3-1 score in 2014 (2 wins, 2 draws, 0 loses) even if Nakamura was supported by Rybka (another chess engine) in the first two

[1] The remarkability of this accomplishment is highlighted by the fact that even today, 23 years after his defeat, G. Kasparov claims that there was cheating from Deep(er) Blue's side in the second and sixth games of the re-matching series of 1997 – namely, that there was some human intervention in certain parts of both games. Whether this claim is true or not, is of little significance since in the years that followed Kasparov's defeat chess machines have displayed remarkably superhuman playing level.

[2] The match was played in 2014 when Nakamura had an Elo ranking o 2798, being #5 in FIDE world ranking.

games and was given a pawn advantage instead of Rybka's support in the rest two (Klein, 2014).

In spite of the aforementioned approaches' efficiency, AI applications that rely on Deep Learning methodologies also have some drawbacks. Of these, probably the most alarming – and the ones of which we are mostly concerned – are non-interpretability and vastness of cognitive load (for both human as well as machine players). Before we proceed to discussing the above, we shall mention that for the rest of this thesis, we will use the definitions of explainability and interpretability presented in (Arrieta, et al., 2020: 85-89). Namely, we will say that a model is:

- ***explainable*** when it is capable of yielding reasons and/or details that clarify its functionality to human users (Arrieta et al., 2020: 85);
- ***interpretable*** when by its design it allows for a human to understand its functionality (Arrieta, et al., 2020: 85).

Note that interpretability is, by its very definition, a static property of a model, totally defined by its design – in the words of Arrieta et al., it is a model's "passive characteristic" (Arrieta, et al., 2020: 84). On the contrary, explainability allows for models that are not necessarily interpretable in terms of their design to be considered explainable in the sense that explanations about their higher level functionality may be produced and presented utilizing post-hoc explanation methodologies.

Also, we will say that a model is ***post-hoc explainable/interpretable*** or ***understandable*** when it is capable of allowing for a human to understand its functionality *without* revealing the way in which data and information are internally processed (Montanov, Samek, Müller, 2018: 2) – i.e. the model is understandable in terms of higher level functionality and not necessarily at a lower level. Lastly, we define, as in (Arrieta et al., 2020: 85), model ***transparency*** as the capability of a model to be understandable by itself, that is, without utilizing any external tools while we also adopt the following three levels of transparency (Lipton, 2018: 42-45, Arrieta, et al., 2020: 88-92):

- ***simulatability***, which refers to the capability of a model's function to be efficiently simulated by a human – i.e. in case a human is provided with access to all input data as well as to all the model's parameters, then they should be able to arrive to the same conclusion as the model in reasonable time (Lipton, 2018: 42);

- ***decomposability***, which refers to the capability of separately explaining and understanding each part of a model – namely, input data, model parameters as well as the model's calculation itself (Lipton, 2018: 44).
- ***algorithmic transparency***, which refers to the attribute of the learning algorithm itself being explainable, in the sense that a human can prove that a unique solution will be produced even if unforeseen data are provided to it (Lipton, 2018: 44-45).

In the above setting, it is clear that levels of transparency are presented in a descending order with respect to how transparent the corresponding models are. That is, a simulatable model is also decomposable and algorithmically transparent – since a human user can fully simulate its function efficiently – and a decomposable model is algorithmically transparent as well – since each of its components, including its learning algorithm, is comprehensible by a human. Also note that the reverse inclusions, in general do not hold – i.e. there exist models which fall into one of the above categories but not to any above them[3].

Bearing in mind the above, Deep Learning paradigms, being by their very definition black-box Machine Learning methods, fail to be characterised as interpretable in a similar manner that they fail to be characterised as transparent – given that they are not even algorithmically transparent, since a Deep Neural Networks' learning algorithm is not ensured to be equivalently functional on different settings (Lipton, 2018: 45). Nevertheless, there are several techniques such as ones deploying feature relevance or the construction of local explanations (Arrieta, et al. 2020:87-88) that do provide explanations on the decisions of a Deep Neural Network in a post-hoc manner – for a more complete review and taxonomy of post-hoc explainability techniques utilised in Deep Learning see (Arrieta, et al., 2020: 95-99) and, especially, Figure 11, p. 99 therein.

The above lack of transparency that characterises Deep Learning approaches also allows for another drawback to emerge: not only is high cognitive load required by humans trying to interpret a deep neural network's behaviour but also the machine itself needs

[3] For more details, consult (Arrieta, et al., 2020: 88-92).

to put effort in terms of computation time and information that needs to be processed. Indeed, as far as the human side is concerned, yielding initially unexplainable output, Deep Neural Networks often demand from users a significant amount of effort in order for the latter (i.e. humans) to comprehend their output. This high demand in cognitive resources undermines the extent to which users trust their results as well as makes maintenance and systematic study of such systems more complicated.

The above weak points of current Deep Learning approaches in general and in particular in computer chess have led us to the following interesting questions:

- Q1. Is it possible to design chess engines whose behaviour – i.e. the moves they suggest and/or play – is interpretable by humans in a non-post-hoc manner? That is, is it possible to design and implement a chess engine which will be comprehensible by a human user without using any other external tool and/or methodology?
- Q2. As an extension of the previous question, how could we transfer experts' domain knowledge to machines so as to make them, on the one hand, at least acceptably efficient chess players as well as, on the other hand, make the learning process more efficient? In a sense, this is equivalent to asking how one could transfer human intuition and heuristics about chess playing to a machine *directly* and not by letting it passively observe other games – be it games that have been played by other entities or by the machine against itself.
- Q3. In case the above are theoretically tractable, which are the domain specific characteristics of chess that should be taken into account as far as the construction of such a system is concerned? In other words, which attributes of chess will be of significant importance in such an approach and how will they affect the system's design and implementation? Furthermore, which are the views of domain experts about the plausibility and feasibility of such an approach?

Our aim is to seek and provide answers to all three questions by finding appropriate learning semantics that would allow for transparent interaction between humans and machines as well as design a chess related user interface that can accommodate such functionality. Moreover, we also aim to present the above to the chess community – i.e. professional chess players as well as chess coaches – in order to assess our work and rationale behind it.

1.2 Motivation behind our work

As we have already demonstrated, Deep Neural Networks as well as other non-transparent approaches are characterised by two major disadvantages: (i) they are not interpretable by humans and; (ii) they demand a vast amount of cognitive resources from humans to comprehend them while they also are computationally expensive to build both in terms of time as well as in terms of required training data.

In the context of chess, the above weaknesses result to the game being, at some extent, "re-invented" by machines with their playing style often alienating even professional chess players and coaches[4]. One may account for two major factors about the aforementioned alienation when it comes to computer chess. At first, machines do not perform, as explained above, in a way transparent to and interpretable by human players, which naturally demands by humans increased cognitive effort when it comes to understanding and studying moves played or suggested by several state of the art chess engines.

Secondly, chess machines do not *explicitly* utilise human knowledge available when it comes to chess playing. On the contrary, they draw all their "knowledge" about the game and its tactics by studying games either played by other (human) chess players or by themselves. Even in tree search based approaches, human knowledge is hard-coded in the machine's software and/or hardware, being, thus, inaccessible by the end-user. As a result, a chess machine plays chess not according to some high level strategic rules accompanied by a tactical understanding of the game but solely based on a tactical comprehension of each position, relying on its massive superiority against human players to overcome any strategic incapability of its own.

1.2.1 Some examples where traditional approaches fail

We will further elaborate on the above ideas – which constitute our basic motivation to study chess and address questions Q1, Q2 and Q3 – by providing some examples. Consider the position shown[5] in Figure 1. In terms of pieces, the situation seems slightly in favour of the white – three pawns and a bishop for a rook. However, the crucial point

[4] These were indeed the words of some of the coaches we have come in contact with during the research for this thesis when discussing AlphaZero's playing style.

[5] We would like to thank an anonymous reviewer who suggested this position as one in which most chess engines fail.

here is not material but whose turn it is to move. In case white moves first, then this position is a draw, since none of the two sides can substantially damage the other - both kings can protect their pawns from the opponent's pieces. However, in case black moves first, there is an option 1. … Rh2 which seems promising since it threatens to take either the white bishop at g2 or the white pawn at f2 – since the white king does not have enough time to protect both of them in this case.

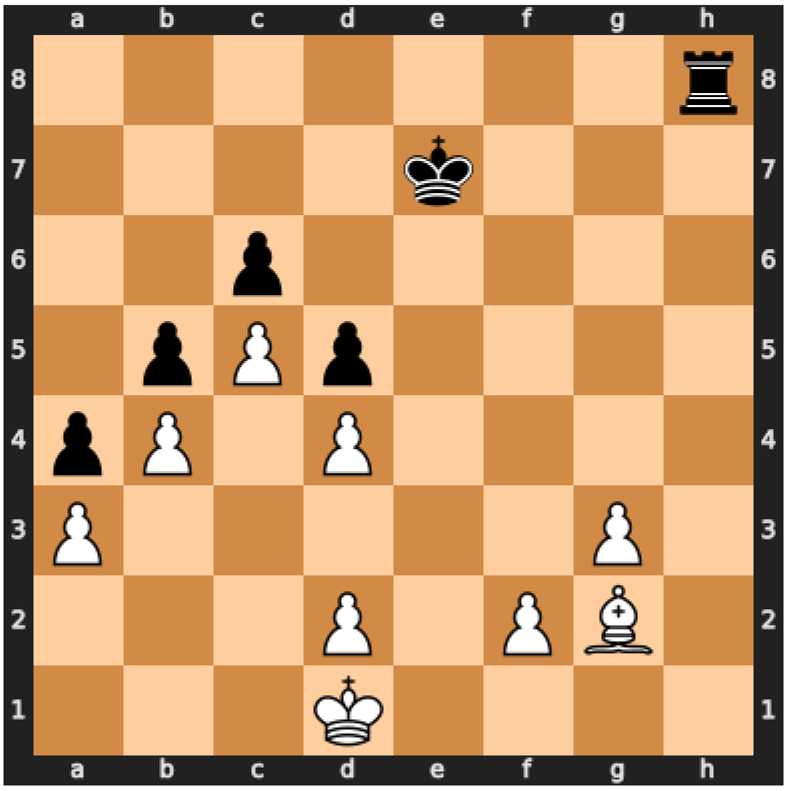


Figure 1: A troublesome position for most chess engines (black to move).

We will analyse both options – i.e. white and black playing first – using Stockfish 10. At first, let us assume that it is white's turn, so, any of 1. Ke1 and 1. Ke2 should be preferred in order to move the white king closer to the king's side pawns and protect the bishop in g2. Stockfish agrees with our view on the situation and it suggests 1. Ke2 as the best move with a -0.00 evaluation – i.e. an absolute draw, as we analyzed above.

We now set up the same position but this time it is back's turn to play. Stockfish now suggests as black's best move 1. … Rh2 with a -3.22 evaluation – i.e. it almost wins the game in favour of black, since ~3 is a typical winning evaluation for most chess engines. However, if we take a closer look to the board, we see that the following move sequence leads to a draw: 2. Ke2 Rxg2 3. Kf1 Rh2 4. Kg1. Indeed, from this position – see Figure 2 –

black has no immediately useful move and after the white plays 5. f3 the game is ensured to be a draw[6].

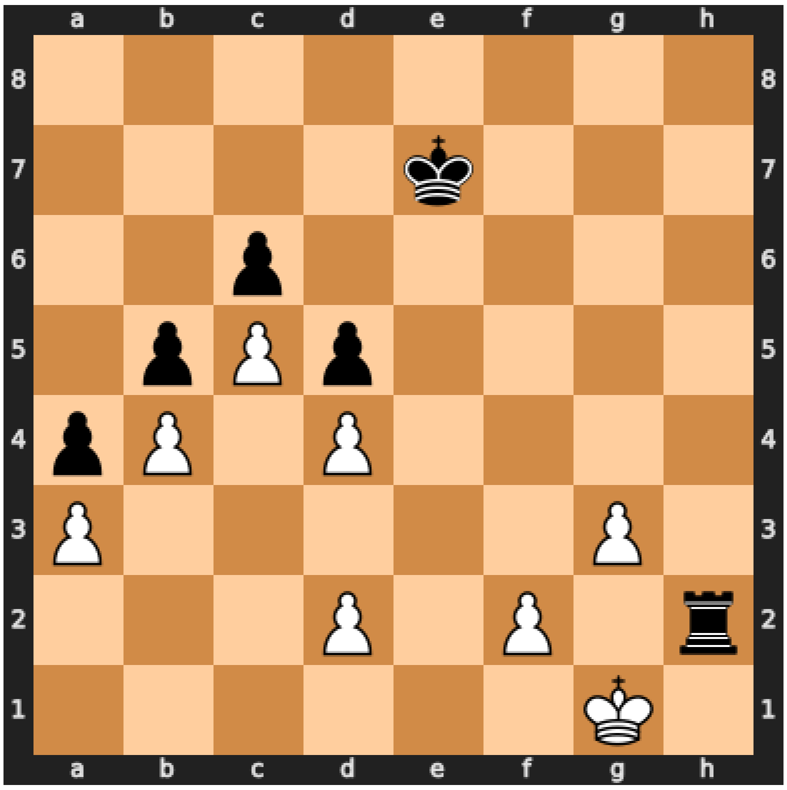


Figure 2: A draw position (black to move). After black's move, 5. f3 constructs a fortress on white's king's side which is impossible to penetrate for black.

This is not the only position known to cause trouble to chess engines. Another famous one is shown[7] in Figure 3. There, black have superior material – two rooks for a bishop – and, were it their turn to play there is no doubt that they win – e.g. 1. … bxc3+ opens the b file for black rooks while it breaks white's chain of pawns.

However, in case it is white's turn, there is a striking path to draw. Indeed, starting with 1. Ba4+!! the black are forced to accept the white's bishop sacrifice, otherwise, 1. … Kc4

[6] Observe that the black king has no way to move behind the white's pawns since the paired white pawns at f3 and g3 form a wall. As far as the remaining black rook is concerned, it has two open files to take advantage of, e and h. Nevertheless, both are within the white king's reach – the white needs exactly one tempo to move their king around g1, g2, f1 and f2 according to where the black rook is located.

[7] This position was suggested by International Master I. Kourkounakis, whom we sincerely thank.

2. Bb3+ Kb5 3. Ba4+ and white threaten with draw by triple repetition. However, the following forced sequence of moves: 1. ... Kxa4, 2. b3+ Kb5 3. c4+ Kc6 4. d5+ Kd7 5. e6+ leads to a clear draw, since, whatever may be black king's fifth move, 6. f5 creates a fortress around the white king – see Figure 4. So, in case white plays there is a single move which leads to a totally closed position which allows white to escape with a draw, in spite of finally being down in material by two rooks!

The key move for white in the above setting, 1. Ba4+!!, is relatively easily spotted by a somewhat experienced amateur. However, it is not such an easy task for most chess engines. Again, we provide Stockfish 10 with the initial position shown in Figure 3 and ask for the best move for the white. Surprisingly, it gives 1. c4+ (-9.92 at 26 moves depth) which naturally leads to 1. ... Kxc4 and gradually to the destruction of white's position, since now the black have the possibility to clear a file with their two rooks.

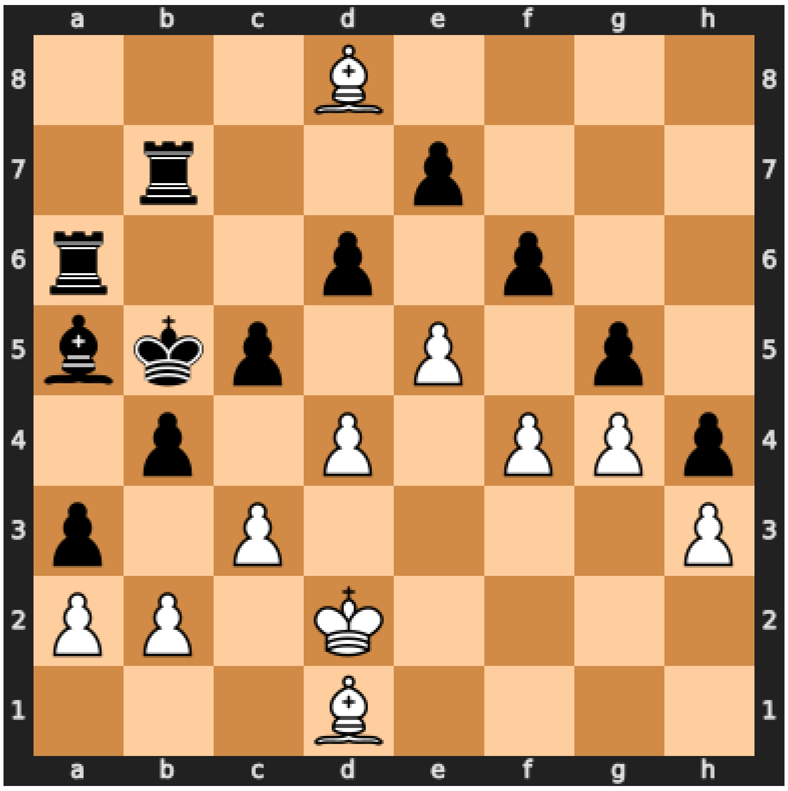


Figure 3: White plays and draws!

Even more impressive is how Stockfish values white's single chance to escape with a draw. When requested to analyse 1. Ba4+ Stockfish returns five different variants, with the best of them being evaluated at -12.69 at 32 moves depth. That is, it completely loses a quite obvious move sequence for most human players and, on top of that, considers it a very bad move.

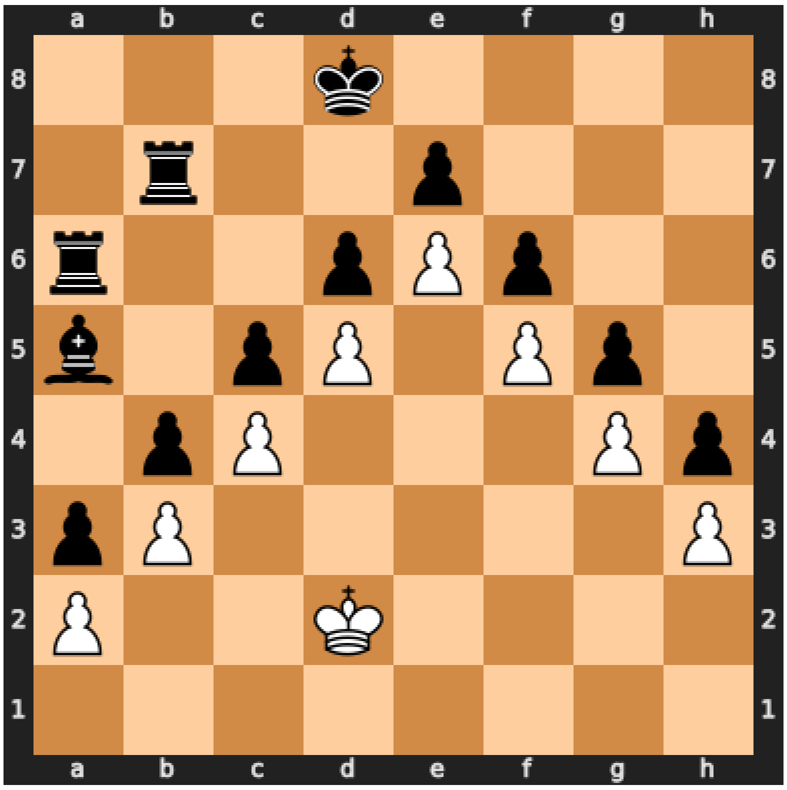


Figure 4: The white's fortress.

The above positions, as well as some similar ones, indicate how several situations in chess which are relatively easy to tackle for most amateur players, pose impossible challenges to otherwise powerful chess engines. A reason that may account for this absurd behaviour may be the way in which most chess engines "learn" chess, when compared to humans. While a human player is capable of seeing both strategic as well as tactical aspects of every position on the chessboard, most contemporary chess machines have, as we have already discussed, access only to the game's tactical aspects as well as any domain specific metrics are hard-coded into them. Nevertheless, given the vastness of training in terms of previous games "studied" by a chess engine, they appear to play according to typical human strategic patterns. Borrowing some terms from biology, a chess engine's "phenotype" appears to be both strategically and tactically oriented – as that of a human chess player – while its "genotype" is purely tactical.

1.2.2 Deeper in the weaknesses of contemporary chess engines

While in most cases, especially against human players, the aforementioned form of lack of game understanding is overwhelmingly covered by such machines' tactical superiority, positions as the above two, which allow for far less tactical manipulations, unveil some of their weaknesses. We consider it useful at this point to proceed to a further analysis of the positions presented in subsection 1.2.1 in order to extract some more abstract common features that could account for the failure of chess engines of Stockfish's calibre[8].

As we have observed, both positions presented in 1.2.1 are more or less closed, pruning tactical capabilities for both sides and also restricting the effect of any additional material on the board to a minimum level – see especially Figure 3. Taking a closer look, we will also observe that both of them rely on the same ideas that white exploits in order to arrive to a drawing position. Indeed, at first, some material is sacrificed – in both our cases, a bishop – while the next moves aim at restricting black's lines of attack and, consequently, building a more closed position.

We could now take another step towards a more abstract view on the two positions presented above. The defending side's sacrifice can be interpreted as sacrificing some tactical feature – in both cases material but not necessarily restricted to it – in order to gain enough time to bring the game to a position from which the opponent's tactical advantages cannot be utilised. Indeed, in both cases the white behaves in a way opposite to that many chess machines would choose. Seeking a better position in terms of strategic attributes, game tactics are sacrificed for the sake of destroying the opponent's tactical advantages as well. That is, instead of trying to improve their own position from a tactical point of view – e.g. by saving their bishop at g2 in Figure 1 which leads to taking the minimum possible tactical damage of losing one pawn (at f2) – the white play so as to minimize black's tactical possibilities to a level that they will be no threat to their position.

Additionally, strategic attributes are the ones which explicitly lead white's playing – alongside with the goal of ruining black's tactical game as we explained above – in the sense that in both positions the white does not strive to improve their tactical game but to take advantage of existing and create new strategic features that will lead to a "steady state" on the board from which no side has something to win.

1.2.3 A possible solution to such cases

Should a chess coach use the above positions as study cases in a class, they would possibly aim, among others, to demonstrate situations in which the concept of "fortress"

[8] At this point we should mention that Leela Chess Zero, an open-source Deep Neural Network chess engine based on Deepmind's AlphaZero approach yielded similar results to the ones reported by Stockfish 10, in terms of suggested moves, position evaluation and so on.

comes into play. Describing the notion of a fortress, one may say that, in short, it is a position from which the attacking side, which typically has some advantage – be it of tactical or strategic nature – cannot "capitalise" this superiority due to the position being too restricting – either by itself, as in Figure 4, or by allowing one side to constantly prohibit any further progress, as in Figure 2, or by other similar means. This is a definition that can be easily captured in terms of natural language and can also be relatively easily explained and discussed between people as well as be recognized on a board whenever it occurs. Nevertheless, would it be possible to take advantage of this piece of human knowledge by transferring it to a chess machine? Or, more broadly, could we coach a chess machine in a way more or less similar to the one a chess coach trains their students and provide to it such fragments of our knowledge directly and explicitly?

Should the above be possible, we could take advantage of the already accumulated knowledge of humans about chess and avoid the excessive amount of computational power needed by Deep Learning or tree search algorithms in order to efficiently train a chess machine. Moreover, by allowing for human chess players to coach a chess machine, it comes as a natural consequence that the machine's konwledge should be extensible in the sense that, any time needed, the coach, would be able to alter the it and, hence, its "view" of the game itself.

1.3 Structure of the Thesis

The rest of this thesis is structured in five chapters which address in various ways the three posed questions (Q1, Q2 and Q3) presented in 1.1.

In chapter 2 we review previous approaches to computer chess as well as how they relate to human chess – i.e. the extent to which they have been successful at competing human players. We also present and discuss to more detail approaches several approaches that have had a significant impact in the history of automated chess playing as far as their design principles as well as their implementation are concerned.

In chapter 3 we discuss possible answers to questions Q1 and Q2 as presented in section 1.1. Namely, we present and explain through numerous examples a theoretical framework of human-machine interaction (Machine Coaching) which will serve as our foundation upon which the rest of our proposal will be based. Also, in this chapter we introduce a (first order) language which is intended to be utilised as a means of interaction between humans and machines in the context of Machine Coaching.

In chapter 4 we present our implementation of all the required functionality regarding Machine Coaching as well as that of a chess Graphical User Interface (GUI). Altogether, we have designed a chess bot which is capable of learning chess with the assistance of a human coach. Namely, the designed chess bot starts with no actual knowledge about the game of chess other than the game's rules and, by receiving feedback from a human coach in the form of arguments in favour or against moves given a board position, it gradually refines its playing style, converging to its coach's theory about the game of chess under certain conditions.

In chapter 5 we present results regarding the designed system's evaluation in two orthogonal directions. At first, we assess the efficiency of our reasoning engine – mostly in terms of execution time against several other system parameters i.e. the time in which it constructs a representation of its theory given knowledge (in the form of prioritized if-then rules) and contextual information. Next, we present and discuss opinions of several professional chess players and coaches regarding the adopted methodology and to what extent it seems applicable as well as what features should it additionally include in their opinion.

Lastly, in chapter 6 we summarize all the work done so far towards the construction of a chess bot capable of capturing a human coach's high-level strategic guidelines as well as the useful feedback we received by chess experts. Moreover, we also present possible future directions towards which the current work could be extended, based, among others, on experts' feedback presented in chapter 5.

Part of the functionalities developed for the purposes of this thesis have been also utilised for other purposes, among which is the WeNet project. For more information, see the project's official website: https://www.internetofus.eu/.

Chapter 2
Literature Review

In this chapter we review works related to chess machines/engines and the approaches adopted in their design. More precisely, this chapter's structure is as follows: (i) in section 2.1 we present a brief historical review of the most seminal attempts to construct human-level playing chess machines until Northwestern University's Chess 4.x chess machines; (ii) in section 2.2 we present and discuss chess machines and chess engines after Chess 4.x including the seminal Deep and Deep(er) Blue as well as contemporary Deep Learning approaches.

2.1 The first attempts to automate chess playing

In this section, we will explore approaches to automate chess playing that date before the emergence of highly competitive chess machines such as Northwestern University's Chess 4.x and the alike. As surprising it might be, humanity attempted to automate chess playing long before computers had emerged in the middle of the twentieth century, nevertheless with no significant success in beating human players in most cases.

2.1.1 Mechanical approaches

As we have already mentioned in chapter 1, probably the first attempt to construct a non-human entity that could play chess dates back to 1770, when Wolfgang von Kempelen constructed *The Turk,* a mechanical chess playing automaton (Standage, 2002: 18-23). The Turk appeared to be a highly skilled player since it managed several victories against professional chess players of its time as well as Napoleon Bonaparte and Benjamin Franklin (Standage, 2002: 18-23). As one may easily suspect, The Turk was actually a fraud, however, this was unveiled no sooner than it was destroyed by a fire in the Chinese Museum of C. W. Peale (Levitt, 2000: 40-41) in 1854. The game was played by a human player who was hidden inside the machine, which was about the size of a table – about 80cm tall, 60cm wide and 1.10m long (Standage, 2002: 22-23). Remarkably, the human player inside the Turk was not seen even if all its four rear doors were opened simultaneously, adding more to the mystery of its successes against human players. Nowadays, The Turk has been reconstructed by J. Gaughan and is periodically exhibited to several plays and conferences, mostly related to magic (Levitt, 2000: 243).

After Kempele's Turk, there were also some other attempts based on the same idea – i.e. a human playing inside a chess "automaton" on its behalf – such as *Ajeeb* or *Memphisto* (Shaeffer, 1997: 90, Gumpel, 1889: 46) with the latter being more innovative compared to the its predecessors since the human player was not hidden inside Memphisto but, instead, controlled it from distance using an electromagnetic controller (Harding, 2012: 284).

It was no sooner than 1912 and Leonard Torres y Quevedo's *El Ajedrecista* (*The Chess Player*) that the first actual chess automaton was designed and presented in public. The Ajedrecista was not capable of playing an entire game of chess but a specific end-game, namely a king and rook versus king finale – the Ajedrecista played as white (king & rook) while the human player had control of the black king. It was always successful at mating the black king while it was also capable of recognizing any illegal moves of the black king and alert the opponent (Atkinson, 1998: 20-22). The Ajedrecista's functionality was based on a chessboard on which pieces were plugged, forming a closed electrical circuit which represented each position on the board (Montfort, 2003: 76). Also, the machine was programed with a simple and complete – yet not optimal, in terms of moves played – algorithm for mating a game using a king and a rook against a sole king – this position is a quite easy endgame since the only thing the white has to be careful of is to avoid stalemate[9]. Quevedo's El Ajedrecista is still functional until today and is kept at Madrid's Universidad Politécnica.

2.1.2 Non-mechanical approaches

Apart from Quevedo's El Ajedrecista there was no other significant mechanical chess automaton throughout human history, at least not one matching the Ajedrecista's level. Also, it is no sooner than the 1940's that another attempt to design a non-human chess playing entity took place, but this time in a different setting. Konrad Zuse, a German computer scientist, often referred to as the inventor of modern computers (Rojas, 1997: 5), started developing at 1942 what is thought to be the first chess engine in human

[9] A *stalemate* is a position in which a player has no legal move to play while it is their turn and is the basic draw position in chess.

history (Knuth, Pardo, 1976: 203). However, Zuse's intention was not to design a chess engine for the sake of it. Instead, his chess engine was more supposed to serve as a complex example about a high level programming language that he had been designing then, *Plankalkül.* As a result, we do not have much evidence about its performance against other (human) players, however, it was known to fully support the game of chess – a remarkable achievement for that time, given that Plankalkül supported only one primitive data type (bits) (Bauer, Wössner, 1972: 679-681).

In the years after Zuse's chess machine, several other attempts to design and implement a chess engine were presented, however most of them failed to run on contemporary computers. We shall at first focus in a remarkable paper by Shannon (Shannon, 1950: 1-18), in which he introduces several of the ideas that would later be part of the design most chess machines. In this direction, after discussing how calculating all possible positions and then deciding which is the best move – either by some ad-hoc measure or by consulting a "dictionary" (Shannon, 1950: 4) which assigns each position to the "best" move, according to some expert – is not a feasible strategy to address the problem, he introduces the notion of an *evaluation function* that, according to some predefined attributes, assigns a utility value to each position[10]. Furthermore, he also describes some algorithms according to which a chess playing machine could play against other players.

Delving into more details, Shannon first describes a greedy minimax algorithm that allows a machine to choose the best move modulo a given depth of search n (Shannon, 1950: 5-12). The algorithm's key idea is that, assuming that an evaluation function assigns positive values to positions that benefit the white and negative values to these that are of black's benefit, the white side in each turn seeks to find that move which maximises the evaluation function given that the black side seeks to minimise it. So, given a depth of search n and assuming that the machine plays as white, the machine seeks to find the move that the black would play after n moves – i.e. the move that *minimises* the evaluation function at the final position. Then, given that move, it returns back to find which of the remaining white moves *maximise* the evaluation function given that black will play the move found previously. Progressing in a similar manner, it returns to the root of the possible positions' tree – i.e. to the current position – and returns the move that maximises the evaluation function given all the next moves that it

[10] The very same idea has also been roughly described by Wiener in (Wiener, 1948: 48-50).

has calculated. Evidently, as Shannon himself observes, this is not an efficient method for finding the next move to play for the machine, at least not for a sufficiently large depth n since it requires for the entire game tree to be computed (Shannon, 1950: 7-8).

Shannon concludes his work with considerations and projections about future directions chess engine design could explore. What he is at most concerned (Shannon, 1950: 12-16) are ways in which the tree of all game positions could be pruned so as to lower the computational complexity of the above algorithm, while he also discusses ways in which excessive pruning could be avoided – so as to allow for seemingly short-term bad moves to be explored in case they can lead to a long-term advantage – e.g. pawn sacrifices or moves that lose a tempo[11] and so on.

About a year after Shannon published the work discussed above, namely in 1951, A. Turing and D. Champernowne publish what was considered to be the first chess program that is capable of playing an entire game of chess. *Turochamp*, as was its name, was designed to play chess against other players by calculating the "best" move according to a position evaluation function – more or less as Shannon had already described in (Shannon, 1950: 5-12) – performing a two-move depth search (Copeland, 2004: 563-564). However, in contrary to Shannon's approach, position evaluation as well as move selection do not follow a recursive minimax approach. Instead, Turochamp assigns a certain value to each occurring position on the board and then proceeds in selecting the move which has the highest average score (Copeland, 2004: 563-564). For instance, should white's 1. d4 lead to say 20 different possible responses by black with the resulting positions evaluating at $f(p_1), f(p_2), \dots, f(p_{20})$, then Turochamp would prefer 1. d4 on condition that the average evaluation of this move, i.e. $\frac{f(p_1)+f(p_2)+\cdots+f(p_{20})}{20}$ is the highest among the corresponding average scores of other moves. As with Shannon's approach on the game, the algorithm was extremely heavy for the time's machines resulting to it never being run during Turing's lifetime. However, Turing and Champernowne executed the algorithm at least once by hand, playing against Champernowne's wife (Champernowne's Obituary, 2000: 262) – for the record, this game was the first and only victory as well as match of Turochamp.

[11] In chess, a *tempo* is a single move of one of the two players. In general, tempo in chess is a significant strategic factor since oftentimes it determines whether a position is winning or not for one side – we shall present and study such examples in next chapters of this thesis.

Following Turing and Champernoewne's example, several other computer scientists decided to explore the possibility of building a chess playing machine, with more or less success. Namely, short after Turochamp, D. Prinz designed, in 1952, a computer program that could be run on contemporary machines and solve any mate-in-two problem (Bowden, 1957: 292-295) – i.e. in case in a position it was guaranteed to exist a mate combination in two moves, then Prinz's program was capable of finding it. Four years later, in 1956, *Los Alamos chess* was designed by P. Stein and M. Wells in the Los Alamos Laboratories (Anderson, 1986: 104-105). Los Alamos chess was a chess machine that could play a full game of a simplified chess variant, named *Los Alamos*[12] and which is also the first actual chess machine that has recorded a victory against an amateur chess player[13] (Pritchard, 1994: 175).

Right after these partial solutions to the problem of designing a machine capable of playing a whole game of chess against a human player, in the late 1950's (namely, 1958) Bernstein's chess program was presented in public. It was a Shannon Type B program (Shannon, 1950: 15-16) in the sense that it used a forward pruning methodology in order to reduce the size of the search space while running a two double-move[14] depth minimax search – four half-moves in total – 2 for the playing side and two for the opponent's (Bernstein, Roberts, 1958: 5-6). As mentioned by the long-time world champion Emmanuel Lasker, who had been invited to play against Bernstein's chess program, "It played a passable amateur game" (McCorduck, 2004: 185).

The decades of 1960's and 1970's were quite fruitful as far as computer chess is concerned. In 1962, one of the first chess machines that played "convincingly well" appeared; *Kotok-McCarthy* – named after its two designers, A. Kotok and J. McCarthy (Kotok, 1962: 12). Kotok-McCarthy was based on a minimax alpha-beta searching

[12] Los Alamos chess variant is played on a 6 × 6 board with no bishops for both sides as well as the following restrictions: (i) no castling is allowed; (ii) no promotion of a pawn to a bishop is allowed; (iii) pawns move strictly one square at a time and, consequently, there is no initial pawn double move or en-passant capture (Pritchard, 1994: 174-176).

[13] The machine won its third match of Los Alamos chess in 23 moves against one of the Los Alamos Scientific Laboratory assistants who had been taught the game's rules for the first time some days prior to the game against the machine (Pritchard, 1994: 175).

[14] In chess, a *double-move* is a pair of moves where the first is played by white and the second by black

approach[15] and, unlike previous approaches, it had variable search depth, in the sense that it stopped either at eight (8) half-moves depth or at a "stable position" (Kotok, 1962: 8). Its evaluation function took under consideration material balance, king protection, pawn structure, tempo advantage and development (Kotok, 1962: 2-6).

In middle 1960's (namely, 1965-67), R. D. Greenblatt presented *Mac Hack 6*, another chess machine that, compared to its predecessors, was highly successful. Indeed, Mac Hack 6 was the first computer program that was allowed to compete in usual chess tournaments against human players while it also was the first one to score a victory against a human player in an official tournament game – namely, Game 3 in Massachusetts State Championship, 1967, the second tournament it took part (Levy, 2013: 65). Sooner that year, it had also managed to draw a game[16] against a human player. Mac Hack 6, as its predecessors, made use of a minimax alpha-beta pruning tree search algorithm accompanied by a position evaluation function to detect the best move in a given board position. Its success compared to previous approaches was due to several factors. To begin with, Greenblatt's good understanding of the game seems to have played some significant role – as per his own words "they were very weak players, both Kotok and McCarthy … And I said, gee, I can do better than that" (Gardner, 2005: 13-14). Indeed, Greenblatt was a more knowledgeable chess player than most computer scientists that had tried to develop chess machines/programs in the past, something reflected in the about fifty different heuristics that were utilized in Mac Hack's design in order to efficiently narrow down the plausible moves list that was used to expand the game tree in minimax search (Greenblatt, 1969: 804).

[15] Alpha-beta tree search relies on the simple idea of pruning branches of the tree that are found to lead to moves of lower utility for the playing player than some previously found move sequence. For instance, if the white have found a move in some branch that evaluates, say, at +1.5 (in favour of them) then any branch found to lead to some move of value $v < +1.5$ is rejected, given that the black will play so as to minimise white's benefit – consequently, maximizing their. As a result of the above, the alpha-beta algorithm is sensitive to the order in which the moves are checked, since a better move encountered early in the search will lead to, expectedly, more branches to be pruned (Mashey, 2005: 12-13).

[16] Again, the first official draw of a computer against a human player – in the winter Amateur Tournament of the Massachusetts State Chess Association, Game 3 (Levy, 2013: 64).

Another factor that can account for Mac Hack's efficiency was that it was the first chess program that included a hash table containing all previous positions played, something that dramatically reduced search time. More precisely, when a position was found during search, it was stored in a hash table alongside with the search results – i.e. the value of the position as calculated by the evaluation function. Furthermore, the depth to which this position was found was also stored among other information, so when the very same position occurred on board and as a terminal position of some branch of the game tree – i.e. leaf node –, the results were immediately recovered from the hash table, saving significant computation time (Greenblatt, 1969: 806-807).

On top of the hash table as well as the refined heuristics regarding plausible move suggestion, there were also other features that facilitated Mac Hack's work. For instance, a secondary search was deployed when a search yielded a new move as a possible optimal choice. The search begun at the depth at which the analysis of the main variant of the new move had stopped, typically for a single double-move, so as to "cheaply" increase the search depth (Greenblatt, 1969: 807). Mac Hack 6 was also provided access to an opening book designed especially for the purposes of the project. The main objective was to reduce the chance that Mac Hack would fall for typical opening traps often set up by human players.

Most of the approaches that appeared during the next years were mainly modifications and extensions of Mac Hack's design principles. The next major shift in computer chess came with Northwestern University's *Chess* program, designed by L. Atkin and D. Slate, starting from 1968 (Jennings, 1978: 108-109). Chess 4.5 was the first computer program to win in an official tournament against human players in 1976 while the year after, the program's improved version, Chess 4.6, won the 84th Minnesota Open against players close to Master level (Hapgood, 1982: 827-830). The same year, 1977, Chess 4.6 also managed to defeat the United States chess champion, W. Browne, who had himself invited the program to a match after the latter had won the 84th Minnesota Open earlier this year (Douglas, 1979: 111). The next version of Chess, Chess 4.7, while it lost a match series against D. Levy by 4½ - 1½ (3 wins, 1 draw and 1 loss for Levy), became the first computer chess program to ever defeat a Master level human player in a single game (Douglas, 1978: 84).

Compared to almost a decade of no significant progress after Greenblatt's chess program, Chess's consecutive wins against highly skilled human players came as a bolt from the blue. As expected, various innovations were included in the program's design. Indeed, apart from maintaining known good practices such as alpha-beta pruning introduced by McCarthy and Kotok (Kotok, 1962: 2-3) and transposition tables introduced by Greenblatt (Greenblatt, 1969: 805-806), Chess also introduced bitboards[17] in computer chess, which facilitated parallel processing in several circumstances. Apart from that, Chess also extended Greenblatt's transposition hash table idea as follows: apart from keeping encountered positions alongside their value, as computed by the machine's evaluation function, and the depth at which they were found, they also allowed for moves that have previously led to a cut-off to be kept in it. As a result, move prioritisation became more efficient, leading to better performance given alpha-beta pruning algorithm's sensitivity in the order by which the moves are examined.

Chess 4.x series also introduced some other techniques which, while being of smaller magnitude than the ones already mentioned, jointly contributed by a significant amount to it being such a sufficient chess player compared to any other previous approach. These include, but do not restrict to, the following ones: (i) instead of generating all plausible moves and then expanding each one of them, Chess did generate one move at a time and abandoned search should that move lead to a cut-off (Frey, Atkin, 1978: 189); (ii) instead of re-calculating the evaluation function at each position, Chess kept it gradually updated as it went through several adjacent positions leading to a significant reduction in computation time (Frey, Atkin, 1978: 190); (iii) serialisation of the way the evaluation function is computed, which means that decisive factors – such as material balance – are computed first and, if needed, any other computations are made (Frey, Atkin, 1978: 191); (iv) use of bitmaps encoding win/draw results in most typical endgame categories, so as to compensate for humans' intuitive play at this stage of the game (Newborn, 1977: 119-129).

By the end of the 1970's, chess machines had started making massive steps in approximating human level of playing. While Chess 4.7 did not manage to finally beat Levy in an official match, it was the first machine to record a win in a single game against a chess master. The leaps done since the beginning of the 20th century and Quevedo's mechanical El Ajedrecista to Northwestern University's Chess were gigantic and

[17] *Bitboards* are data structures that encode board squares and/or game pieces using bits (Atkin, Slate, 1983: 84).

astonishing, should one take into account that computer science was a relatively young field. However, it yet remained for the best human chess players to be convincingly defeated by machines.

2.2 Mastering the game

The rise of the 1980's was accompanied by the announcement of the Fredkin Prize by the Edward Fredkin Foundation of Cambridge, Massachusetts (Mittman, 1980: 5). The prize was three-tiered, including: (i) a $5,000 prize for any chess machine that would first achieve a Master status; (ii) a $10,000 prize for any chess machine that would be the first to acquire an International Master status and; (iii) a $100,000 prize to any chess machine that would manage to become World Chess Champion – i.e. beat a running World Chess Champion.

The first tier prize, as Chess 4.7 had already shown, was closer than one would suspect. Indeed, in 1981 *Belle chess*, a chess machine developed by K. Thompson and J. Condon at Bell Laboratories was the first chess machine to get a Master level ranking. While previous advances in computer chess were, mostly, due to a parallel work in creating more efficient algorithms as well as more powerful machines, Belle was superior compared to most chess machines of its time due to its hardware based approach. While algorithmically it run an alpha-beta pruning algorithm as its predecessors – using some new improvements which sped it up (Fishburn, 1980: 29) – it was the faster move generation as well as move sorting and evaluation that made it successful. As per Thompson's words "It ran about 160,000 positions per second. Typical software ran about 6,000 positions per second; that's on a fast machine" (Mashey, 2005: 14).

Another computer machine that introduced several innovations in computer chess was *Cray-Blitz*, developed by R. M. Hyatt, A. E. Gower and H. L. Nelson. While, as Belle did, Cray-Blitz relied on state of the art hardware, it also utilised software that had not been used in earlier chess machines (Hyatt et al., 1990: 111). As with all major chess machines of its time, it relied on alpha-beta pruning as well as on most of the heuristics and search space reduction techniques that had been introduced in the Chess 4.x series of chess machines (Hyatt et al., 1990: 111-112). Its most significant contribution was essentially its *quiescence search* methodology. To begin with, during search the search tree was split into three regions with the first of them being a typical full-width search utilising alpha-beta pruning (Hyatt et al., 1990: 112-115) as well as known heuristics – e.g. null move, transposition tables and so on.

Once search in tree region one had come to its end, search to tree region two started from certain tree nodes by expanding by two double moves – i.e. four plies – from the point where search in region one had stopped. As stated in (Hyatt et al., 1990: 115-116), the main purpose of this second (quiescence) search region was to further elaborate on positions where a king seems to be in a difficult position – where difficulty is measured in terms of the evaluation function – aiming to gain some advantage, usually in material. Then, Cray-Blitz entered the third region of the tree, which at some occasions overlapped with region two, and was dedicated to further exploring solely capture moves.

Another remarkable feature of Cray-Blitz is the way in which position evaluation was conducted. It is indeed one of the first machines that introduced quite sophisticated strategic parameters into piece ranking which allowed for some kind of contextually variable value of pieces – e.g. in certain positions, knights were considered better than bishops while in others the opposite was true. Also, Cray-Blitz introduced a novelty regarding tournament games – both against machines as well as against human players. Namely, it took a flexible stance against time utilisation by entering a "deep think" mode whenever it was in a difficult position – be it the game's current position or some best move in a main variation examined. During that "deep think" mode it allocated additional time than the predetermined one in order to find a move/variation in which it could restore balance in the game (if possible). That additional time allocation was proportional to material loss that occurred in that position (Hyatt, 1990: 129-130).

Having achieved Master ranking, the next goal for computer chess was to beat a chess Grand Master in an official game. This took place no sooner than 1988, when *HiTech,* designed by H. Berliner and C. Ebeling, beat A. Denker, US grandmaster in a four game match with a 3½ - ½ score (3 wins and one draw for HiTech). HiTech was quite innovative in terms of algorithmic means it utilised, since, as claimed by the H. Berliner, the machine advanced "from Master to Senior Master with no hardware change" (Berliner, 1989: 12). While it remained close to the almost sacred alpha-beta pruning algorithm as well as to most techniques that had been introduced by the Chess 4.x machines, it also introduced some significant novelties.

At first, it built up on Cray-Blitz's legacy and set time utilisation as one of its priorities in terms of design. HiTech internally ranked moves to "obvious" and "hard" ones and accordingly allocated time – the characterisation was at some extent hard-coded while HiTech also had access to an oracle of knowledge in order to facilitate such assertions. Thus, it could efficiently dedicate more time in searching moves that would lead it to losing positions so as to find possibly better candidate moves. Also, HiTech developers utilised the notion of Singular Extensions (Anantharaman et al., 1988: 2) in order to easily detect moves that were considerably better than other sibling moves – i.e. moves at the same depth in the search tree – and focus on path variations of these moves, leading to deeper search when a sequence of strong moves was found (Berliner, 1989: 18-19).

Furthermore, HiTech deviated from previous approaches regarding another crucial factor that allowed it to capture more complex relations on the chessboard and, consequently, drastically improve its performance. Until its time, most known approaches to computer chess made use of linear evaluation functions, resembling more or less weighted means of all the factors taken into account. On top of that tradition, HiTech introduced non-linear evaluation functions (Berliner, 1989: 16-17) which are, in fact, arbitrary functions of the state of any square on the chessboard. Such functions are not computed serially, since this was expected to lead in a blow-up in computation time, but, instead, they are computed through a reduction process using a table look-up method (Berliner, 1989: 17-18).

Following HiTech, there were several approaches to adopt and refine its methodologies such as Deep Thought, an ancestor of Deep Blue. For reasons of completeness, we shall refer to both of them. Deep Thought was the first chess machine to beat a human Grand Master in a tournament game in 1989 (Hsu, et al., 1990: 44). It ran on a quite strong machine for its time, allowing it to compute about 750,000 positions per second (Hsu, et al., 1990: 45). Deep Thought, in parallel with HiTech, made use of the singular extension algorithm which helped focus on particularly important positions, as in the case of HiTech, and was, along with the latter, the first known chess machines to use selective search – i.e. deviating from a typical alpha-beta pruning by temporarily looking only towards one direction and much deeper than the rest branches of the search tree.

But, possibly the most important feature introduced by Deep Thought was the notion of a "self-training" evaluation function (Hsu, et al., 1990: 45-46). Instead of hard-coding the weights on a linear evaluation function as with most past approaches or introduce non-linearity as HiTech did, Deep Thought used a hill climbing algorithm to find optimal values for the evaluation function's weights making use of a pool of 900 grandmaster games (Hsu, et al., 1990: 47). However, since the above method was computationally expensive, it was used in certain difficult cases, while in most cases a simpler approach was adopted that sought to minimise mean square error of the estimated move against the optimal move – which was either a move that was played by some grandmaster in a sample game or a result returned from a deep search, in case of some known concept (Hsu, et al., 1990: 48).

Building on the innovative approaches introduced by Deep Thought, its team proceeded in building Deep Thought 2 – which was intended to be stepping stone towards the construction of Deep Blue (Campbell, et. al., 2002: 58) – as well as Deep and Deep(er) Blue. For the rest of this chapter, we will refer to both Deep and Deep(er) Blue as Deep Blue, referring primarily to the latter version, which beat G. Kasparov in 1997.

Extending the work done in Deep Thought, Deep Blue adopted a somewhat different stance on game tree search. Having at its disposal vast amounts of computational resources and utilising strong parallel processing – resulting to searching capacity of 2-2.5 million positions per second, the highest of its time – instead of typical alpha-beta pruning techniques, Deep Blue adopted a highly selective search, avoiding pruning at earlier stages in almost any case (Campbel, et al., 2002, 60-61) so as to efficiently explore any variant to some minimum depth, in case it led to a better position later.

Furthermore, Deep Blue took advantage of both software search, utilising sophisticated heuristics as Deep Thought, which modifying a set of about 8,000 parameters, as well as enhanced hardware search – which was, nevertheless, non-alterable (Campbell, et al., 2002: 64-72). Not restricting to the above, Deep Blue was also granted access to an opening book as well as an "override" book which allowed for last minute changes prior to some game so as to avoid typical and known opening traps. Once the opening of a game had come to its end, rendering Deep Blue's opening book useless, Deep Blue could take advantage of an "extended" book which provided access to about 700,000 games played by grandmasters. Using information from this book, Deep Blue was able to assess moves not only with respect to its own evaluation function but also to enhance its "judgement" by providing bonuses or penalties to its evaluations (Campbell, et al. 2002: 76-78). At last, Deep Blue could also access a series of endgame databases, mostly the ones provided by K. Thompson (Mashey, 2005: 18-22) which contained endgame positions and their outcome/winning move sequences encoded in bitboards and obtained by a retrograde analysis starting from a final board position and proceeding backwards.

The years that followed Deep Blue's victory against G. Kasparov, the then-reigning World Chess Champion, saw several chess engines, both open as well as closed-source (e.g. Rybka). Probably the most successful of them all that also still uses an alpha-beta pruning based methodology is Stockfish (Romstad, et al., 2008). Stockfish relies on most of the aforementioned heuristics – e.g. the null move heuristic – as well as typical representation methods – such as bit-boards and transposition tables – but amongst its most significant attributes is its very aggressive pruning policy. However, since it has repeatedly been empirically validated that over-pruning may lead to winning variants being pruned, eventually leading to defeat, Stockfish also adopts a late move reduction policy. That is, it does not prune branches of the game search tree immediately but allows for a shallow search at first so as to verify, up to some certain level, that no good move is being cut off.

Recently, approaches that deviate from the typical tree search methodologies presented up to now have emerged. From these, Deepmind's AlphaZero is undoubtedly the most eminent. AlphaZero, in contrast with most chess computer programs, is not chess specific, as stated in (Silver, 2017: 1). Instead, it relies on a tabula rasa reinforcement learning methodology under which it can accumulate knowledge about the game by self-playing and self-assessment. More precisely, when it comes to choosing a move, instead of utilising a typical alpha-beta pruning variant, AlphaZero plays simulated games against itself while in each simulation it picks up its next move by being one of high probability and value according to its deep neural network. Once all simulations have been completed, it returns the corresponding probability vector that occurs from the above process – also known as Monte-Carlo Tree Search (MCTS). Given the above probability vector, π, at a certain position AlphaZero picks moves according to it while, once the game is over, it compares its evaluation ($+1$, 0 or -1 in the case of chess corresponding to win, draw and loss respectively) with its neural network's estimation about the outcome of the game and it adjusts the parameters accordingly using gradient descent so as to minimise the occurring mean square error. Simultaneously, it also seeks to maximise the similarity of its network's policy[18] vector to the probability vector π (Silver, 2017, 2-3). AlphaZero, in spite of being game independent, has successfully managed to defeat Stockfish in recent experiments by an impressive score: 28 wins, 72 draws and 0 losses.

[18] A *policy* in terms of reinforcement learning is a probability distribution defined over the set of pairs of states and actions according to which the agent acts.

Chapter 3
Machine Coaching

In this chapter we will present and explain through examples of application the learning methodology of Machine Coaching as well as the induced human-machine interaction protocol. The structure of this chapter is as follows: (i) in section 3.1, Machine Coaching is presented as a meeting point between typical Machine Learning methodologies and Declarative Programming; (ii) in section 3.2, a first order language suitable for reasoning in the context of Machine Coaching as well as its syntax are defined; (iii) in section 3.3, the theoretical principles of a reasoning engine that conducts reasoning in the context of Machine Coaching are presented.

3.1 High-level description of Machine Coaching

As we have already discussed in chapter 1, in order to utilise human chess players' knowledge about the game in a chess machine's training process as well as improve the machine's performance on several cases where it is easy for a human to adapt, we argue that a more declarative approach is necessary. For these purposes, Machine Coaching (Michael, 2019: 81-82) seems an appropriate choice, as it will be further explained in this chapter.

At a higher level, Machine Coaching is a Machine Learning paradigm that allows a human user to transfer knowledge, personal preferences and/or heuristics to a machine by providing pieces of advice to it in the form of arguments in favour or against certain actions/behaviours. Assuming for a while that we have at our hands a language through which a machine and a human can interact in such way, a typical Machine Coaching scenario would be the following one:

1. At first, a human user asks for some piece of advice – in our context, it could be some move suggestion, for example – from the machine on a certain circumstance – e.g. on a specific board position.
2. The machine, with any knowledge it currently has at its disposal – which, in the beginning of the coaching process may include nothing – as well as any

contextual information available, returns a piece of advice as well as an argument supporting that piece of advice which, at the same time, serves as an explanation to the human user.

3. The user, on seeing the suggested action/behaviour as well as the corresponding explanation, has two options[19]:
    a. Either to accept the machine's advice as well as the explanation it has provided about it, when consequently nothing changes in the machine's knowledge and the user may ask for another piece of advice – return, hence, to step 1;
    b. Or to not accept the machine's advice and/or the corresponding explanation, in which case, the user is prompted to provide counter-argumentation to the machine about why they did so. The machine integrates the above counter argumentation to its knowledge base and is again ready to accept any new advice request from the user – return, hence, to step 1.

In order to clearly demonstrate the above human-machine interaction, we present a simple example of the above in the context of chess. Again, we assume that there exists a common language of communication between the human user and the machine in which all the following arguments are being expressed. Also, let us assume that we have already had some brief training session with our bot and have given it a single guideline: to play its pieces towards the centre[20] of the board when still in the opening phase of a game[21].

Assume that our chess bot plays as black and it is its turn to play in the position shown in Figure 5, which has occurred from white's 1. e4. Upon our request for its next move – i.e. its advice/suggestion in the given context (position on the chessboard) – it responds with the following passage:

[19] The exact interaction protocol will be presented and discussed later on in this chapter.

[20] We define as the board's centre squares the squares e4, d4, d5 and e5.

[21] We make this assumption for the purposes of our demonstration only.

"I would suggest making the move 1. ... e5 since we are still in the game's opening phase and in such cases it is better to move pieces towards the board's centre."

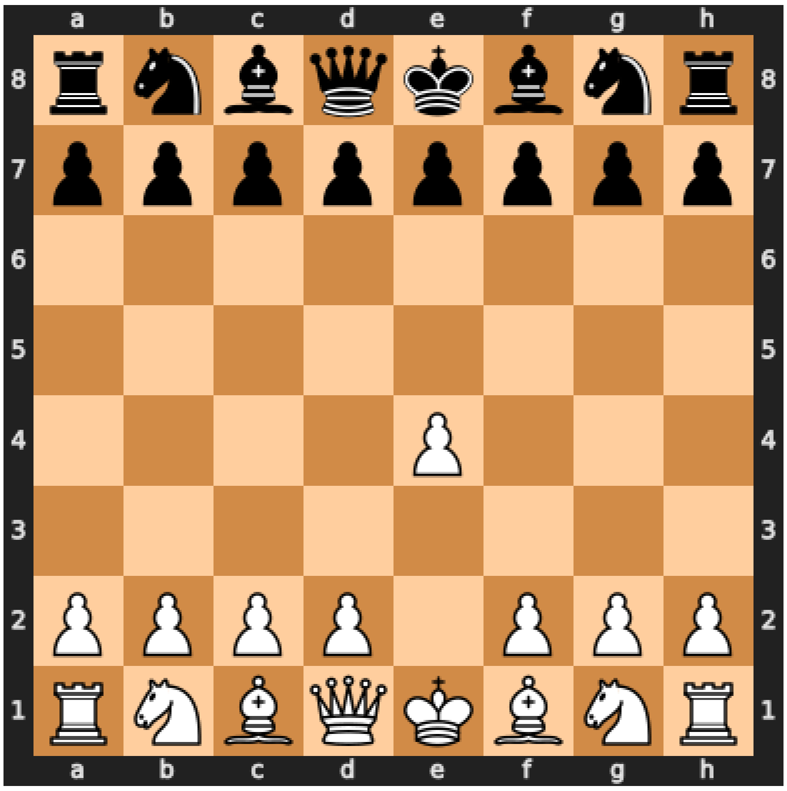


Figure 5: The position after 1.e4.

So far so good, since 1. ... e5 is an acceptable response to 1. e4. Next, the white plays 2. Nf3, threatening the black pawn at e5. We ask again what is on our bot's "mind", given this situation on the board – shown in Figure 6 – and we receive the following response:

"I would suggest making the move 2. ... d5 since we are still in the game's opening phase and in such cases it is better to move pieces towards the board's centre."

We should expect such advice since we have assumed that the only guideline our bot has is to play its pieces in the centre of the chessboard when it is in the opening stage of a game and the only move towards the board's centre left is 2. ... d5. However, since our bot is still "inexperienced", we would like to prohibit it from playing gambits[22], especially ones known to lead to inconvenient positions in the middle of the game for

[22] A *gambit* in chess is an opening variant in which some material is sacrificed in order to gain some strategic or tactical advantage as compensation for it – e.g. better control of the board's centre or the initiative against the opponent.

the black – this one, in particular, is known as the *Elephant's Gambit* (de Firmian, 1999: 150-151). So, we decide to advice our bot to be more cautious by protecting its pieces when they are under threat by providing the following counter-argument to its (suggestion, explanation) tuple:

> "When you are still in the opening phase of a game and some of your pieces is threatened, prefer moves that defend this threat by supporting the piece under threat rather than moves that simply bring your pieces to the centre."

Given this counter-argument, we *locally* de-activate our previous advice of moving pieces towards the board's centre squares in situations where there is also another feature on the board, other than the fact that we are currently on the opening stage of the game – i.e. an opponent's threat to win some material over us. The refined knowledge base would lead the bot to choose between moves such as 2. ... Nc6, 2. ... d6, 2. ... Qe7 and so on, which all defend the black pawn at e5. By continuing in a similar manner, we provide further advice to the chess bot, gradually allowing it to capture larger parts of our theory about playing chess.

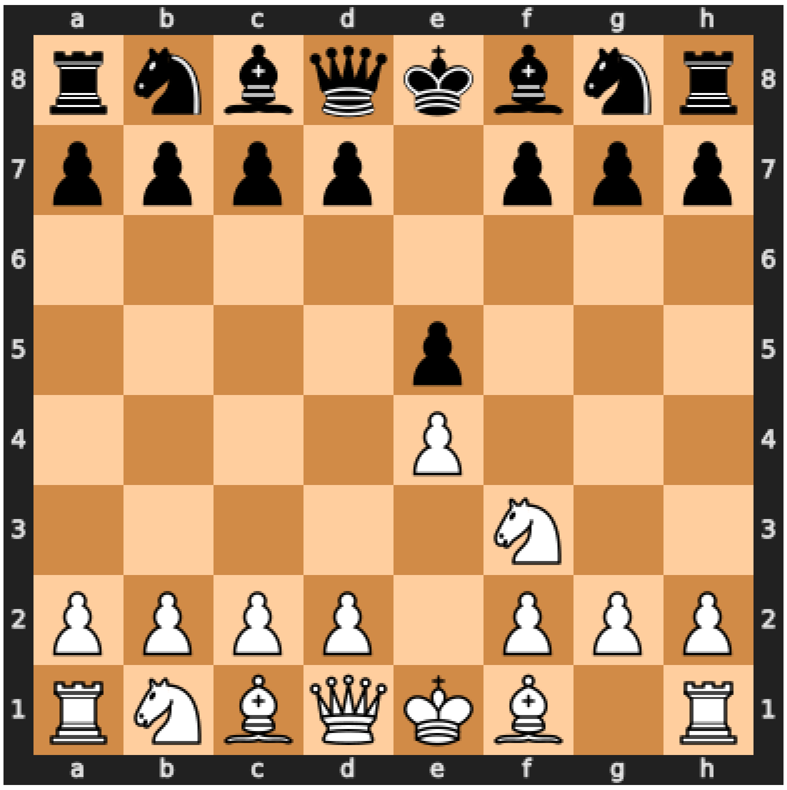


Figure 6: The position after 2. Nf3.

As one may observe, Machine Coaching stands between Machine Learning and Declarative Programming. On the one hand, it falls in the broader category of Machine

Learning since it allows for a machine to accumulate (domain specific) knowledge by capturing a (human) user's heuristics and preferences. On the other hand, in the paradigm of Machine Coaching the machine is not instructed in an imperative way how to search for or construct that knowledge. On the contrary, the user expresses declaratively what they want the machine to behave like by providing advice and, in a sense, the machine finds a way to follow these instructions, if possible.

However, apart from allowing a human to train a machine in a more declarative way, Machine Coaching also allows for the learning process to be transparent with respect to the system's functionality. At each time, the machine informs the user about the rationale on which it relied to reach its conclusion, so the user is always aware of the way it operated. The above will allow us to consider Machine Coaching as an interpretable learning methodology, as defined in (Arrieta, et al., 2020: 84) since at each time the user is capable of understanding the entire process by which the model reasons and learns based on their own advice. Moreover, Machine coaching may also be considered a simulatable paradigm since each suggestion is based on rules that originate from the user's theory about some domain and, hence, the user, given all the information available to the machine is expected to be capable of simulating its function[23].

In the following sections we will thoroughly discuss Machine Coaching as presented in (Michael, 2019: 83-85) providing chess-specific examples about each new notion introduced. Thus we expect that more light will be shed on how interpretability and transparency are achieved to a significant extent by adopting Machine Coaching as our learning methodology.

3.2 A language for Machine Coaching

In this section we formalise a language that will support interaction between humans and machines in the context of Machine Coaching while it will also allow for a further

[23] This may need to be further studied since, even if the machine's theory is at every time a subset of that of the user, assuming the former had no previous knowledge, this may not necessarily imply that a user can simulate the machine's functionality within a reasonable time frame. One factor that could account for this is that some fragments of the user's theory may lead to counter-intuitive inferences – from the user's viewpoint – which may lead to spending more time than expected to understand the machine's rationale behind them.

formalisation of Machine Coaching's theory. As with most parts referring to Machine Coaching in this thesis, we mostly follow the presentation in (Michael, 2019: 83-85).

3.2.1 Describing the language of Machine Coaching

Until now, we have vaguely used the term *knowledge base* to refer to a data structure where the machine's knowledge, as provided by the user, is kept. However, as the previous example clearly demonstrates, it does not suffice for the machine's knowledge base to be a typical list/array of rules. Indeed, in the above example, we may rewrite the two arguments presented as follows – again, we use natural language to express these arguments for reasons of simplicity:

> **Argument 1**: [Rule 1]
>
> *Rule 1*: If a move brings a piece to the centre of the board then suggest that move.
>
> **Argument 2**: [Rule 2, Rule 3]
>
> *Rule 2*: If a move brings a piece to the centre of the board but there exists a threat to one of my pieces and another move that defends that threat then reject the first move.
>
> *Rule 3*: If a move brings a piece to the centre of the board but there exists a threat to one of my pieces and another move that defends that threat then suggest the second move.

As we see, in the context of Figure 6 all three rules are triggered which leads to a conflict between rules 1 and 2. As a result, we need some mechanism in which such conflicts are resolved, which cannot be accommodated by a typical knowledge base. Thus, as described in (Michael, 2019: 83), the notion of a *prioritised knowledge base* is introduced. In a prioritised knowledge base, apart from the rules themselves, a priority relation is defined over all pairs of conflicting rules which facilitates conflict resolution, in the sense that rules of higher priority are preferred when a conflict arises.

So, in our example, we would like Rule 2, which is conflicting with Rule 1, to be declared of higher priority than Rule 1 so as to capture the exceptional character Rule 2 has over Rule 1. Indeed, while Rule 1 expresses a general principle in chess opening theory – i.e. that of moving one's pieces towards the centre of the board in order to control it – Rule 2 describes a position on the board which demands a different manipulation due to an additional feature – i.e. the opponent's threat to take a piece of ours – and, hence, the usual way of action should not be followed in this case.

Bearing these in mind, we now proceed on presenting and discussing the theoretic tools we will need in order to strictly formulate the aforementioned notion of a prioritised knowledge base as presented in (Michael, 2019: 83). At first, let us assume that we have at our disposal a first order language L which:

- Allows for countably many **constant** symbols (also called *constants* for short). Constants are intended to be interpreted as our universe's entities, so, in the context of chess some typical constant symbols could be `e4`, `a3` and `d7` which all represent squares or `pawn`, `queen` and `rook` which represent chess pieces or even `black` and `white` which represent the colours of the two players.
- Allows for countably mane **variable** symbols (also called *variables* for short). Variables are intended to serve as placeholders for constants in various expressions.
- Allows for countably many **predicate** symbols of arity n, where $n \in \mathbb{Z}_{>0}$ (also called *n-ary predicates* or simply *predicates* for short). Predicates are intended to be interpreted as relations among entities of our universe. In the context of chess, a binary predicate could be `starts_from(Move,Square)` which is intended to express the relation of a square (`Square`) being the starting square of a move (`Move`). In the same way, a unary relation in the universe of chess could be `plays_as(white)` which is intended to express the fact that the bot plays as white.
- Contains a special binary predicate symbol, called *the equality symbol*, which is, *a priori* interpreted in any case as the binary relation of congruence between two entities of our universe[24].
- Contains a **universal quantifier** symbol, which is intended to be interpreted as *for all entities in our universe*.

[24] This is not verbosely stated in (Michael, 2019: 83) however we considered useful to include an equality symbol in our language since it was found to be necessary in several occasions – see next chapter for more.

- Contains three **logical connective** symbols which are intended to be interpreted as logical conjunction, material implication and logical (classical) negation respectively.

At this point it would be useful to introduce some concrete notation for our language so as to help clarify future expressions and definitions. More precisely:

- We will denote variables by any finite alphanumeric sequence – allowing also for the text underscore symbol – which starts with a capital letter of the latin alphabet. As a result, `Piece_2` and `Colour` do denote variable symbols while `_Piece` or `colour` or `2piece` do not.
- We will denote constants by any finite alphanumeric sequence – again, allowing for the text underscore symbol – which starts with a lowercase letter of the latin alphabet. As a result, `pawn` and `e2` do denote constant symbols while `Pawn` or `3queen` are not.
- We will denote predicates by any finite alphanumeric sequence – again, allowing for the text underscore symbol – which starts with a lowercase letter of the latin alphabet and is preceded by a comma-separated list of its arguments enclosed in parentheses.
- We will denote the special equality binary predicate by `?=(X,Y)`, where `X` and `Y` are its two arguments.
- We will denote the logical connector of negation with the symbol - (the minus symbol) while we will use the comma symbol (,) so as to denote logical conjunction. So, if `colour(pawn,white), is_at(pawn,e4)` holds it means that both `colour(pawn,white)` and `is_at(pawn,e4)` hold, while in the opposite case in which `-is_at(pawn,d4)` holds then `is_at(pawn,d4)` does not hold.
- We will also denote the logical connector of material implication by the word `implies`. For instance, `is_at(Piece,e4) implies is_at_centre(Piece)` means that either `is_at_centre(Piece)` or `-is_at(Piece,e4)` holds.
- In all expressions, we will assume that any variables are within the scope of some universal quantifier and, as a result, we will not introduce any special notation for it.

Now, using the above, we define the following:

- A **literal** is either a predicate itself or its negation. Also, a literal is called *negative* when it consists of a negated predicate and *positive* otherwise. Positive literals are interpreted as predicates – i.e. they represent n-ary relations between entities of our universe – while negative literals are interpreted as relations that don't hold in our universe. We also define *conflicting literals* to be two literals such that one of the is the negation of the other. For instance, literals `-colour(black)` and `colour(black)` are conflicting.
- A **rule** is a triplet (`name`, `body`, `head`) were:
  - `name` is any finite alphanumeric sequence – including text underscore, as above – and denotes the rule's name;
  - `body` is a conjunction of literals, that is, given the notation defined above, a comma-separated list of plain of negated predicates;
  - `head` is a single literal.

Also, as far as rules are concerned, we define the following notation:

```
name :: body implies head;
```

where `implies` is the material implication connective, `name`, `body` and `head` correspond to the rule's name, body and head respectively, `::` is a delimiter separating the rule's name from its main part and `;` denotes the rule's end. Lastly, we also say that two rules are *conflicting* in case their heads are conflicting literals, as defined above.

Now, given the above, we also define a **context** to be a non-empty collection of pariwise non-conflicting literals. Our intention is for a context to be interpreted as a set of facts that describe a specific situation in a given setting. So, in our case (chess), a context describing the initial position on the board may contain literals such as `is_at(pawn,white,a2)` or `is_at(queen,black,d8)` and other similar ones, where `is_at(Piece,Colour,Square)` is intended to be interpreted as "a piece of type `Piece` and of colour `Colour` is at square `Square`".

Now, we have now at our disposal all the needed means in order to define a prioritised knowledge base. We will say that a tuple $k = (\rho, \prec)$ is a **prioritised knowledge base** if:

1. ρ is a set of rules as defined above;

2. $\prec$ is an irreflexive antisymmetric priority relation[25] $\prec$ over all pairs of conflicting rules in $\rho \times \rho$.

At a higher level, a prioritised knowledge base seems capable of capturing the subtleties of the advice taking procedure described in 3.1. Indeed, as we have already argued, it is mandatory that conflicting rules are included in the machine's knowledge base since in this way the refutable nature of human argumentation with respect to contextual information can be efficiently captured.

It remains to provide a formal definition of an argument. However, we consider it useful to first provide some examples of application of the language we have described above in the context of chess. As a result, we will present and discuss the formal definition of arguments as well as anything related to reasoning and learning in the context of Machine Coaching at the end of the chapter, in section 3.3.

3.2.2 Examples of Machine Coaching in chess

Up to now we have defined a language through which all interaction between humans and machines will take place in the context of Machine Coaching. In this subsection we are going to present some examples specifically in the context of chess so as to demonstrate the capabilities of Machine Coaching's language as described in 3.2.1.

To begin with, we will first present the example discussed in section 3.1 in order to make a direct comparison between natural language and the language we have designed for Machine Coaching as well as for reasons of completeness. So, at first we would like to describe the rule:

Rule 1: If a move brings a piece to the centre of the board then suggest that move.

To do so, we will at first assume that we have access to a binary predicate, `to_square(Move,Square)` where `Square` is intended to be substituted by any square constant, hence, meaningful substitutes for `Square` are:

- `Square`: `a1`, `a2`, … , `h7`, `h8`.

while meaningful substitutes for variable `Move` could be:

[25] In other words, $\prec$ satisfies the following: (i) there is no rule r such that $r \prec r$ (irreflexivity); (ii) if $r \prec r'$ for two conflicting rules r, r' then $r' \prec r$ does not hold (antisymmetry).

- `Move`: `e2e4`, `g2f3` and, in general, any legal move in a UCI format[26].

Also, we will make use of a literal `suggest(Move)` which will denote that a move should be suggested as an appropriate one. Using the above, we can at first express the meaning of the phrase "a piece is moved at the centre of the board", using the following five rules[27]:

```
C_1 :: ?=(Square,e4) implies is_at_centre(Square);
C_2 :: ?=(Square,d4) implies is_at_centre(Square);
C_3 :: ?=(Square,e5) implies is_at_centre(Square);
C_4 :: ?=(Square,d5) implies is_at_centre(Square);
Occ :: to_square(Move,Square), is_at_centre(Square) implies
occupies_centre(Move);
```

Next, we need to define what the "opening phase" of a game of chess is. To do so, we will need a predicate like `current_move(Count)` where `Count` is some integer valued variable which corresponds to the current double-move count. Using this predicate, we may define the game's opening phase as follows[28]:

```
Op :: move_count(X), ?<(X,11) implies game_phase(opening);
```

In the above expression we also used a mathematical comparison predicate, `?<(X,Y)` which is intended to be interpreted[29] as $X < Y$. In order to express Rule 1, we will also

[26] A move in *UCI format* is a string of 4 or 5 characters where: (i) the first two characters denote the rank and the file of the move's starting square; (ii) the third and the fourth characters denote the rank and file of the square to which the piece is moved and; (iii) the last (fifth) character denotes the piece to which a pawn is promoted in case the move is a promotion move - equal to the empty character if the move is not a promotion move.

[27] The symbol `?=(·,·)` that appears in the following rules denotes our language's equality symbol.

[28] Note at this point that we could have also used a constant symbol `opening` as `Op` rule's head - indeed, our implementation of Machine Coaching's language allows for constants to be interpreted as predicate symbols of zero arity – see next chapter for more. So, it is more of a matter of preference rather than anything else whether one chooses to define a new predicate symbol or just a constant.

[29] We will elaborate on the definition of mathematical relations and operations in the context of our language in the next chapter, where we discuss issues regarding the language's

need a predicate plays_as(Colour) so as to be able to distinguish between the colours of each player as well as a moves(Move,Piece) predicate which is intended to be interpreted as: move Move moves a piece of type Piece. Lastly, we will also assume that we have access to another binary predicate[30], move_played_by(Move,Colour) which is interpreted as: move Move is played by player of colour Colour. Now, given the above predicates as well as rules, we can express Rule 1 as indicated below:

```
Rule_1 :: plays_as(Colour), occupies_centre(Move),
move_played_by(Move,Colour), game_phase(opening) implies
suggest(Move);
```

Before proceeding to expressing Rules 2 and 3 in the language we have defined, it would be useful to first make some remarks regarding the above. At first, we could have avoided rules C_1 to C_4 as well as Occ by creating four "instances" of Rule_1 by using to_square(Move,Square) in a similar fashion as in rules C_1 to C_4 and Occ. However, we preferred to describe a new predicate, occupies_centre(Move) primarily to demonstrate the possibility of "defining" our own predicates as well as because doing so leads to more efficient coding – i.e. it is considered a good practice in the context of our language. Indeed, in the above way we are introducing eventually less rules in the machine's knowledge base since, were rules C_1 to C_4 not used, we would be obliged, to create four "instances" of Occ as well as of each rule that refers in some way to central squares of the board.

Furthermore by "defining" new predicates, we are allowed to define any new higher-level notion we need, provided that it is within the expressive limits of our language – i.e. it is definable through first order if-then rules as defined in 3.2.1 as well as any built-in predicates provided. Moreover, by having concretely defined the notion of central squares once, should it occur that this definition needs to be changed at some time in the future – e.g. extended to include more squares – it suffices to overwrite rules C_1 to C_4

implementation. In general, the implementation allows for all typical mathematical operations and functions to be computed, even if such function symbols are not allowed in our language, in principle.

[30] This predicate is needed since even if exactly one player moves at a time in a game of chess, it is useful for each context describing a certain position to also include the opponent's moves in case a null move is played by us.

with new ones – in case we had not used such rules, we would have to override any rule referring to the notion of "centre".

Now, we proceed to expressing Rule 2 in our language. Rule 2 is expressed in natural language as follows:

> *Rule 2*: If a move brings a piece to the centre of the board but there exists a threat to one of my pieces and another move that defends that threat then reject the first move.

So, as with Rule 1, we will first describe what primitive information is required – which will be encoded in some built-in predicates – as well as define any new higher level information needed. To begin with, we will suppose that we have access to a ternary predicate, `is_at(Piece,Colour,Square)` which is intended to encode the fact that a piece `Piece` of colour `Colour` is located at square `Square`. As a result, meaningful constants that could be substituted in place of each variable are:

- `Piece`: `pawn`, `knight`, `bishop`, `rook`, `queen`, `king`.
- `Colour`: `black`, `white`.
- `Square`: `a1`, `a2`, … , `h7`, `h8`.

Furthermore, we also need access to a predicate `attacked_by(Colour,Square)` which is intended to represent the fact that the player of colour `Colour` attacks square `Square` with at least one of their pieces[31]. Using the above, we could describe a threat as follows:

```
Threat :: plays_as(Colour1), -plays_as(Colour2),
is_at(Piece,Colour1,Square), attacked_by(Colour2,Square)
implies threatened(Square);
```

[31] Note that `atttacked_by` cannot be expressed in terms of `to_square` since pawn capture and non-capture moves are not the same – in contrast with what happens with any other piece.

Having defined what a threat[32] is, it remains to express the notion of defending a square. To do so, we will make use of a new predicate we assume we have access to: `controls(Square1,Piece,Colour,Square2)` which we intend to be interpreted as: a piece of type `Piece` and colour[33] `Colour` form square `Square1` controls[34] square `Square2`.

Using the above predicates we can define the notion of a move defending a square as indicated below:

```
Def :: plays_as(Colour1), -plays_as(Colour2),
attacked_by(Colour2,Square1), to_square(Move,Square2),
moves(Move,Piece), move_played_by(Move,Colour1),
controls(Square2,Piece,Colour1,Square1) implies
defends(Move,Square1);
```

Observe how we do not demand for a piece to occupy a square we defend, since we take care of that in the definition of a threat. Now, we proceed in expressing Rule 2 using all the above, as follows[35]:

```
Rule_2 :: occupies_centre(Move1), to_square(Move1,Square),
threatened(Square), defends(Move2,Square) implies
-suggest(Move1);
```

In a similar fashion, we can express Rule 3 as indicated below:

[32] `Threat` defines threat only for the bot, in the sense that we could not use the same rule to express the fact that we are threatening an opponent's piece at some square. Nevertheless, this suffices for the purposes of our demonstration here.

[33] We need to know the piece's colour since legal capture moves – as well as legal pawn moves in general – are depended on the pawn's colour and not only on its type – i.e being a pawn.

[34] We say that a piece *controls* a square if it can make a capture move towards that square on condition that a piece of opposite colour is located there. So, control moves coincide with pseudo-legal moves of any piece but for pawns which move restricted to their file but can capture pieces only on adjacent files.

[35] We could well have avoided being so explicit in the declaration of `Threat` which would have led to including `is_at` in the declaration of Rule 2, however we decided to keep Rule 2 as simple as possible, using higher level predicates, so as to bear more resemblance to its natural language representation.

```
Rule_3 :: occupies_centre(Move1), to_square(Move1,Square),
threatened(Square), defends(Move2,Square) implies
suggest(Move2);
```

Observe that we have not demanded in none of Rules 2 and 3 that the two moves under consideration are different. So, it could happen that a move that both moves a piece towards the centre of the board as well as defends an opponent's threat exists, which would trigger both Rule 2 as well as Rule 3, leading thus to a conflict. Depending on how we have set priorities among Rules 2 and 3, this move will be suggested – in case Rule 3 is of higher priority than Rule 2 – or not – in the opposite case. We will delve into more details about rule prioritisation in section 2.3 as well as in chapter 4, where we discuss the chess specific user interface we have designed.

3.3 Argumentation and Learning in the context of Machine Coaching

In this section we will discuss the way in which reasoning is conducted in the context of Machine Coaching. To do so, we first need a notion of arguments. In general, as we will see in this section, arguments appear in several occasions throughout Machine Coaching. Namely:

- The machine uses arguments internally in order to reach to a conclusion about which action(s), behaviour(s) or item(s) should be suggested to the user.
- The machine user arguments as a means of interaction with the user. More precisely, when the machine returns a piece of advice, it also returns, as we have already discussed, an explanation about it. The explanation is, actually, the very same internal argument that led the machine to this conclusion. As a result, one may claim that the machine is being by its definition transparent as well as interpretable – in the way defined in (Arrieta, 2020: 85-89) – since it provides access to its internal mechanisms to the user.
- The machine accepts any user feedback in the form of an argument. Should a user disagree with the machine's suggestion and/or explanation, as we have already discussed in 3.1, the user expresses their disagreement by providing counter-argumentation of some kind to the machine.

3.3.1 Arguments in the language of Machine Coaching

We will now proceed to defining an argument, as presented in (Michael, 2019: 83). Let g be some literal, $k = (\rho, \prec)$ be a prioritised knowledge base, x be some context – i.e. a set of pairwise non-conflicting literals – and A a subset of $\rho \cup x$. We say that A is an **argument** for g in x under k if the following hold:

1. $A \neq \emptyset$,
2. starting form literals in $A \cap x$ and by repeatedly applying modus ponens using rules in $A \cap \rho$ we can infer g,
3. g cannot be inferred by any proper subset of A, i.e. if $\emptyset \neq B \subset A$ then we cannot infer g as described in 2 by substituting B in place of A.

Also, we will refer to the (unique) rule $r \in A$ that has g as its head as the argument's *crown rule*[36].

Before we present some examples of arguments, we will first discuss the conditions that appear in the above definition. The first condition demands that an argument is a non-empty set, i.e. it contains at least one literal from x or at least one rule from[37] ρ. As far as the second condition is concerned, it demands that our argument is actually an argument for ρ, in the sense that we can infer it from our hypothesis. Note at this point that it is possible that an argument does not contain any rule at all, in which case it should contain g. One could interpret such detrimental arguments as restating some already known fact – since we assume that literals belonging to a context are by default interpreted as facts that are true in a certain situation within our setting.

The third condition is a more technical, yet quite important one. It demands for arguments to be minimal in the sense that nothing else other than what is needed is included in an argument for g. This allows us to avoid trivial cases of making a distinction between arguments for a certain goal literal g that differ by, say, a rule or a literal that does not lead to further implications regarding g.

[36] Observe that such a rule may not always exist – e.g. in case $A = \{g\}$ – while its uniqueness, whenever it exists, is guaranteed by condition 3.

[37] In case an argument A is a singleton it can either be $\{g\}$ or $\{r\}$ where r is a rule that has g as its head.

We now proceed in some examples, so as to clarify the above. To begin with, let us revisit the example we have presented in 3.1. Consider the following fraction of the example's knowledge base, let k, where we have simplified `Rule_1` to `Rule_1b` by dropping the dependency on the game's phase for reasons of simplicity:

```
Rule_1b :: occupies_centre(Move), move_played_by(Move,Colour),
plays_as(Colour) implies suggest(Move);
Occ :: to_square(Move,Square), is_at_centre(Square) implies
occupies_centre(Move);
C_1 :: ?=(Square,e4) implies is_at_centre(Square);
C_2 :: ?=(Square,d4) implies is_at_centre(Square);
C_3 :: ?=(Square,e5) implies is_at_centre(Square);
C_4 :: ?=(Square,d5) implies is_at_centre(Square);
```

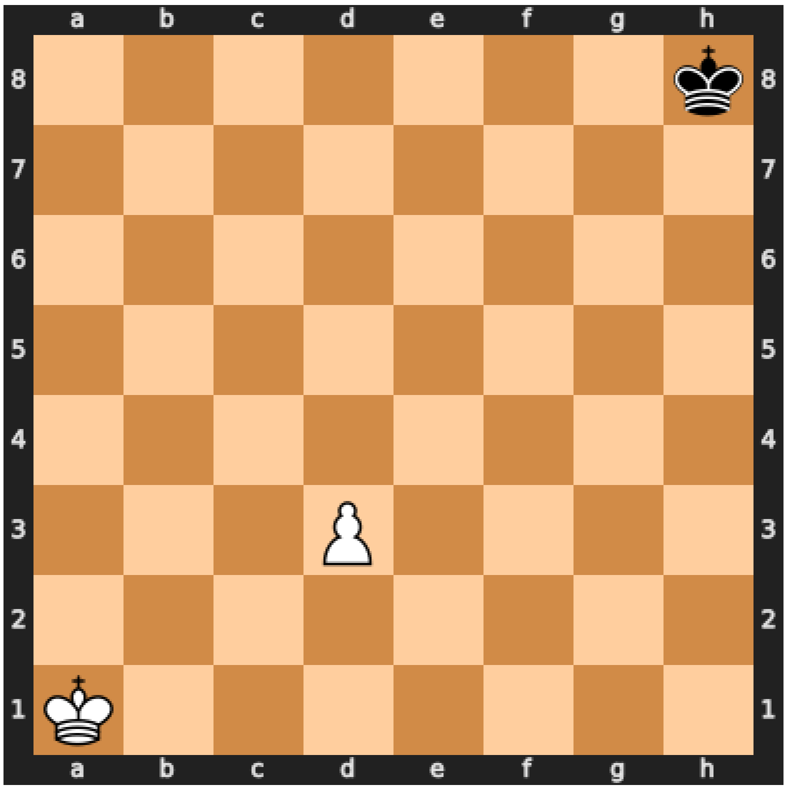


Figure 7: White to move

Also, let us consider the following context, let x – again, we assume there are no other predicates available, so as to keep this first example minimal – which describes the position shown in Figure 7 (white to move).

```
    to_square(a1a2,a2);
    to_square(a1b1,b1);
    to_square(a1b2,b2);
    to_square(d3d4,d4);
    to_square(h8h7,h7);
    to_square(h8g8,g8);
    to_square(h8g7,g7);
```

Let us now assume that we want to examine whether there is an argument for `suggest(d3d4)` in context x under the above knowledge base k. At first, we observe that by considering $A = x \cup k \neq \emptyset$ we can infer `suggest(d3d4)` using rules and literals in A. So, the first two conditions are satisfied. However, the third condition regarding the argument's minimality is not satisfied by any means, since we could, for instance, remove `to_square(h8g7,g7)` or rule `C_3` and we could still infer `suggest(d3d4)` from the new reduced set.

As one may easily observe, the only choice for A that also conforms with the third condition in the definition of an argument is the following one[38]:

```
    A = {to_square(d3d4,d4), C_2, Occ, Rule_1b};
```

Indeed, removing any of the above from A would lead to `suggest(d3d4)` not to be inferred from A – e.g. removing rule `C_2` would result in `Rule_1b` not being triggered and, as a result, its head would not be inferred.

Note that, given a context x, a prioritised knowledge base k and a goal literal g, should there exist an argument A for g in x under k it is by no means guaranteed to be unique. In order to demonstrate this, let us add the following rules to k, with priority higher than any other rule:

[38] Note at this point that `?=(e4,e4)` is not included in our context, however rule `C_2` is included in our argument. As we shall see in chapter 4, `?=(X,Y)` is treated in a broader sense than a congruence relation. More specifically it is treated in the more general context of unification. In our case, this means that `?=(Square,e4)`, given that Square is unassigned, leads to the substitution `Square/e4` which triggers rule `C_2`. In general we will not explicitly include such literals in the representation of a context, however we will assume that they are included whenever needed, as in this example.

```
Rule_5 :: controls_centre(Move) implies suggest(Move);
Rule_4 :: plays_as(Colour), move_played_by(Move,Colour),
to_square(Move,Square1), moves(Move,Piece),
controls(Square1,Piece,Colour,Square2), is_at_centre(Square2)
implies controls_centre(Move);
```

Also, let us extend the previous context with the following literals – we will not refer verbosely to any literals that would be typically included but are not useful in this example:

```
plays_as(white);
moves(d3d4,pawn);
move_played_by(d3d4,white);
controls(d4,pawn,white,e5);
```

Using the above knowledge base and context, we can also see that there is a second argument B for `suggest(d3d4)`, namely:

```
B = {plays_as(white), moves(d3d4,pawn), to_square(d3d4,d4),
move_played_by(d3d4,white), C_3, Rule_4, Rule_5};
```

Furthermore, it is also possible that arguments both in favour as well as against the same literal may be constructed given a context x and a prioritised knowledge base k. Indeed, consider the following knowledge base, where the predicate symbol `is_checkmate(Move)` indicates that move `Move` is a checkmate move:

```
Rule_7 :: plays_as(Colour), move_played_by(Move,Colour),
is_checkmate(Move) implies suggest(Move);
Rule_6 :: plays_as(Colour), game_phase(opening),
moves(Move,queen), move_played_by(Move,Colour)
implies –suggest(Move);
Op :: move_count(X), ?<(X,11) implies game_phase(opening);
```

Also, consider the following significant fragment of a context which describes the position shown in Figure 8 (black to move):

```
plays_as(black);
move_played_by(d8h4,black);
```

```
moves(d8h4,queen);
move_count(2);
is_checkmate(d8h4);
```

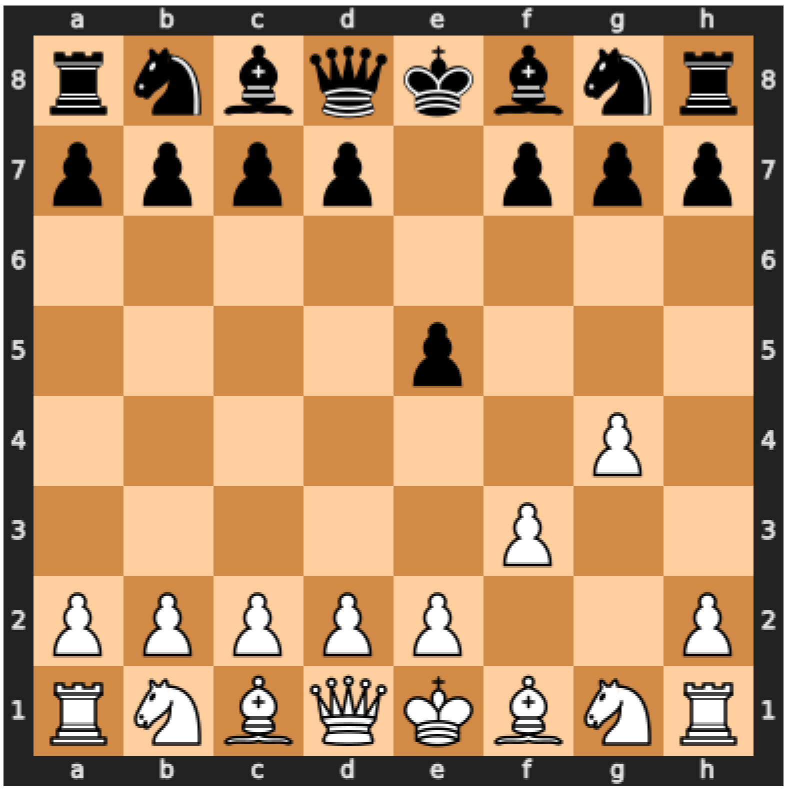


Figure 8: Black to move and win.

Based on the above, we can construct the following argument supporting `suggest(d8h4)`:

```
A = {is_checkmate(d8h4), move_played_by(d8h4,black),
plays_as(black), Rule_7};
```

But, again based on the above context and knowledge base, we can also construct an argument for `-suggest(d8h4)`:

```
B = {move_played_by(d8h4,black), plays_as(black),
move_count(2), moves(d8h4,queen), Op, Rule_6};
```

3.3.2 Defining an Argumentation Framework

Having defined and thoroughly presented the notion of argument in the context of Machine Coaching, we will now proceed in defining an argumentation framework as declared in (Dung, 1995: 325-334). To do so we need to define an ordered pair $(Args, Att)$ where *Args* is a set of arguments and *Att* is a binary *attack* relation on *Args*, i.e. $Att \subseteq Args \times Args$ (Dung, 1995: 326). For the rest of this subsection, let x be a context and $k = (\rho, \prec)$ prioritised knowledge base.

As far as the set of all arguments, $Args$, is concerned, we let $Args$ be the set of all arguments in x under k (Michael, 2019: 83). As far as the Att relation of attacks between arguments is concerned, following (Michael, 2019: 83) we will make the following choices under the ASPIC+ framework (Prakken, 2010: 96-114):

- We choose the context x as an *axiom set* (Prakken, 2010: 98) – i.e. as a set of premises against which it is not possible to argue. This means that arguments cannot be attacked on their premises, so, contextual information is considered to be always true. Intuitively, this expresses the idea that facts that describe a certain situation cannot be disputed – e.g., in our context, a board position cannot be disputed by any of the players, as well as the same applies to the game's rules.
- We choose all of the rules in our knowledge base to be *defeasible*, that is, given that the assumptions of a rule all hold then the rule's head *may hold* (Prakken, 2010: 97). This, defeasibility is one of the factors that is intended to capture the *refutable* character of human reasoning, since, in most everyday situations – as well as in chess – little are the chances that a rule holds absolutely, on every occasion. Instead, most rules are dependent on the wider context into which they are applied.
- We choose rebutting attacks between arguments (Prakken, 2010: 101). In more detail, an argument B rebuts another argument A when argument B leads, possibly among others, to a conclusion that contradicts some of the conclusions of A. In a sense, allowing for rebutting attacks means that for an argument A to hold, no other *counter-argument* may be triggered by a given context, so each single conclusion of A is accepted in that given context.
- We also order arguments according to the *last-link principle* (Prakken, 2010: 109). That is, we say that an argument A is preferred over another argument B if the last rule of A is preferred over the last rule of B according to the priority relation $\prec$.

As a result, we could say that an argument A supporting g attacks another argument B which contains a rule r with head $-g$ (i.e. $(A, B) \in Att$) in a context x under a knowledge base $k = (\rho, \prec)$ if one of the following conditions is true:

1. $g \in x$ – i.e. the conclusion of argument A, g, is an indisputable fact;
2. $t \nprec r$, where t is the crown rule of argument A – i.e. argument A is not less preferred than the sub-argument B' of B which has r as its crown rule.

We will now present some examples so as to clarify the above notions. To begin with, consider the two last arguments A and B presented in the previous subsection (3.3.1), namely:

```
A = {is_checkmate(d8h4), move_played_by(d8h4,black),
plays_as(black), Rule_7};
B = {move_played_by(d8h4,black), plays_as(black),
move_count(2), moves(d8h4,queen), Op, Rule_6};
```

As we can see, A attacks B but not the other way around. Indeed, `Rule_7` of argument A attacks argument B on `Rule_6`. However, the same does not hold for B, as it may contain a rule that leads to a conflict with A, nevertheless `Rule_7` is preferred over `Rule_6` – since `Rule_7` appears above `Rule_6` in the knowledge base.

As another example, consider the following knowledge base[39] which contains `Op`, – where, as we have previously declared, rules are listed by descending priority:

```
Sac :: plays_as(Colour1), -plays_as(Colour2),
move_played_by(Move,Coulour1), to_square(Move,Square1),
controls(Square2,Piece,Colour2,Square1) implies –suggest(Move);
Cap :: plays_as(Colour), move_played_by(Move,Colour),
is_capture(Move) implies suggest(Move);
Noq :: plays_as(Colour), move_played_by(Move,Colour),
game_phase(opening), moves(Move,queen) implies –suggest(Move);
Op :: move_count(X), ?<(X,11) implies game_phase(opening);
```

[39] Any predicates that have not been presented yet, are intended to be interpreted as indicated by their names – e.g. `is_capture(Move)` is intended to be interpreted a: move `Move` is a capture move.

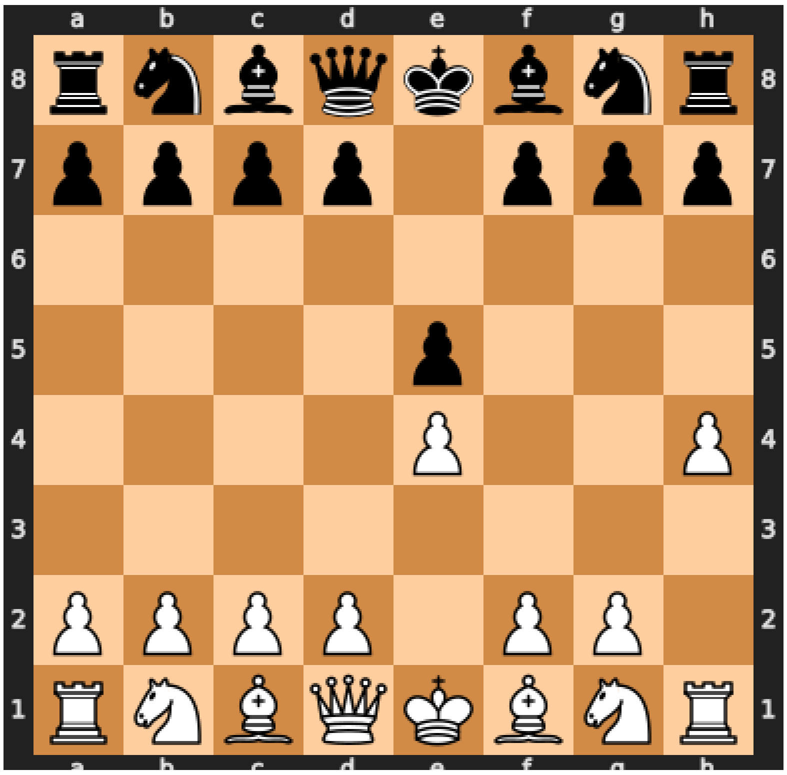


Figure 9: A not so difficult choice.

Also, consider the following fragment of a context which describes the position shown in Figure 9:

```
plays_as(black);
-plays_as(white);
move_played_by(d8h4,black);
to_square(d8h4,h4);
is_capture(d8h4);
move_count(2);
moves(d8h4,queen);
controls(h1,rook,white,h4);
```

In the above context we can detect three interesting arguments, let A, B, C:

```
A = {move_count(2), move_played_by(d8h4,black),
plays_as(black), moves(d8h4,queen), Op, Noq};
B = {move_played_by(d8h4,black), plays_as(black),
is_capture(d8h4), Cap};
C = {plays_as(black), move_played_by(d8h4,black),
-plays_as(white), to_square(d8h4,h4),
controls(h1,rook,white,h4), Sac};
```

Among the three arguments described above, given that $Noq \prec Cap \prec Sac$, we have an attack from argument B to argument A – since $Noq \prec Cap$ – as well as another attack from argument C to argument A – since $Cap \prec Sac$.

3.3.3 Grounded Extension of an Argumentation Framework

As our next step, we would like to investigate what one could safely deduce given a prioritised knowledge base k and a context x as well as whether there is an efficient algorithm that computes the set of inferred literals. A possible answer to this question, is to adopt, as in (Michael, 2019: 83-84), Dung's Grounded extension of an argumentation framework which expresses the grounded (skeptical) semantics as declared in (Dung, 1995: 329).

We will need some definitions before we proceed to the definition of an argumentation framework's grounded extension. Given an argument A and a set of arguments, S, we will say that A is **acceptable** with respect to S if and only if for any other argument B that attacks A there exists another argument $C \in S$ such that C attacks B (Dung, 1995: 326). That is, S provides enough arguments so as to defend all attacks against[40] A.

Let now $\mathcal{A} = (Args, Att)$ be an argumentation framework. Then, the **characteristic function** of $\mathcal{A}$ is a function $F_{\mathcal{A}}: P(\mathcal{A}) \rightarrow P(\mathcal{A})$, where by $P(X)$ we denote the powerset of a set X, such that $F_{\mathcal{A}}(S) = \{A \in Args : A \text{ is acceptable with respect to } S\}$ (Dung, 1995: 328-329). Using $F_{\mathcal{A}}$ the **grounded extension** $G_{\mathcal{A}}$ of $\mathcal{A}$ is defined as the first, with respect to set inclusion, fixed point of $F_{\mathcal{A}}$ (Dung, 1995: 329).

Let us clarify the above definition by describing a process in which one may find the grounded extension of an argumentation framework. Let, as above, $\mathcal{A} = (Args, Att)$ denote an argumentation framework and also let F be its characteristic function. We start from the empty set, $\emptyset$, and proceeds as follows:

1. If $F(\emptyset) = \emptyset$ we have found the first fixed point of F so $G_{\mathcal{A}} = \emptyset$ and we are done.
2. If $F(\emptyset) \neq \emptyset$ then we proceed to finding $F^2(\emptyset) \coloneqq F(F(\emptyset))$. Again, in case $F^2(\emptyset) = F(\emptyset)$ then $G_{\mathcal{A}} = F(\emptyset)$ and we are done.
3. If $F^2(\emptyset) \neq F(\emptyset)$ then we proceed to computing $F^3(\emptyset)$ and so on.

[40] Intuitively, this corresponds in some way to having a view A on some certain topic as well as sufficient evidence and/or knowledge – enclosed in S – so as effectively argue against any attempt to dispute that view.

Given that F preserves set inclusion[41], so at each step we either stop to some fixed point or proceed with a larger set of arguments, it is guaranteed that we can compute the grounded extension[42], $G_{\mathcal{A}}$, of $\mathcal{A}$.

Let us now discuss the intuition behind grounded semantics, as defined above[43]. At first, we compute $F(\emptyset)$ which corresponds to the set of arguments $A \in Args$ that are acceptable by $\emptyset$. This intuitively describes inferences that need no further support to defend against attacks from any other arguments – i.e. there is no argument B that attacks any argument A in $F(\emptyset)$. Should $F(\emptyset) = \emptyset$ then we cannot proceed any further since $\emptyset = F(\emptyset) = F\big(F(\emptyset)\big) = \cdots = F^n(\emptyset) = \cdots$ and, consequently, there is no argument included in $G_{\mathcal{A}}$.

However, in case $F(\emptyset) \neq \emptyset$ we proceed by computing $F(F(\emptyset))$, that is, the set of arguments that, even if attacked by some other arguments, can be defended by arguments that, themselves, are not attacked by any other argument. Hence, we can also guarantee that such inferences can be convincingly trusted. In the same fashion as above, if $F\big(F(\emptyset)\big) = F(\emptyset)$ then we have constructed tour argumentations framework's grounded extension while, in case the above does not hold, we proceed by computing $F\Big(F\big(F(\emptyset)\big)\Big) = F^3(\emptyset)$. As with the previous case of $F^2(\emptyset)$, we make a step forward towards arguments that, apart from the previous two cases, may be attacked by arguments that are themselves attacked by other arguments in $F(\emptyset)$. We continue in the same way until we reach F's first fixed point.

[41] Indeed, let $S \subseteq T$ and $A \in F(S)$. Then, since any attack against A is defended by an argument from S the same applies for T, with leads to $A \in F(T)$ and, consequently, to $F(S) \subseteq F(T)$.

[42] Indeed, to prove this, it suffices to prove that F has always at least one fixed point. To prove this, let $C \coloneqq \{S \subseteq Args\colon S \subseteq F(S)\}$. Observe that $C \neq \emptyset$ since $\emptyset \in C$ and let $T \coloneqq \bigcup_{S\in C} S$ – which is well defined, since $C \neq \emptyset$. At first, we will prove that $T \subseteq F(T)$. Indeed, since $S \subseteq T$ for any $S \in C$, since F preserves set inclusion we get $F(S) \subseteq F(T)$ for any $S \in C$. As a result, $T = \bigcup_{S\in C} S \subseteq F(T)$ and, since $T = \sup C$ – this is easy to prove – we obtain $T \subseteq F(T)$. For the inverse inclusion, observe that, since $T \subseteq F(T)$, since F preserves set inclusion, we also have $F(T) \subseteq F(F(T))$. Hence, by C's definition, we obtain $F(T) \in C$ so, since $T = \sup C$ we also have $F(T) \subseteq T$. So, $T = F(T)$ and, as a result, T is a fixed point of F.

[43] They are also referred to as *skeptical* semantics in (Dung, 1995: 329), which unveils that the intention is, as we will explain, to capture a set of inferences that can be *safely* proved under a knowledge base k in context x.

At a higher level, we could describe the above process of constructing the grounded extension of an argumentation framework as an iterative process in which we start by arguments that need not be defended by other arguments and then gradually add arguments such that they (i.e. the new arguments) can be supported against other attacks by arguments that we have already accepted – by counter-attacking those attacking arguments. In the above context, the terms grounded and skeptical seem plausible.

The above property of "groundedness" of grounded semantics is evidently a desired property when it comes to argumentation, in the sense that it allows for inferences to be "safely" conducted. Furthermore, skeptical semantics have been chosen for two more reasons: (i) they lead to a single model[44] which conforms to results from other fields about the emergence of a single model in human reasoning (Stenning, Lambalgen, 2012: 125-128); (ii) the grounded extension of an argumentation framework can be efficiently computed (Michael, 2019: 83-84).

Having explained at an abstract level the motivating ideas behind the definition of an argumentation framework's grounded extension as well as our own motivation for choosing it in our setting, we shall present an example of a grounded extension in our context – i.e. chess. Consider again the following arguments $Args \coloneqq \{A, B, C\}$ where:

```
A = {move_count(2), move_played_by(d8h4,black),
plays_as(black), moves(d8h4,queen), Op, Noq};
B = {move_count(2), move_played_by(d8h4,black),
plays_as(black), is_capture(d8h4), Cap};
C = {plays_as(black), move_played_by(d8h4,black),
-plays_as(white), to_square(d8h4,h4),
controls(h1,rook,white,h4), Sac};
```

Also, let the attack relation be defined as follows: $Att \coloneqq \{(C, B), (B, A)\}$. Then we compute the grounded extension of the argumentation framework $(Args, Att)$ as indicated below:

[44] Indeed, given that the grounded extension of an argumentation framework $\mathcal{A}$ is defined as the *least* fixed point of its characteristic function $F_{\mathcal{A}}$, it is by its definition, unique.

- At first, we compute $F_{\mathcal{A}}(\emptyset)$. Evidently, given our attack relation, the only argument that can stand unsupported by other arguments is argument C so $F_{\mathcal{A}}(\emptyset) = \{C\}$. Since $F_{\mathcal{A}}(\emptyset) \neq \emptyset$ we need to proceed further.
- Next, we compute $F_{\mathcal{A}}^2(\emptyset) = F_{\mathcal{A}}(F_{\mathcal{A}}(\emptyset))$. Since $C \in F_{\mathcal{A}}(\emptyset)$ and $F_{\mathcal{A}}$ preserves set inclusion, we also have $C \in F_{\mathcal{A}}^2(\emptyset)$. Moreover, since B's attack on A is defended by $C \in F_{\mathcal{A}}(\emptyset)$ we obtain that $A \in F_{\mathcal{A}}^2(\emptyset)$. Also, observe that $B \notin F_{\mathcal{A}}^2(\emptyset)$ since it is being attacked by C.
- Next, we compute $F_{\mathcal{A}}^3(\emptyset) = F_{\mathcal{A}}\left(F_{\mathcal{A}}^2(\emptyset)\right)$. Observe that B, the only argument not already included in the framework's grounded extension cannot be included since B is being attacked by C. Given the fact that $F_{\mathcal{A}}$ preserves set inclusion, we obtain that $F_{\mathcal{A}}^3(\emptyset) = F_{\mathcal{A}}^2(\emptyset)$, hence, $F_{\mathcal{A}}$'s first fixed point is $\{A, C\}$.

As a result, the corresponding grounded extension of the given argumentation framework $(Args, Att)$ is $\{A, C\}$ which reflects also our intuition that, given the above three arguments as well as their attack relation, we cannot convincingly argue in favour of B while, as shown above, we can when it comes to$A$ and/or C.

3.3.4 Efficient Computation of an Argumentation Framework's Grounded Extension

While the computation of the grounded extension of an argumentation framework is efficient in terms of the number of arguments included in it, this does not hold when it comes to the sizes of the knowledge base $k = (\rho, \prec)$ and the context x from which the argumentation framework has been constructed (Michael, 2019: 84). Indeed, computation time may well be exponential in terms of the number $n = |\rho|$ of rules included in k as well as in the number of literals, $\lambda = |x|$. To demonstrate this, consider the following knowledge base:

```
R_1 :: a1(X1) implies a2(X1);
R_2 :: a2(X1), b(X2) implies a3(X1,X2);
R_3 :: a3(X1,X2), b(X3) implies a4(X1,X2,X3);
…
R_n :: an(X1,X2,…,Xn-1) b(Xn) implies an+1(X1,X2,…,Xn);
```

Also, consider the following context:

```
a1(1); a1(0); b(1); b(0);
```

Given the above, we can construct the following arguments, possibly among others – where, by $A \rightarrow g$ we denote that argument A supports some literal g:

```
A0 = {a1(0), R_1} -> a2(0);
A1 = {a1(1), R_1} -> a2(1);
A00 = {a1(0), b(0), R_1, R_2} -> a3(0,0);
A01 = {a1(0), b(1), R_1, R_2} -> a3(0,1);
A10 = {a1(1), b(0), R_1, R_2} -> a3(1,0);
A11 = {a1(1), b(1), R_1, R_2} -> a3(1,1);
A000 = {a1(0), b(0), R_1, R_2, R_3} -> a4(0,0,0);
…
A111…1 = {a1(1), b(1), R_1, R_2, …, R_n} -> an(1,1,…,1);
```

Evidently, the above arguments are $2 + 2^2 + 2^3 + \cdots + 2^{n-1} = 2(2^{n-1} - 1) = 2^n - 2$, so, indeed, the corresponding grounded extension needs time at least exponential in n to be computed. Observe in the above how the size of the context, λ, did not play any role in the exponential blowup of the representation of the argumentation framework – the same context containing four (4) literals suits for any value of $n \in \mathbb{Z}_{>0}$. As a result, we shall look for a better representation for a grounded extension since using arguments may lead to exponentially large computations.

A way to efficiently compute the grounded extension of an argumentation framework $\mathcal{A}_k(x)$ induced by a prioritised knowledge base k and a context x is presented in (Michael, 2019: 84). To do so, we introduce at first the **dual representation** of a grounded extension being the tuple $(x, \rho_k(x))$ where $\rho_k(x)$ is the set of rules included in the arguments of the grounded extension of an argumentation framework $\mathcal{A}_k(x)$. As demonstrated in (Michael, 2019: 84), dual representations are to a one-to-one correspondence with grounded extensions in the sense that any argument contained in $\mathcal{A}_k(x)$ can be reconstructed using the context x as well as the set of rules $\rho_k(x)$ and, conversely, any argument in x under $\rho_k(x)$ is an argument included in the grounded extension of $\mathcal{A}_k(x)$.

Next, we present an algorithm which efficiently computes the dual representation of a grounded extension of an argumentation framework $\mathcal{A}_k(x)$ with respect to the number of rules included in $\rho_k(x)$ as well as the context's x size.

The algorithm is the following one:

1. At first, given k and x, construct the inference graph G of k – i.e. the graph that includes anything that may be inferred from x and k using modus ponens.
2. Then, construct a list, $markedLiterals$, which initially contains all the literals included in the given context x, as well as a $markedRules$ list which is initially empty.
3. Loop through the following steps until no new literal is added in the $markedLiterals$ list:
    a. Remove from the inference graph any literals that conflict with literals in $markedLiterals$.
    b. For each argument that has remained in the inference graph G keep only its crown rule[45].
    c. Add any rule whose body literals are **all** included in $markedLiterals$ and which also is preferred against to any other conflicting rule with respect to k's priority relation, $\prec$, to the $markedRules$ list.
    d. For each rule in $markedRules$ add its head to $markedliterals$.

We shall now, as with previously introduced notions, explain the above algorithm by providing an example. Let us consider again the three arguments A, B, C of our previous example as well as the corresponding attack relation. Given the set of all rules contained in them, ρ, as well as the literals of context x we can construct the corresponding inference graph G as shown in Figure 10. Also, let $markedLiterals$ and $markedRules$ be the two lists of marked literals and marked rules accordingly. Then:

1. At first, we initialise $markedLiterals$ by adding all context literals to it. Thus:
    ```
    markedLiterals = {plays_as(black), -plays_as(white),
    move_played_by(d8h4,black), to_square(d8h4,h4),
    is_capture(d8h4), move_count(2), moves(d8h4,queen),
    controls(h1,rook,white,h4)};
    ```
2. Next, we proceed to step 3b – since no literals conflict with marked ones at the moment. For the time being, all rules are maintained since all may be crown rules for a suitably selected argument[46].

[45] Remember that and for an argument A for g, the argument's *crown rule* has been defined to be the (single) rule that has g as its head, provided that such a rule exists.

3. Now, as we see in Figure 10, given our list of marked literals, rules `Op` as well as `Cap` and `Sac` have all their body literals marked, so we add them to `markedRules` – there are no conflicts to resolve here. We also add the rules' heads `-suggest(d8h4)` and `game_phase(opening)` to `markedLiterals`.
4. We remove `suggest(d8h4)` from G since it conflicts with some marked literal.
5. Now, we see that the only remaining rule, Noq, has all its body literals marked and there is no other rule left in G conflicting with Noq and is also preferred against it, so we shall not include Noq in `markedRules` and, consequently, we add no new literals to `markedLiterals`.
6. Since no new literals were added during the previous iteration, the process terminates.

So, we see that indeed, we can reconstruct using literals in `markedLiterals` and rules in `markedRules` all arguments found in the grounded model of our argumentation framework.

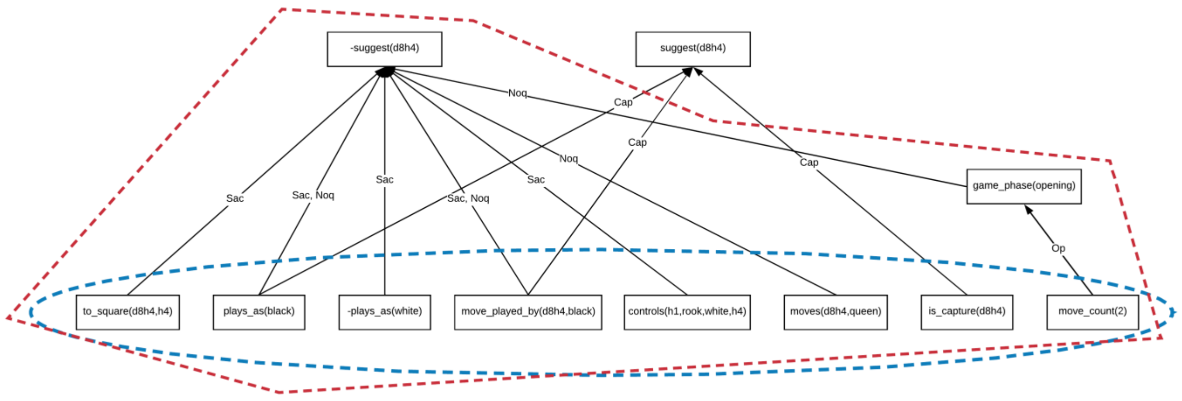


Figure 10: The inference graph of our example. The blue ellipse includes all context literals while the red polygon all marked ones.

3.3.5 Learning in the context of Machine Coaching

Up to now, we have defined how argumentation is conducted within the scope of Machine Coaching, as well as a language within which argumentation is conducted. It remains to describe the learning process. In general, learning in the context of Machine Coaching is defined (Michael, 2019: 84) using Probably Approximately Correct (PAC)

[46] As far as rules `Noq`, `Cap` and `Sac` are concerned, they are evidently included since they are crown rules for the arguments A, B and C respectively. As for rule `Op`, we can construct an argument with it as crown rule, namely `D = {move_count(2), Op}`.

semantics, as described in (Valiant, 1984: 1136-1137). Namely, we say that an algorithm is a **probably approximately conformant learner** of some feedback class[47] $F = F(I, O, A)$ using a hypothesis class[48] H if for every $\delta, \varepsilon \in (0,1]$, every probability distribution D over inputs in I of size n and every $f \in F$ of size s it can, given access to δ, ε, F, iteratively invoke the following process:

1. Get an input $x \in I$ either randomly under D or by actively choosing it.
2. Select an output from $y \in O$.
3. Ask for some advice $f(x, y)$.

After time at most $g\left(\frac{1}{\delta}, \frac{1}{\varepsilon}, n, s\right)$ the algorithm terminates and returns, except with probability δ, a hypothesis $h \in H$ such that $f\big(x, h(x)\big) = noAdvice$ except with probability ε (Michael, 2019: 84). We will also say that the algorithm is an efficient conformant learner if g is of polynomial complexity with respect to its parameters.

Explaining the above definition, we could roughly say that, given any desired probability of failure $\delta \in (0,1]$ as well as any desired probability of accuracy $\varepsilon \in (0,1]$ we can say that there exists an algorithm capable of capturing a theory about something – e.g. chess – by a process during which the algorithm makes predictions about examples it encounters and receives pieces of advice about them (i.e. its predictions). Then, given that the algorithm terminates at some time – dependent on the two given probabilities as well as the size of each example and the corresponding pieces of advice – by providing a model of our theory which is accurate allowing for errors to occur with probability ε. Additionally, the algorithm may not yield the above model of our theory with probability at most δ.

[47] A feedback class is defined to be a set of feedback functions $F \coloneqq \{f: I \times O \to A\}$, where I, O, A are the input, output and advice sets respectively. That is, F contains functions which, given an input – i.e a learning example – and an output – i.e. the machine's prediction – return an advice. We also demand, as in (Michael, 2019: 84) that for each input $x \in I$ there exists at least one $y \in O$ such that $f(x, y) = noAdvice$ – i.e. there exists at least one acceptable output for any input.

[48] A hypothesis class is defined as a set of functions $H \coloneqq \{h: I \to O\}$ which, given an input $x \in I$ returns an output $h(x) \in O$ – in other words, given a learning example it returns a prediction about it.

The above is a quite general definition of learnability, however, we shall restrict ourselves to discussing a certain algorithm presented in (Michael, 2019: 84-85) which describes a learning protocol under which learning is guaranteed to be efficient[49]. Under this protocol, we choose a specific feedback class F described as follows – let k denote a knowledge base which describes the theory we aim to learn as well as x denote some arbitrary context:

- A predicted rule will be considered *unrecognised* if it is not encountered in k – i.e. if it is not some rule in our theory.
- A predicted rule will be considered *superfluous* if it does not contribute to any argument in x under k.
- A rule will be considered *incomplete* if it does not appear in a prediction while it is included in k and its inclusion would lead to some additional argument from the machine.
- An argument will be considered *indefensible* if there exists no other argument in x under k that attacks it while it attacks some argument in the machine's prediction.
- No response, if none of the above holds.

Given the above feedback class as well as the condition that conflicting rules in k are linearly order with respect to $\prec$, then, as proved in (Michael, 2019: 85) the following algorithm is a probably approximately conformant learner:

1. Let $k = \emptyset$ be the machine's initial knowledge base.
2. For each randomly chosen input x:
   a. Predict the corresponding dual representation y.
   b. Receive the user's advice $f(x, y)$ according to the above protocol.
   c. Delete any rules considered superfluous or unrecognised, add rules that are labeled as incomplete or lead to counter-arguments by the user with higher priority than any existing rule in the knowledge base, k.
3. Repeat the above until the user provides no response for m consecutive cycles, where m is polynomial with respect to the aforementioned parameters – see the definition of a probably approximately conformant learner above.

[49] By efficient, we mean that learning is conducted in polynomial time with respect to various PAC related parameters.

Observe how the above learning protocol accommodates all the desired functionality as described in chapter 3, section 3.1, where we demanded for the capability of a user to provide the machine with counter-argumentation with respect to suggestions and explanations provided by the machine.

Chapter 4
Implementation

In this chapter, the implementation of the chess coach is presented. The structure of the chapter is as follows: (i) in section 4.1, we present the implementation of our first order language as well as make some crucial remarks regarding several features; (ii) in section 4.2, we present the implementation of the reasoning engine; (iii) in section 4.3, we present some chess-specific features regarding our final chess bot and; (iv) in section 4.4, we present the designed Graphical User Interface (GUI) that accommodates all the above functionalities as well as our implementation of the interaction protocol presented in chapter 3.

4.1 Implementing a First Order Language

As we have seen in the previous chapter, both human-machine interaction as well as reasoning in the context of Machine coaching do rely on a first order language that efficiently captures the semantics of the domain of application. As a result, the way in which the aforementioned language will be implemented is expected to affect our work at a significant extent, as far as its technical aspects are concerned.

We aimed towards a generic implementation, that is, our language is not domain-specific and can accommodate any other task that would require a similar functionality. Moreover, as one may see throughout the rest of this section, the language could be extended to allow for general first order reasoning by also implementing function symbols as well as altering the way unification is conducted in our context – i.e. without taking into account function symbols. In the rest of this section we will discuss not only the way the language presented in chapter 3, section 3.2 has been implemented but also discuss some subtle points of our implementation.

4.1.1 Basic implementation of the language

In this subsection we will discuss the way in which the language described above has been implemented in java. The implementation is, as we will see, quite generic in the sense that by additionally implementing function symbols, any algorithm that needs a typical first order language could utilise its structures. Also, this language will be used, as presented in chapter 3, section 3.2, as a base upon which any Machine Coaching related algorithm, as presented in (Michael, 2019: 83), will be implemented – also in java.

Given the structure of the language of Machine Coaching as described above - as well as, in general, the structure of any first order language - an object-oriented approach seemed to be the most appropriate for our task. Under such a view, each entity of our language - i.e. each class of symbols - would be considered a separate object, allowing for it to have any functions related to its object specific functionalities, which would naturally lead into more efficient and intuitive coding, among others. Below, we present how each of the components of a first order language, as defined in chapter 3, section 3.2, has been implemented:

1. First order predicates are described by a `Predicate` class which includes the following fields: (i) an integer field, `arity`, which corresponds to the predicate's arity; (ii) a String field, `name`, which corresponds to the predicate's name; (iii) a list of variables field, `variableList`, which corresponds to the predicate's list of variables/arguments.
2. Constants are considered as predicates of zero arity (`arity=0`). Hence, each constant is an instance of `Predicate` class with `arity=0` and its variable list equal to the `null` object[50].
3. Variables are described by a `Variable` class which has as fields: (i) a string field, `name`, which corresponds to the variable's name; (ii) a Predicate field, `value`, which corresponds to the variable's value, in case it is assigned (`null` otherwise). Evidently, since our target is to implement a first order language, the `value` field is not intended to take any other values other than zero arity predicates – i.e. constants, as explained above[51].

[50] While this contradicts a usual practice in first order logic of thinking of constants as *function* symbols of zero arity – e.g. (Enderton, 2012: 80-81). Our intention is to also allow for propositional knowledge bases and contexts to be expressed in our language, even if this is not relevant to our specific domain of application – i.e. chess.

[51] Actually, there is no restriction when it comes to the implementation of the `Variable` class, that prohibits assigning a predicate of positive arity to a variable as its value. However, the way

4. Literals are described by a `Literal` class which has as fields: (i) a Predicate field, `atom`, which corresponds to the literal's predicate; (ii) a Boolean field, `sign`, which corresponds to the literal's sign – i.e. `sign=false` means that the literal is negative while `sign=true` means that the literal is positive.
5. Rules are described by a `Rule` class which has as fields: (i) a String field, `name`, which corresponds to the rule's name; (ii) a list of literals field, `body`, which corresponds to the rule's body; (iii) a literal field, `head`, which corresponds to the rule's head.
6. Knowledge bases are described by a `KnowledgeBase` class which has as fields: (i) a list of rules field, `rules`, which contains the knowledge base's rules; (ii) a CHECK DATA STRUCTURE field, `priorities`, which corresponds to the priority relation defined between conflicting rules of the knowledge base.
7. Contexts are described by a `Context` class which has a single list of literals field, CHECK `literals`, which corresponds to the context's literals. This class was included mostly in order to facilitate the design of certain context-related java methods – since, as one may observe, a list of literals itself need not be a separate class, should no other additional functionality be introduced in it[52].

4.1.2 Remarks about the language's implementation

We shall discuss now some remarkable features of the designed language. To begin with, negation in the context of our language is treated as classical negation, in contrast to other approaches were negation is treated in the context of some sort of close world assumption – i.e., negation as failure.

Next, our implementation takes into account the fact that the language is intended to have an *equality symbol*, that is, a binary predicate symbol denoted by `?=(X,Y)` which is always interpreted as the congruence relation in any universe. As a result, for any constant `c` in our universe, we have that `?=(c,c)` while no other instances[53] of this

in which variables are treated in the rest of the code ensures that only zero arity predicates – i.e. constants – are used as values for variables.

[52] As one may observe, apart from the absence of a separate constant class, the rest of our implementation reflects the structure of a typical first order language with no function symbols, as described in chapter 3, section 3.2.

[53] By saying "instances of some predicate" we refer to any literal built from that predicate, either positive or negative

predicate are true. Note that, even if the above holds for any constant of our universe, it is not explicitly included in any context generated. Instead, we have adopted a different methodology so as to avoid adding numerous trivial instances of `?=(·,·)` in all contexts and, as a result, leading to an unnecessary increment of the reasoning algorithm's execution time.

The desired behaviour, as described above, has been achieved in the following way:

- In case both arguments of `?=(·,·)` are constants of our universe, let `a` and `b` respectively, then the `?=(a,b)` holds exactly when constants `a` and `b` coincide – i.e. they refer to the same entity in our universe. In terms of our implementation, it means that all fields of the two predicate objects that correspond to `a` and `b` are equal.
- In case exactly one of the two arguments of `?=(·,·)` is a constant, let `a`, while the other is an unassigned variable, let `X`, then we one of the following takes place:
    - If `?=(X,a)` – or `?=(a,X)` – is the only literal in the rule's body[54] then `X` is unified with `a`, yielding the substitution `{X -> a}`.
    - If `?=(X,a)` – or `?=(a,X)` – is not the only literal in the rule's body then, given a substitution `S` that includes all variables in any other (non-numerical) literal[55] in the rule's body, should the substitution contain an assignment for `X`, then `X` is assigned with the corresponding value, let `b`, and `?=(b,a)` is evaluated as in the first case. However, should `X` do not appear in `S`, `X` is unified with `a` and `{X -> a}` is added to `S`.
- In case none of the arguments of `?=(·,·)` is a constant, hence both are unassigned variables, let `X` and `Y`, similarly to the previous case, one of the following takes place:
    - If `?=(X,Y)` is the only literal in the rule's body, then `X` is unified with `Y` – see next section about more details on how this would affect inference[56].

[54] It is not allowed to use ?=(·,·) in a rule's head, given that ?=(·,·) is *defined* to express the congruence relation between entities of any specified universe and, as a result, it cannot be overridden.

[55] See the next remark for more details about numerical literals and the way they have been implemented in our language.

[56] Under normal conditions, given that all literals are variable-free in our setting, such an occasion is not expected to occur.

- If `?=(X,Y)` is not the only literal in the rule's body, then, given a substitution `S` that includes all variables in any other (non-numerical) literal in the rule's body, we have three further cases: (i) if `S` does not include `X` nor `Y` then `X` is unified with `Y` as above; (ii) if `S` includes one of the two but not the other, then the unassigned variable, without loss of generality, let `X`, is assigned with the same value as `Y` and `{X -> c}` is added to `S`, where `c` is the value of `Y`; (iii) if both variables are included in `S` then, as in the first case, `?=(X,Y)` holds if and only if both variables are substituted by the same constant symbol in `S`.

We will refer again to the way `?=(·,·)` has been defined as well as to how it affects the implementation of the reasoning algorithm described in chapter 3, section 3.3, in the next section.

Our next remark is related to the execution of usual numerical operations, the introduction of several mathematical functions as well as the definition of the *less than* relation between numbers (<) in our language. While addition, multiplication and the usual order relation between numbers may be defined in terms of our (first order) language, we preferred to allow for such expressions to be computed externally, using built-in functions provided by java, so as to avoid unnecessary long arguments[57] where a large part of them would be dedicated to proving e.g. that $3 + 7 < 98 - 5$.

At this point we should also note that equality between numerical expressions in the context of our language is also captured by the already defined `?=(·,·)` predicate. As far as equality between numerical constants is concerned, there was no need to make any changes in the algorithm we have presented above, since each number is considered

[57] Indeed, in order to do so, one way would be to include two ternary predicates `+(X,Y,Z)` and `·(X,Y,Z)` with the intended interpretation being that `Z` is the sum/product of `X` and `Y` respectively, as well as a binary predicate `<(X,Y)` denoting the usual order relation between numbers. Moreover, we would need to provide all axioms describing a totally ordered field as well sufficiently many constant symbols describing "enough" numbers, which is evidently insufficient. Even restricting ourselves to natural numbers, it would still be required to allow for a function symbol `succ(X)` denoting the successor of `X` – or, equivalently, a predicate symbol `succ(X,Y)` denoting that `Y` is the successor of `X` – as well as include `<(0,S(0))` in every context – so as to allow for any comparison to be computed by the rest of the order axioms.

a constant of our universe[58]. However, in the case non-constant numerical expressions, several changes were needed to be made. We will describe in full detail this part of our implementation in section 4.2, after having presented our implementation of the reasoning algorithm described in chapter 3, section 3.3.

We should also note that in the context of our language, any mathematical expression should be written in usual *infix* notation, using parentheses were needed to determine priority among mathematical operators and functions. So, for example, an expression such as the following one:

$$\sqrt{x^2 + 3x + 11} - \sin x$$

is written as follows, using infix notation[59]:

```
sqrt(X^2 + 3*X + 11) - sin(X)
```

At last, all mathematical operations and functions[60] that are allowed in our language as of the time this thesis was written are presented in Table 1.

Name	Symbol	Example
Addition	`+`	`32+5`
Multiplication	`*`	`4*6`
Subtraction	`-`	`4-56`
Division	`/`	`7/8`
Integer division (remainder)	`mod`	`14 mod 3`

[58] This is achieved during parsing, following our language's syntax, as defined in chapter 3, section 3.2. Namely, during parsing a predicate's arguments, any string starting with a capital letter of the latin alphabet is considered to denote a variable, while any other string sequence that starts either with a lowercase letter or a numerical digit (including the minus symbol, $-$) is considered as a constant symbol.

[59] Space characters between symbols indicating nothing and were inserted only to facilitate reading.

[60] As far as mathematical constants are concerned, the only constant recognised during parsing, apart from real numbers represented in usual decimal notation, is π, as of the time this thesis was written. Euler's number e was not considered necessary to be included since it could be accessed by using the exponential function – `exp(1)`.

Integer division (quotient)	`div`	`5 div 2`
Exponentiation	`^`	`5^3`
Sine function	`sin`	`sin(4)`
Cosine function	`cos`	`cos(pi)`
Tangent function	`tan`	`tan(2*pi)`
Cotangent function	`cot`	`cot(-6)`
Exponential function	`exp`	`exp(2*6)`
Natural logarithm function	`log`	`log(pi)`
Base-2 logarithm function	`log2`	`log2(8)`

Table 1: Mathematical operations and functions recognised by the language.

4.2 Reasoning and Argumentation

In this subsection we will discuss how the algorithm presented in chapter 3, section 3.3, for the construction of the dual representation of the grounded extension of a contextualised argumentation framework $\mathcal{A}_k(x)$ has been implemented, utilising all the above structures of our first order language.

4.2.1 Implementation

In order to implement the algorithm as described in chapter 3, section 3.3 as well as in (Michael, 2019: 84), we introduce the following classes:

1. A `Substitution` class which represents a substitution of variables. It consists of a java Hash Map field, `substitutions`, which maps variables to constants[61] – i.e. instances of the `Variable` class to instances of the `Predicate` class with zero arity (`arity=0`). Apart from that, it also accommodates several substitution-specific methods that are useful when it comes to First Order Forward Chaining – e.g. applying a substitution to a given literal, which is utilized mostly in the unification algorithm.
2. An `InferenceGraph` class which has the following fields: (i) a list of literals field, `nodes`, which corresponds to the graph's nodes – which, in the current setting, are literals that are either included in a given context or that can be inferred from the given context and rules; (ii) a square Boolean array field, `edges`, which

[61] It also contains another Hash Map field, `tiedVariables`, which maps unassigned variables to other unassigned variables. However, this field is not utilised in our work.

corresponds to the (directed) edges of the inference graph - with `true` at position `(i,j)` implying that there exists an edge starting from node `i` and ending to node `j`, as they indexed in the `nodes` list; (iii) a Hash Map field, `originRules`, which maps each graph node - i.e. each literal - to a non-empty list of rules which led that literal to be included in the graph as a node during some stage of its construction loop – see below. The class also accommodates methods for properly adding and removing nodes from and to the graph as well as returning the crown rules of arguments in the graph.

3. A `DualRepresentation` class which describes the dual representation of a grounded model. It contains as fields: (i) a `KnowledgeBase` field, `kb`, which corresponds to the given knowledge base; (ii) a `Context` field, `context`, which corresponds to some given context; (iii) an `InferenceGraph` field, `graph`, which corresponds to the inference graph constructed by `kb` and `context` – see below for more details; (iv) a list of literals field, `markedLiterals`, which corresponds to the list of marked literals in the dual representation – see (Michael, 2019: 84) as well as below for more; (v) a list of rules field, `markedRules`, which corresponds to the list of marked rules constructed during the construction of the dual representation – for more, see below, as well. Also, the class accommodates methods for finding arguments in the grounded model in favour of a certain literal, in case such an argument exists.

As far as the dual representation of a grounded model is concerned, the construction algorithm described in (Michael, 2019: 84) is implemented as follows, given access to a context x and a prioritised knowledge base $k = (\rho, \prec)$:

1. At first, using the context's literals as well as the rules contained in the knowledge base, the inference graph, `graph`, describing all the possible inferences in x under k is constructed in the following way:
    a. Given all the literals in the given context and an initially empty list of literals, `inferred`, for each rule in the given knowledge base:
        i. A list of substitutions, `Subs`, of all the substitutions that unify all literals in the rule's body given the context's literals is constructed.
        ii. For each substitution in `Subs` apply that substitution to the rule's head and, if it has not been included in inferred, then add it.

   b. Terminate when no new literals is added in `inferred` for an entire loop over all rules.

2. Next, a list of literals, `markedLiterals`, is constructed, initially populated by the context's literals, as well as an initially empty list of rules, `markedRules`.
3. Any literal in `graph` that conflicts with some literal in `markedLiterals` is removed from `graph`.
4. All rules in k whose body literals are all included in `markedLiterals` and which, at the same time, are preferred over any other conflicting rule whose body literals are also all in `markedLiterals` are added to the `markedRules` list.
5. Next, the head of each rule in `markedRules` is added to `markedLiterals`.
6. Steps 3 to 5 are repeated until convergence – i.e. until no more literals are added in `markedLiterals`.

4.2.2 Remarks about the above implementation

As we have mentioned in section 4.1, a subtle point which should be further clarified is how numerical expressions are evaluated in the setting presented above. To begin with, we will refer to any "numerical" instance of `?=(·,·)` as well as any instance of `?<(·,·)` as *numerical literals* while we will refer to any other literals as *typical literals* or simply as *literals* if this introduces no ambiguity. So, `?=(·,·)` may be both a numerical as well as a typical literal, depending on whether at least one of its arguments is a numerical constant or expression using mathematical functions and/or operators[62].

The crucial point here is to describe how unification is conducted so as to allow for numerical literals to be externally evaluated using built-in java functions and operators. So, let some rule, `rule`, and also let `body` denote its body – i.e. `body` is a list of literals that serve as the rule's antecedents. Also, let `context` denote a context. Then in order to find all substitutions of `body` according to `context`:

1. All typical literals apart from instances of `?=(·,·)` are unified with respect to `context`. Let `subs` denote the set of substitutions that occurs from this step.

[62] One may say that `?=(·,·)` is *overloaded* however this is not exactly true, given that the actual deviation from the formal definition of our language is that we introduce certain function symbols – i.e. mathematical functions. Apart from that, `?=(·,·)` "behaves" as it should – i.e. as our language's equality symbol.

2. For each substitution, `sub`, in `subs` and for each typical instance `inst` of `?=(·,·)` in `body` apply `sub` to `inst` and:
    a. If both arguments of `inst` are constants but they are not unifiable then remove `sub` from `subs`.
    b. If both arguments are unassigned variables then remove `sub` from `subs`.
    c. If one of them, let `X`, is an unassigned variable and the other is some constant, let `c`, then extend `sub` by adding `{X -> c}` to it.
3. For each substitution `sub` in `subs` and any *numerical instance* `inst` of `?=(·,·)` in `body` apply `sub` to `inst`, evaluate externally any mathematical expressions and unify as in step 2.
4. For each substitution `sub` in `subs` and any numerical literal `num` that has remained unchecked apply `sub` to `num` and evaluate externally all the occurring mathematical expressions. Since `num` is necessarily an instance of `?<(·,·)`, act as indicated below:
    a. If at least one of its arguments is an unassigned variable, then remove `sub` from `subs`.
    b. Otherwise, let `a` and `b` be the values of the literal's left and right argument respectively. If $a < b$ then proceed to the next literal or terminate in case there is no literal to proceed while if not then remove `sub` from `subs`.

So, for instance, should we have a rule with body `[p(A), ?<(A-4,3)]` and a context containing only the literal `p(5)`, then, according to the above algorithm, we have:

1. At first, `A` is unified with `5`, giving the only possible substitution: `{A -> 5}`.
2. Then, the aforementioned substitution is applied to the left argument of `?<(A-4,3)` resulting to the mathematical expression $5 - 3 < 3$ which is true, hence `{A -> 5}` is accepted.

4.3 Chess related features

In this section we will discuss the domain specific features of the designed application, as well as the rationale behind them in cases where it is considered necessary to do so.

To begin with, any chess related function is eventually based on python's chess module which, in short, gives access to:

- An object-oriented implementation of the game of chess accommodating all typical chess functionalities as well as all current FIDE regulations – updated with each new version of the module.
- A Scalable Vector Graphics (SVG) representation of a chessboard including last move highlighting, check highlighting as well as several other attributes that facilitate the game's graphical representation.
- Built-in functions providing access to several game-related information. For instance, using the module's built functions one may have access to squares attacked by a certain player (in terms of colour) or by a specific piece on the chessboard in a given position. Moreover, access to higher level information such as pins in a given position on the board is also provided.

The latter are utilised during context construction as we shall present next. In order to construct a context describing a certain position on the chessboard, we need first to define which built-in predicates other than the ones included in the already implemented generic language of Machine Coaching[63] we consider necessary. Our choices will, at a significant extent, determine the expressiveness of our language since the available predicates determine the relations which we can utilise in our rules. In general we could split built-in predicates in the following classes:

- Predicates regarding static features of each position such as possible moves and pieces of data about them – i.e., which side makes the move, what piece is moved, from which and towards which square it that piece moved, whether that move is a capture one and so on – or the absolute location of each piece on the board and so on.
- Predicates regarding features related to game history as well as each position, such as castling rights for each player and side – i.e. kingside/queenside castling – en passant captures and so on.
- Predicates regarding static features of the game and which are independent of the current position, such as whether the bot plays as black or white and so on.

A comprehensive list of the built-in predicates as of the time this thesis is being written may be found in Appendix A.

[63] As a reminder, these literals are `?=(X,Y)`, and `?<(X,Y)`.

Using built-in predicates, each time that it is the bot's turn to play, a context describing the current position on the board as well as other necessary facts about the game is constructed and provided as an input to the reasoning algorithm – which is invoked by the application as a separate java sub-process. Before we proceed, it is important to make a remark about the context describing each position. As one may have already realised, in each situation both players' legal moves are provided within the context. As it has also been explained before, it is useful to have information about what the opponent is capable of doing in a given position, should one forfeit their own turn[64].

However, since this leads to not all moves in a given context to be legal for the side to play, it is possible that a user may provide rules that allow for illegal moves to be considered adequate by the bot. To avoid such an absurd behaviour, a posterior check is performed once a move has been suggested by the bot, so as to ensure that it is legal and, in case it isn't, an alternative move is chosen randomly from the set of available legal moves[65].

4.4 Graphical User Interface

Chess coach's GUI has been designed using python's PyQt5 module – actually, a python binding of the Qt toolkit. While java offers plenty of native GUI development options, such as Java Swing, PyQt5 was preferred due to python's chess module which provides access to a plethora of utilities related to chess – see previous section for more details. Thus, using PyQt5 alongside with python's chess module, while invoking PRUDENS's

[64] This may happen in several occasions. At first, some certain moves of the opponent, like a checkmate move, are necessary so to avoid defeat in the next move. Secondly, it has been reported by several professional players – for more, see chapter 5, section 5.2 – as well as by most of the contemporary computer chess works that forfeiting one's move and exploring the opponent's moves on that position provides wider insight on the opponent's plan – see also the discussion about the null move heuristic in (Greenblatt, 1969: 808).

[65] There is also another path to ensure that always legal moves are suggested by the bot which was not preferred due to leading to a modification of the bot's knowledge base from entities other than the coach. Namely, a rule off the form `Rule :: -plays_as(Colour), move_played_by(Move,Colour) implies -suggest(Move);` could be introduced with higher priority than any other conflicting rule in the knowledge base, so as to provide an argument against the inclusion of illegal moves attacking any other argument for them.

reasoning engine as a service facilitated a more concrete and efficient implementation not only of the chess coach's GUI but also of the application as a whole.

In the rest of this section we present, in short, the designed GUI as well as the functionalities it accommodates as far as human-machine interaction and Machine Coaching in general are concerned.

4.4.1 Starting screen: choosing a mode of coaching

The application's first screen is shown in Figure 11. As one may observe there, the screen consists of the following parts – starting from top left and proceeding right and down towards the opposite bottom right corner:

- An "Import" button. This button, once pressed, redirects the user to a file opening dialog, where they are prompted to load a previous game. As we shall see later on, this facilitates an asynchronous model of coaching, in the sense that the coach provides advice based on a game that the bot has already played - e.g. against another bot or human player.
- An "Approve" button. This button, once pressed, declares that a move played by the bot as well as the corresponding explanation for that move have been approved by the coach – i.e no counter-argument needs to be provided by the user.
- A "Delete rule" button. This button, once pressed, redirects the user to a rule deletion dialog from which they can delete any rule they wish from the bot's knowledge base - for more details consult subsection 4.4.2.
- An "Add rule" button. This button, once pressed, redirects the user to a rule addition dialog from which they can add any rule they wish to the bot's knowledge base - for more details, consult subsection 4.4.2.
- A "Play as" button box which contains two buttons: "White" and "Black". Once one of these buttons is pressed, the button box is replaced by a chessboard and the user is set to play accordingly to their choice - i.e. as white, on condition they have pressed "White" and as black otherwise.
- A "Game" text field, where the game's moves appear. The moves are displayed in typical UCI format - so as to match the format in which they appear in the "Explanation" section, see next for details.

- A "Knowledge Base" text field, where the prioritised knowledge base used by the bot in the current game - whether it is a loaded previous game or a game the user is playing against/with the bot in live mode.
- An "Explanation" text field, where the explanation regarding a move made by the bot is presented. The explanation for a move is, by its definition, the argument that has internally led to its suggestion – for more details, consult 4.4.2.

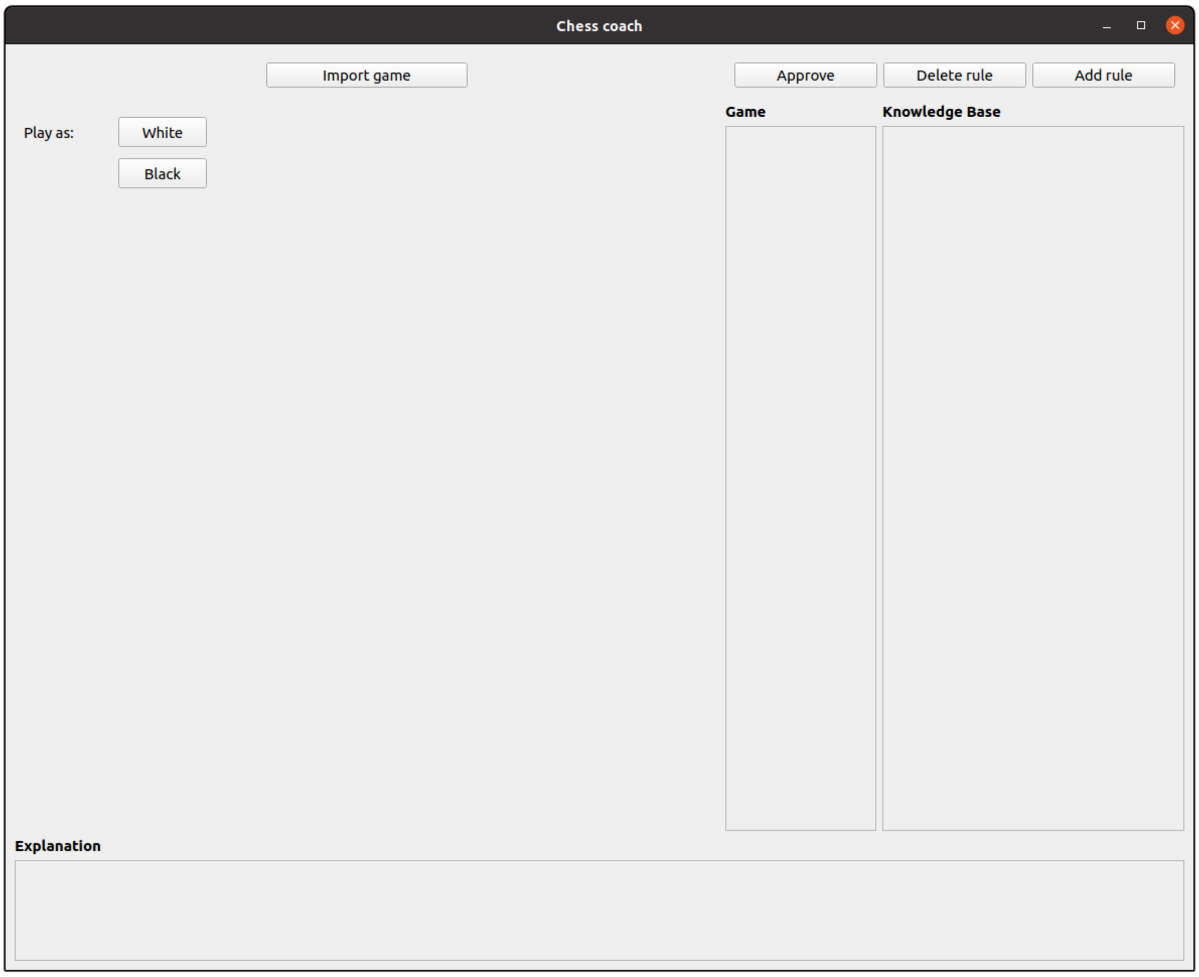


Figure 11: The application's starting screen.

We now proceed by describing the two available modes of coaching.

4.4.2 Live coaching

Should the user choose to play a game against/with the bot, they are redirected to the screen[66] shown in Figure 12. The only actual difference between this and the starting

[66] When playing in live mode, there are two reasonable scenarios: (i) the coach is playing against the bot; (ii) the coach is playing alongside the bot by inspecting its moves during a game against

screen is that now a chess board appears in place of the “Play as” button box - we have assumed that the user has chosen to play as white for the purposes of our demonstration.

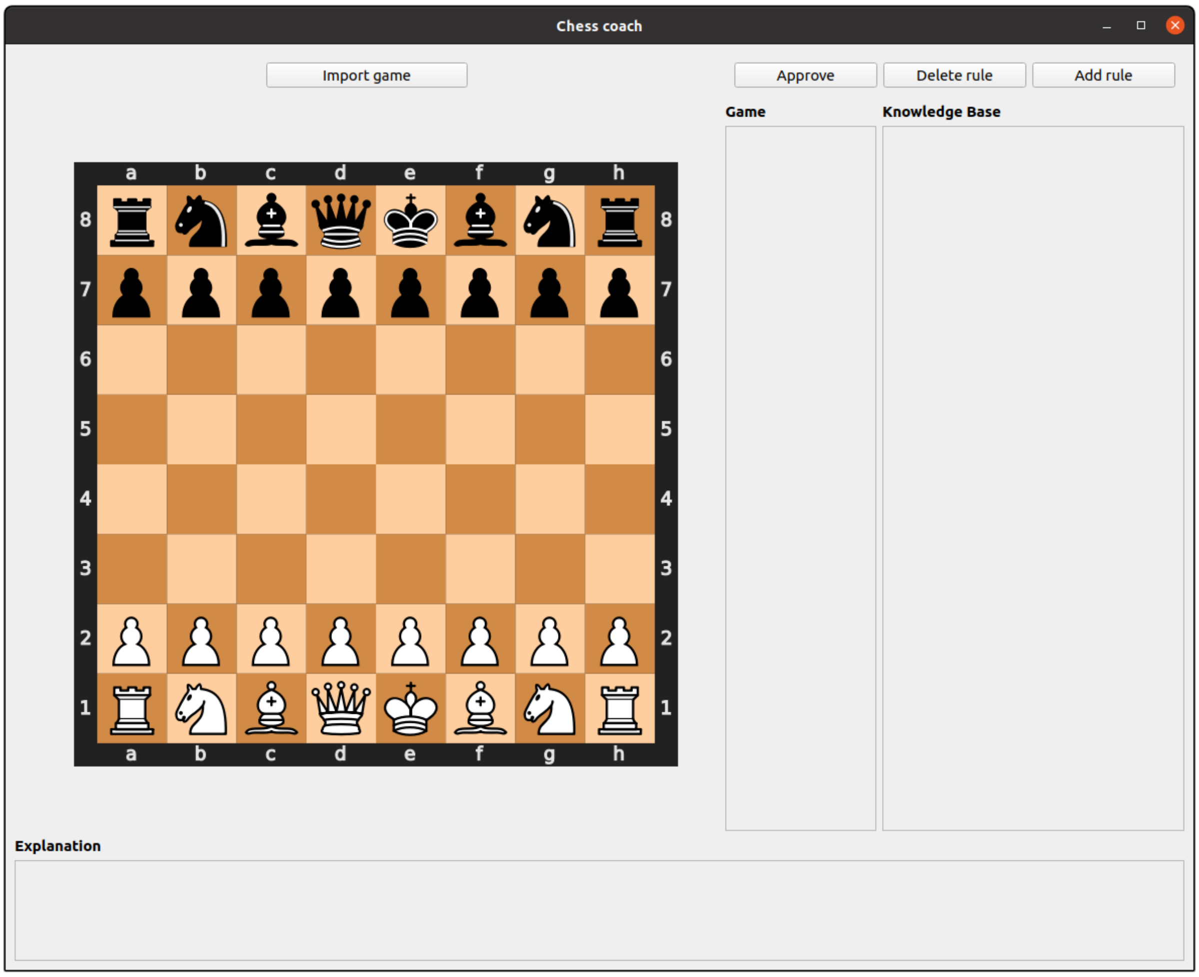


Figure 12: Playing against/alognside our bot.

As far as moves from the side of the user are concerned, they are executed by clicking first on the desired move’s starting square and right after on its destination square. Provided that the move is legal, it is executed, otherwise no change takes place and the application waits for the user’s next click. Regarding the bot’s moves, they are executed automatically, under the following protocol:

1. A list of all legal moves alongside any other meaningful information about the position are properly encoded in the first order language presented in chapter 3,

another player. For more details about the plausibility of these scenarios related to actual chess coaching, consult chapter 5, section 5.2.

section 3.2. This list constitutes the context describing the current position on the board.

2. Next, the corresponding dual representation of the grounded extension of the emerging argumentation framework is being constructed – for more details, see chapter 4, section 4.2.
3. Given the moves that have been suggested in step 2, the bot chooses randomly - i.e. with equal probability - one of them and plays it. In case no move was suggested, the bot, by default, chooses randomly a move from the set of all legal moves in the current board position.

After the bot makes a move, the "Explanation" text field is updated with the corresponding explanation that has led the bot to that specific suggestion or nothing, in case no move was eventually suggested and the played move was merely randomly selected.

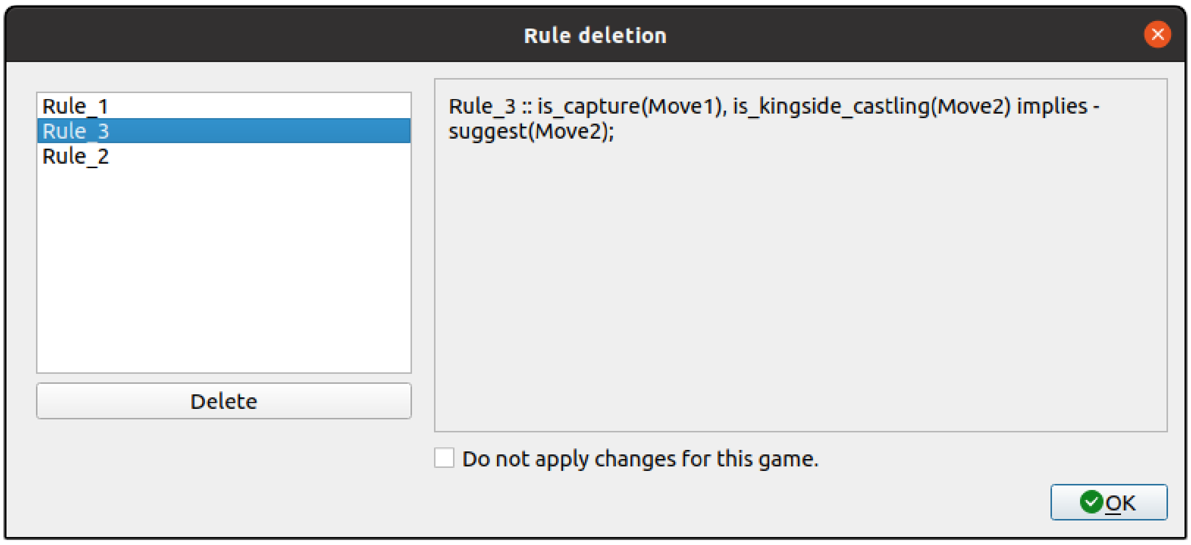
After a move has been played by the bot and the user has checked the provided explanation, there are three options provided to the user, represented by the three buttons provided on the top right corner of the application's window. Namely, the user is allowed to:

1. Approve a move as well as the corresponding explanation for it.
2. Delete a rule or a set of rules from the bot's knowledge base.
3. Add a rule or a set of rules to the bot's knowledge base alongside their priority level with respect to other conflicting rules.

In order to approve a played move as well as the corresponding explanation, the user can press the "Approve" button right over the "Game" and "Knowledge Base" fields or simply play their next move, without providing any counter-argumentation.

In order to delete some rule(s) form the bot's knowledge base, upon seeing the bot's move as well as the corresponding explanation, the user can open the Rule deletion dialog by pressing the "Delete rule" button – see Figure 13. From there, they are allowed to delete any rules they wish as well as choose whether the changes should apply from the current game or once this game is over.

As far as adding a rule – or more – to the bot's knowledge base, this can be done by pressing the "Add rule" button – see Figure 12 – which redirects the user to the corresponding dialog – see Figure 14. In this dialog, there are several options as far as rule construction is concerned. To begin with, the user is allowed to edit the Rule's name, body and head fields on the left part of the window by hand. Alternatively, there is also the possibility of adding literals to the rule's body and/or head by clicking on the desired built-in predicate in the "Built-in predicates" list and then pressing the "Add to body" or "Add to head" button accordingly[67]. Also, observe that at the rightmost part of the window, there is a text box in which a description about the currently clicked built-in predicate is presented. This description includes the intended interpretation of that predicate in our context, a small example of some positions on the board described by it as well as a list of all meaningful constants for each of its variables – for a full list of all the built-in predicates as of the time this thesis was written, consult Appendix A.



Rule_3 :: is_capture(Move1), is_kingside_castling(Move2) implies -suggest(Move2);

Figure 13: Rule deletion dialog.

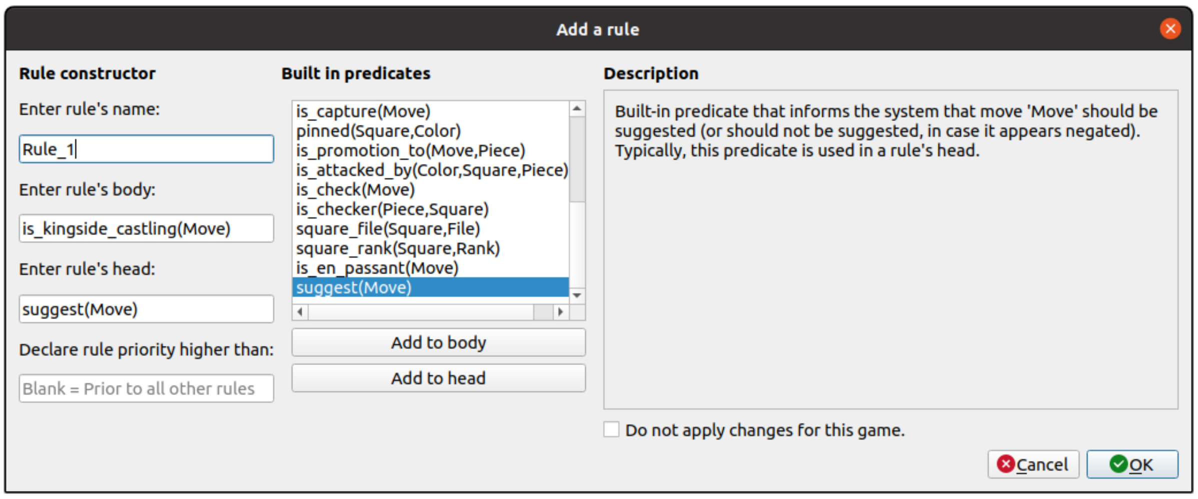
Observe that the user is also allowed to determine the new rule's priority, by providing the rule right above which they wish their new rule to be included – on condition this field is left blank, the new rule is assumed to be preferred over any other conflicting rule already included in the bot's knowledge base. While, at most cases, any new rules inserted in the bot's knowledge base are expected to be considered of higher priority than the rest conflicting rules – since they are expected to constitute refinements/exceptions to already existing ones – there are certain occasions in which

[67] The only built-in predicate that makes sense to be on a rule's head is suggest(Move) which is by default used to indicate that a move should – or should not, in case it appears negated – be suggested as an adequate move.

we would prefer to keep a rule always on top of all the others. For instance, consider the following rule:

```
Mate_1 :: plays_as(Colour), move_played_by(Move,Colour),
is_checkmate(Move) implies suggest(Move);
```

This rule, being of ground importance, is expected to be provided to the bot at some early stage of the learning process. In case `Mate_1` was overridden by another rule, say for instance `Noq`, then in positions like the one presented in Figure 8, the bot would lose a mate in one move, which is, evidently, an undesired behaviour.


Add a rule

Rule constructor

Enter rule's name: Rule_1

Enter rule's body: is_kingside_castling(Move)

Enter rule's head: suggest(Move)

Declare rule priority higher than: Blank = Prior to all other rules

Built in predicates

is_capture(Move)
pinned(Square,Color)
is_promotion_to(Move,Piece)
is_attacked_by(Color,Square,Piece)
is_check(Move)
is_checker(Piece,Square)
square_file(Square,File)
square_rank(Square,Rank)
is_en_passant(Move)
suggest(Move)

Description

Built-in predicate that informs the system that move 'Move' should be suggested (or should not be suggested, in case it appears negated). Typically, this predicate is used in a rule's head.

Figure 14: Rule addition dialog

Even if no conflicting rule of higher priority than `Mate_1` exists in the bot's knowledge base, it is expected that a checkmate move will not be the only one suggested in most positions – e.g. the position shown in Figure 8 – and, given that, the chance that it is finally played by the bot is narrowed down. To avoid such cases, one may also introduce the following rule, so as to override any other suggestion on condition that a checkmate move exists:

```
Mate_2 :: plays_as(Colour), move_played_by(Move1,Colour),
is_checkmate(Move1), move_played_by(Move2,Colour),
-?=(Move1,Move2) implies -suggest(Move2);
```

Examples as the above one indicate that it is of crucial importance to allow for rules to be maintained on top of others constantly. Under the semantics of Machine Coaching, as declared in (Michael, 2019: 85) and described in detail in chapter 3, section 3.3, we can achieve this by allowing the user to decide at each time whether a rule should be preferred over any other conflicting ones or be set at a certain place among other conflicting rules[68].

Given the above three actions, the user may provide feedback in all five (5) ways declared in the Machine Coaching interaction protocol that is presented in (Michael, 2019: 85). Namely:

1. *Unrecognised* rules are deleted, as described in (Michael, 2019: 85), in the proof sketch of Theorem 4.3.
2. *Superfluous* rules are deleted as well, again as described in (Michael, 2019: 85).
3. *Incomplete* rules, i.e. rules in the user's grounded model that do exist in the returned argument/explanation and whose inclusion in the machine's knowledge base would have led to an argument conformant to the user's theory (Michael, 2019: 85), are added to the machine's knowledge base.
4. *Indefensible* arguments, i.e. arguments in the user's grounded model that attack an argument of the machine's theory but are not attacked by any other argument in the user's model (Michael, 2019: 85), are provided rule by rule setting priorities accordingly.
5. *Approved* explanations are treated by actually performing no operation on the machine's knowledge base.

Given the above implementation of the interaction protocol described in (Michael, 2019: 85) and the additional assumption that rules in a user's knowledge base $k = (\rho, \prec)$ in our domain of application – i.e. chess – can be linearly ordered with respect to $\prec$, then Theorem 4.3 (Michael, 2019: 85) ensures that learning is conducted efficiently in the sense defined in chapter 3, section 3.3.

[68] There are also options other than this that facilitate the aforementioned desired behaviour. For instance, on could also allow for strict rules (Prakken, 2010: 97) which cannot be disputed by any other contextual counter-argument. Moreover, one could also restrict randomness in the sense of specifying a (possibly non-uniform) probability distribution over suggested moves – this could also be manipulated by the user themselves e.g. by introduce a built-in predicate `suggest(Move,Weight)` where `Move` is some legal move and `Weight` is a number indicating how highly should `Move` be ranked during random selection. All these are addressed in more detail in chapter 6.

Summarising, during live coaching, the user is allowed to play a game against the bot – or, equivalently, spectate a game played by the bot against another (human or machine) player – and provide feedback to it either in the form of counter-argumentation or by approving its decisions, leading, under certain assumptions, to the bot converging to the user's theory under the semantics defined in chapter 3, section 3.3.

4.4.3 Study mode

Apart from live coaching, the designed application also allows for an asynchronous coaching option, in the following sense:

- The bot plays a game at some time against another (human or machine) player and records the game itself alongside with any arguments for the moves it has played.
- At a later time, a (human) coach asks the machine to load the game so as to study it together.
- At each of the machine's move, the coach, viewing the argument that led to that move, provides feedback in one of the five ways declared in chapter 3, section 3.3.

The above is implemented by the "Import" button in our design – consult Figure 11. By pressing it, the user is redirected to an "Open file" dialog where they can choose the specific game file to load[69]. Then, a hidden button, namely "Next move", appears next to the "Import" button, above the chessboard – top left corner of the application's window. Using this button, the user can proceed to the next move of the loaded game.

As with live coaching, the same interaction options are provided to the (human) coach. More precisely, the coach may:

1. Approve a move played in the loaded game as well as the corresponding explanation by pressing the "Approve" button – see Figure 11.
2. Delete a set of rules included in the bot's knowledge base by pressing the corresponding "Delete rule" button and making their choices from within the "Rule deletion" dialog.

[69] Games are stored in JSON files loaded as python dictionaries with game moves as keys and the corresponding explanations, should they exist, as values, or `None` otherwise.

3. Add rules or entire arguments in the bot's knowledge base by pressing the corresponding "Add rule" button and making their choices from within the "Rule addition" dialog accordingly.

Note that in either case – i.e. live of study mode – the user is allowed to choose one way of interaction at a time, conforming to the interaction protocol presented in (Michael, 2019: 85). That is, should a user choose e.g. to delete some rules from the bot's knowledge base then they are not allowed to add any new rules to it no sooner than the next move has been played by the bot – the same applies to adding rules[70]. In a sense, this corresponds to the view that a user may consider machine's explanations erroneous based on one criterion at a time.

4.4.4 Game over dialog

Once a game has come to its end, the user is prompted to make some decisions according to their next actions. More precisely:

1. They are asked whether they wish to save the game played or not – this applies only to games played in live mode since games loaded in study mode are already saved.
2. The user is asked whether they wish to play another game against the bot or not – this applies to both live as well as study modes – and, in case they wish to, they are prompted to pick a colour to play with.

Also, once a game is over and the user has specified their preferences with respect to the above, any (temporary) files related to the previous game are deleted – consequently, an unsaved game cannot be restored.

[70] In any case, this is implemented by temporarily deactivating the three interaction buttons on the top right corner of the application's window once one of them has been pressed.

Chapter 5
Evaluation

In this chapter we will present several results regarding the evaluation of our work. We decided to assess our work in two orthogonal directions. On the one hand, we measured the efficiency of our implementation in terms of computation time with respect to several parametres. Doing so, we intended – apart from assessing the implementation itself – to empirically verify the theoretical results that are reported in (Michael, 2019: 84) regarding the efficiency of the dual representation of a grounded model. On the other hand, given that our intention is to capture human knowledge and heuristics regarding chess and transfer them to a machine, we also interviewed domain experts in order to receive meaningful domain specific feedback.

The structure of this chapter is as follows: (i) in section 5.1 we describe the methodologies used so as to assess the efficiency of our implementation, while we also present and discuss the results we obtained; (ii) in section 5.2 we present and discuss the feedback we received from professional players and chess coaches regarding our work.

5.1 Evaluation of the implementation

In this section we shall present several results regarding the assessment of the efficiency of our implementation as a whole as presented in chapter 4, sections 4.1 and 4.2. To do so, we will rely on synthetic data, given that no satisfactory sets of real data were found that could allow us to systematically modify and control all the major parameters we would like to.

5.1.1 Generation of synthetic knowledge bases and contexts

In this subsection we will describe the way in which our synthetic data – namely, pairs of knowledge bases and contexts – were generated. Before we describe our random knowledge base/context generation protocol in full detail we shall first mention that the parameters against which we decided to measure the build time of a grounded model's dual representation are:

- *Negation ratio*, i.e. the ratio of negative literals over the total literals count included in a knowledge base – namely, in the rules' bodies – as well as in a context. For the rest of this section, we will refer to the former as *negation ratio* unless it is differently indicated.
- *Predicate arity*, i.e. the number of arguments a predicate includes, as defined in chapter 3, section 3.2.
- *(Rule) Body size*, i.e. the number of literals a rule contains in its body. We will also refer to this parameter simply as *rule size*.
- *Knowledge base size*, i.e. the number of rules a knowledge base contains.

Proceeding now to the description of our random knowledge base/context generation protocol to begin with, in order to randomly generate a knowledge base, we need to decide upon a protocol by which we will randomly generate rules – we set aside the knowledge base's priority relation between these rules for now. Furthermore, in order to generate a rule, we have to decide how literals are randomly generated and, consequently, predicates, variables and constants.

Starting with the lowest levels of our language, the number of variables was not considered important since variables serve mostly as placeholders. So, in all the following experiments we take care to include enough variables from a combinatorial perspective so as to allow for all other parameters to be properly set. Similarly, since it was not among our intentions to explicitly measure the effect of context complexity - e.g. in terms of its size - we simply allowed for each variable to take values from an constants array of size five (5).

Proceeding to predicates, given a fixed arity n as well as a list of variables $vars$ with length at least n, we randomly select a random sample of size n from $vars$ without repetition and assign it as the predicate's list of arguments. We also randomly assign each predicate a name of the form pX, where X is some randomly generated integer, unique for each predicate[71].

Next, given a list of predicates $pred$ as well as a negation ratio $p \in [0,1]$ we generate a literal by randomly choosing the literal's sign with probability p being negative and

[71] In general, we avoided lengthy numbers in all generated instances. While this led to some difficulties in proof-checking that knowledge bases as well as contexts were properly generated, it also drastically reduced the generated files' size.

positive otherwise. Given a list of literals, $list$, a rule is generated by selecting body literals from $list$ such that no repetitions occur as well as no conflicting literals are included in the occurring sample. As far as the rule's head is concerned, it is selected from $list$ similarly.

In order to generate a random knowledge base, we work as follows, given a desired size n as well as a list of rules, $rules$:

1. For each rule, $rule$, in $rules$:
    a. We randomly select an integer $depth$ from $\{1,2,3,4,5\}$.
    b. Let $exceptions$ be an empty list as well as denote $rule$ by $rule_0$
    c. For i from to $depth$ repeat:
        i. If $body_{i-1}$ is the list of body literals of $rule_{i-1}$ then let $body_i = body_{i-1} \cup \{lit\}$ where lit is some literal that neither itself nor its opposite appear in $body$.
        ii. Let $head_i$ be the opposite of $head_{i-1}$, where $head_{i-1}$ is the head of $rule_{i-1}$.
        iii. Define rule $rule_i$ to have $body_i$ as its body as well as $head_i$ as its head and add it to[72] $exceptions$.
    d. If by adding $exceptions$ the desired size of the knowledge base kb is not exceeded, then add them to the knowledge base by ascending priority with respect to their indices as well as with higher priority than any previous rules, otherwise add as many randomly generated rules are needed so as to reach the desired knowledge base size.

At last, given a knowledge base k as well as a list $preds$ of all the predicates included in rules in k, we construct a context by randomly selecting a number of predicates from $\{1,2,\dots,10\}$ and then substitute all its variables with corresponding constants as well as deciding upon their sign given a negation ratio $p \in [0,1]$. After variable instantiation has

[72] We preferred this approach instead of simply adding rules that have not yet been included in our knowledge base so as to ensure that any knowledge base would contain some structure that could lead to conflicts and hence substantially assess our reasoning mechanism. However, since exception depth was not further manipulated in this series of experiments, no further results will be reported about it.

been completed, we add them to the context, so as to avoid duplicates as well as conflicts with already included literals.

At this point we should observe that, given a negation ratio $p \in [0,1]$, a randomly selected literal lit is negative with probability p should it be selected from the pool of all literals present within a knowledge base while the same is not necessarily true for a rule's body. That is, there is a significant probability that given any non-trivial negation ratio $p \in (0,1)$ there exist rules in a randomly generated knowledge base whose body literals do not follow the distribution of the rest knowledge base as far as negative/positive literals are concerned.

5.1.2 Build time against negation ratio

One of the first variables we would like to investigate is negation ratio – i.e the fraction of negative literals among all literals appearing in rules as well as in contexts. Intuitively, we expect that the effect of negation ratio will be insignificant as far as build time of a grounded model's dual representation is concerned. Indeed, consider two values of negation ratio, let $p_1, p_2 \in [0,1]$ such that $p_1 \neq p_2$ and let $t = t(p)$ be the dual representation's build time with respect to negation ratio, p. Let also (k_1, x_1) and (k_2, x_2) denote two (knowledge base, context) pairs with negation rations p_1 and p_2 respectively. Should $t(p_1) > t(p_2)$ then we can rename[73] enough predicates in k_1 so as to change its negation ratio to p_2 or approximately p_2 as well as analogously modify the corresponding context x_1. In the same way, we can also change the negation ratio of the (k_2, x_2) pair from p_2 to p_1.

Given that build time is not expected to be sensitive to predicates' renaming[74], we should expect that pre- and post-renaming built times should be the same – or, at least, similar –, which would lead to $t(p_2) > t(p_1)$ which contradicts our hypothesis. So, we expect $t(p_1) = t(p_2)$ or at least $t(p_1) \approx t(p_2)$ for any negation ratios $p_1, p_2 \in [0,1]$.

[73] By that we mean changing the name of a predicate from –p to p and accordingly for its opposite predicate in any occasion they appear in a given knowledge base as well as context.

[74] This is expected since in any occasion where two predicates are compared – e.g. during unification – whether a predicate appears negated or not in a literal is always taken into account in a symmetric way – i.e., there is not structurally different action taken when a literal is or is not negative.

Evidently, since our data have been randomly generated, we expect that some error may be introduce in the final result, however, this is not expected to be of much significance.

Let us now proceed to presenting some related results so as to examine the validity of our thoughts. In Figure 15 one may observe average build time plotted against negation ratio. It appears that average build time and negation ratio are slightly positively correlated, however, both Pearson's correlation $\rho_p = 0.509$ as well as Spearman's correlation $\rho_s = 0.5$ are statistically insignificant (p-values 0.381 and 0.45 respectively), suggesting that build time seems to be independent of negation ratio, as we expected.

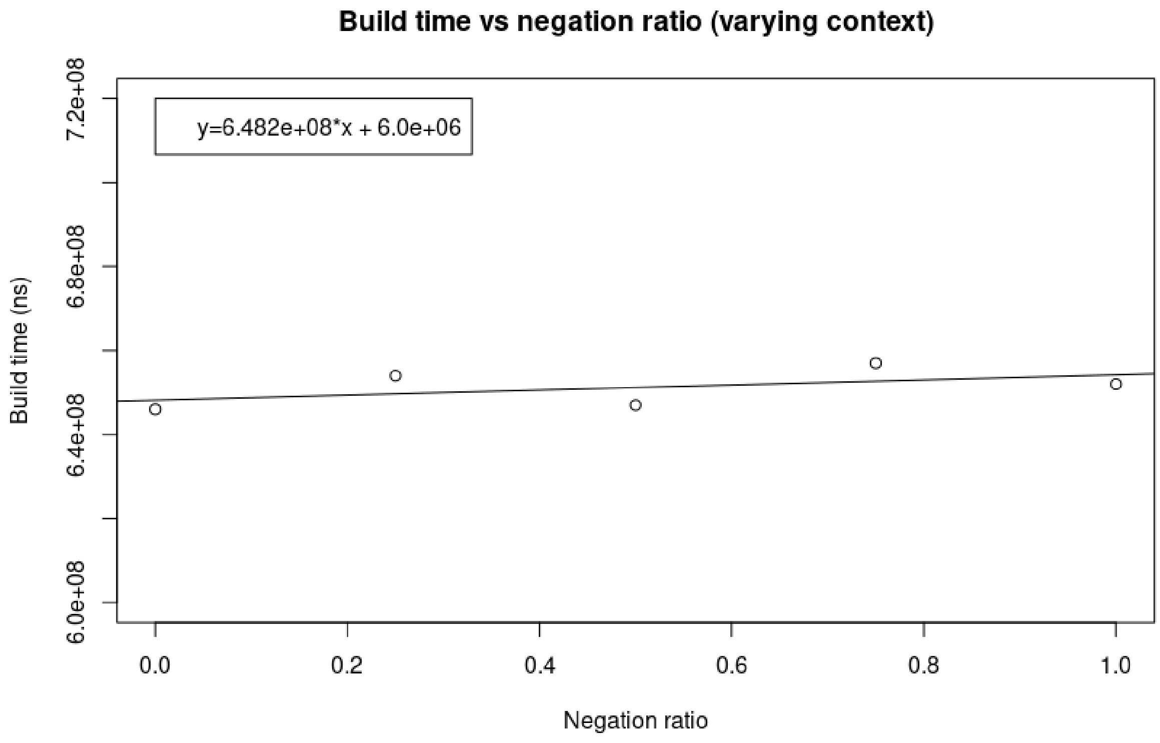


Figure 15: Build time against negation ratio. No significant correlation between the two appears to exist.

Note that in the above setting, negation ratio is manipulated in both each knowledge base as well as each corresponding context. That is, apart from manipulating the percentage of negative versus positive literals in a knowledge base's rules' bodies, we also manipulated accordingly the negation ratio within the context used alongside each knowledge base. In case we had left negation ratio constant throughout the above experiments we would expect build time to vary depending on knowledge base negation ratio since this would implicitly affect the number of rules triggered by a context and, consequently, the complexity of the emerging argumentation framework as well as its dual representation. For instance, should we demand that context negation ratio is fixed, say, to 0.2 - that is, 20% of literals are expected to be negative within each context - then we would expect that build time should be *decreasing* as negation ratio in knowledge bases increases[75]. Indeed, in Figure 16 we see how build time varies as a function of knowledge base negation ratio given that negation ratio in corresponding contexts is fixed to 0.2.

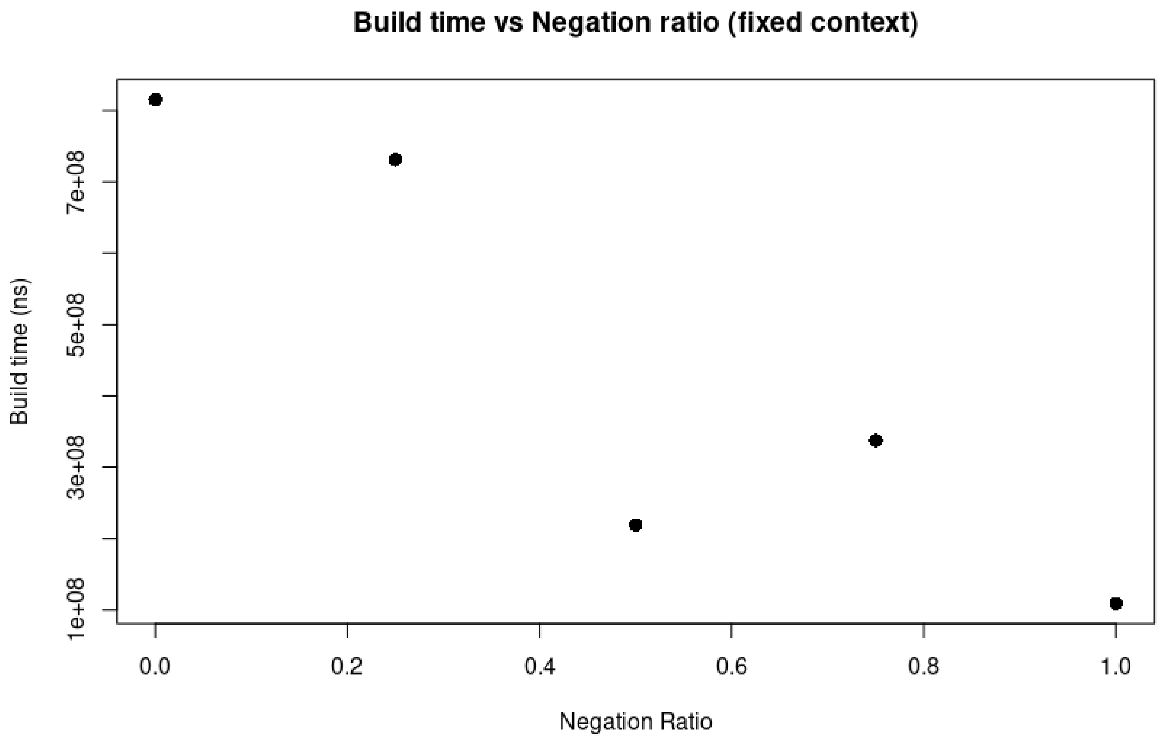


Figure 16: Build time against negation ratio given a relatively "positive" context.

Calculating Spearman's correlation[76] for the above data, we take $\rho_s = -0.9$ with the corresponding p-value being 0.083, modestly supporting our assertion that build time would be decreasing as negation ratio within knowledge bases increases.

---

[75] This is expected since as negation ratio increases, more rules are expected to contain negative literals in their bodies and, consequently, less rules are expected to be triggered from the knowledge base's corresponding context of which a little fraction consists of negative literals.

[76] Spearman's correlation, in contrast to Pearson's correlation, is robust to noise in data and is in general used to examine whether there exists a monotonic dependency between two variables – not necessarily linear, as with Spearman's correlation. The closer Spearman's correlation is to 1 the better is the dependency between two variables described by an increasing function while the closer it is to -1 the better it is described by a decreasing function.

5.1.3 Build time against predicate arity

We shall now explore the way in which build time is affected by manipulating predicate arity within the knowledge base's rules. As a first estimate, we would expect that by increasing predicate arity within rules of a knowledge base, build time would also be increased, since the overall complexity of the knowledge base would be higher than in cases with less complex predicates[77]. We now proceed in analysing data we have obtained from the above series of experiments.

As we see in Figure 17, build time, averaged over all the other parametres we are manipulating, seems to be, as expected, an increasing function of predicate arity. This also conforms to our intuition that more complex concepts require more processing time so as to make conclusions about them.

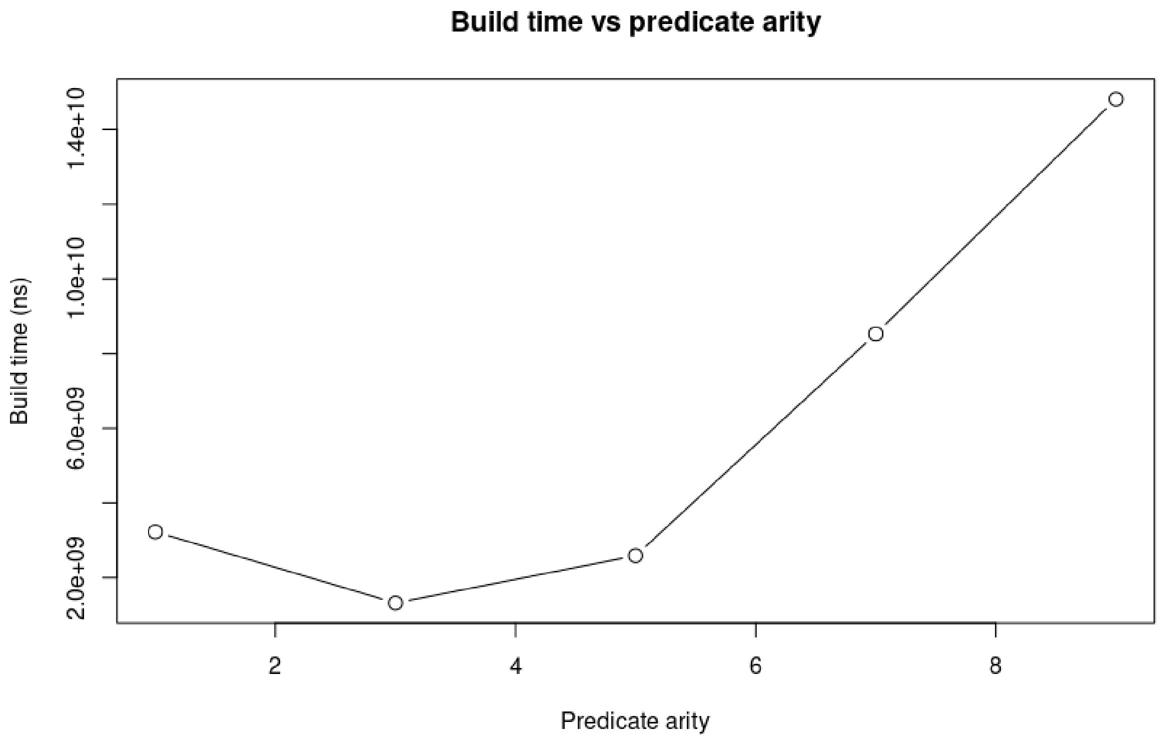


Figure 17: Build time against predicate arity

As one may observe in Figure 17, build time for predicates of arity equal to 1 is higher than build time for predicates of arity 3 or even 5. This behaviour is attributed to caching, which leads to reduced time loss in recalling cached data, explaining this

[77] More intuitively, the more arguments a predicate has, the more complex dependencies it expresses amongst our universe's entities

seemingly unexpected phenomenon. Supporting to our view will be findings about other parametres of our experiments which will also follow the very same pattern – i.e. build time will be slower in the beginning of an experimental cycle than right after due to the latter benefiting from caching.

The above results regarding build time being increasing with respect to predicate arity are also modestly supported by the value of the Spearman correlation coefficient for the above data, namely $\rho_s = 1$, with a corresponding p-value of 0.083.

5.1.4 Build time against rule body size

Apart from predicate arity, the size of rules contained in a knowledge base also capture the emerging argumentation framework's complexity. Indeed, rules with larger bodies[78] may be interpreted as expressing more complex relations on top of other simpler ones. So, with this interpretation, the longer a rule's list of antecedent the more complex its conclusion – or, more precisely, the relation its conclusion describes – is.

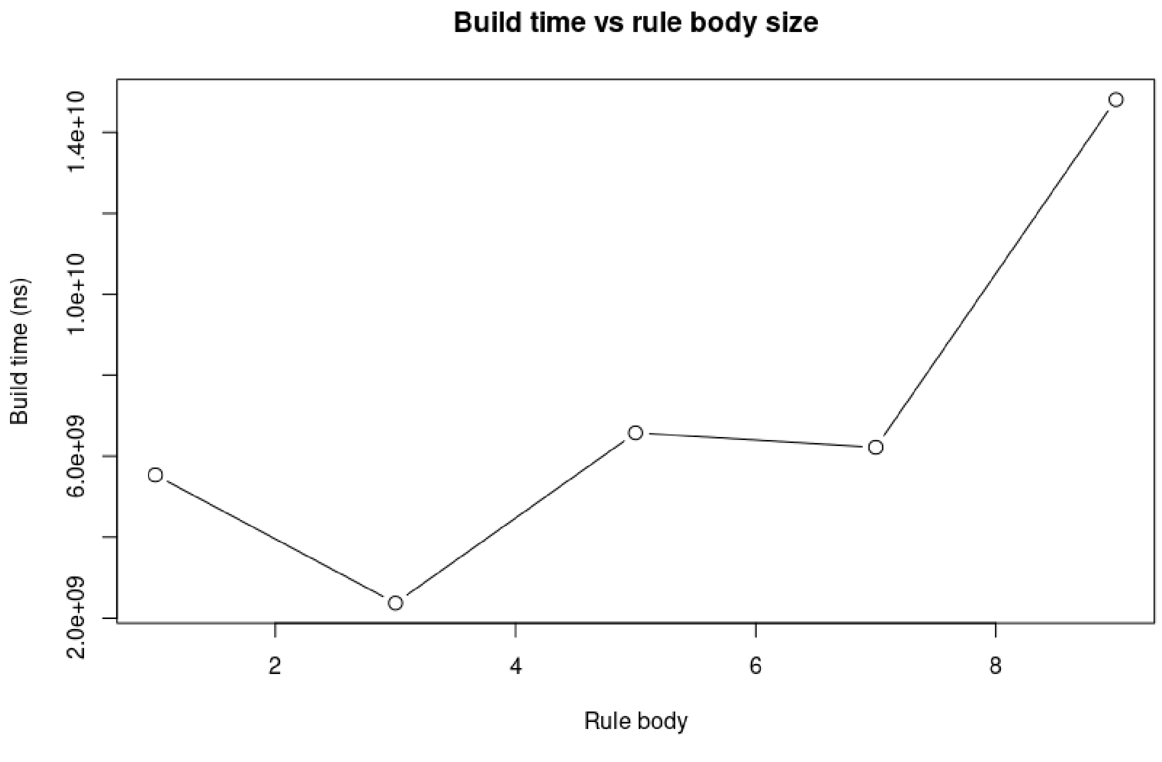


Figure 18: Build time against rule complexity.

[78] Since a rule by definition has exactly one head literals, its size is fully determined by the size of its body.

Bearing in mind the above, we see that Figure 18 supports our assertion, since rule body seems to be an increasing function of rule size. To verify it, we calculate Spearman's correlation on the above data – after, as above, discarding the first point in our dataset due to the caching effect –, which yields a value of $\rho_S = 0.8$ with the corresponding p-value being 0.133, which again, modestly supports our intuition.

5.1.5 Build time against knowledge base size

A factor that is expected to drastically determine the efficiency of our implementation is the extent to which knowledge base size affects build time. As indicated in (Michael, 2019: 84), build time increases polynomially with respect to knowledge base size and it is this assertion that we would like to empirically evaluate. To begin with, in Figure 19 we see how build time, averaged over any other parametre other than knowledge base size, grows in terms of knowledge base size. As expected, this dependency is increasing – Spearman's correlation coefficient, $\rho_S = 0.987$, p-value equal to $2.2 \cdot 10^{-16}$. It remains now to examine whether it is also polynomial with respect to knowledge base size.

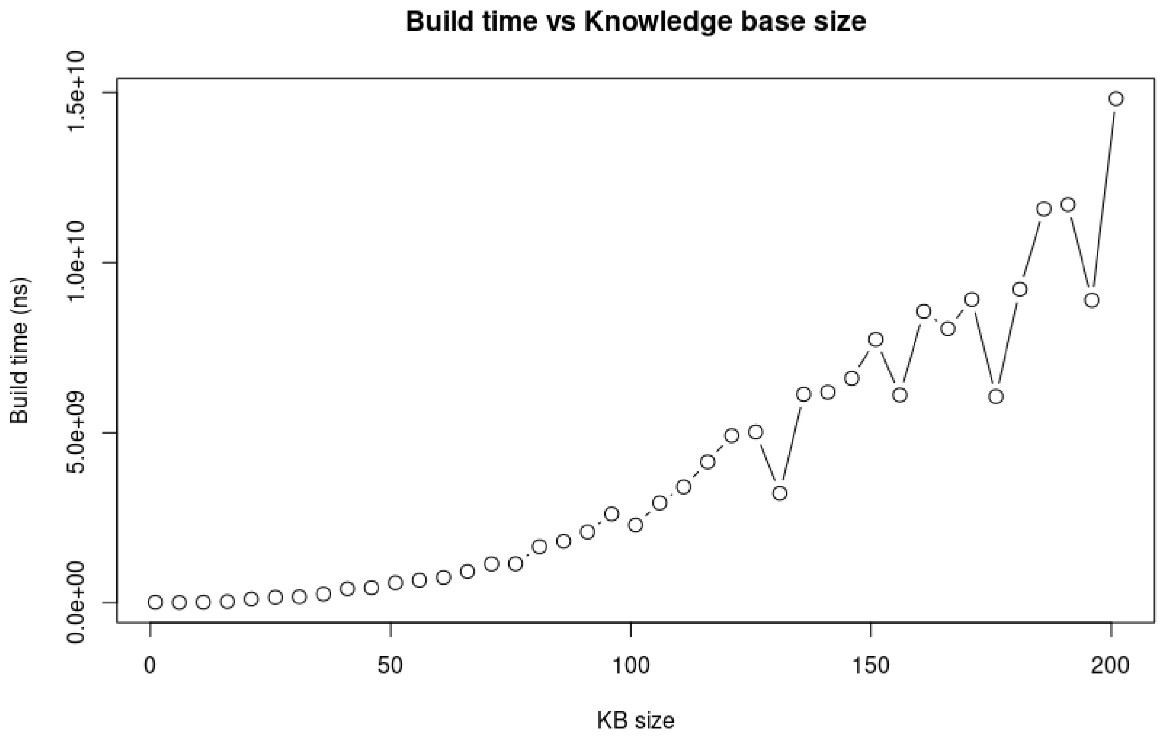


Figure 19: Build time against knowledge base size

To do so, we will at first plot our data in log-log scale, as shown in Figure 20. This is done since, if x, y are two variables with y depending on x based on a power rule, i.e. $y = Ax^k$ for some $A \in \mathbb{R}$, $k \in \mathbb{N}$, then, letting $u = \log x$ and $v = \log y$, we obtain:

$$y = Ax^k \Leftrightarrow \log y = \log Ax^k \Leftrightarrow \log y = \log A + k \log x \Leftrightarrow v = \log A + ku\,.$$

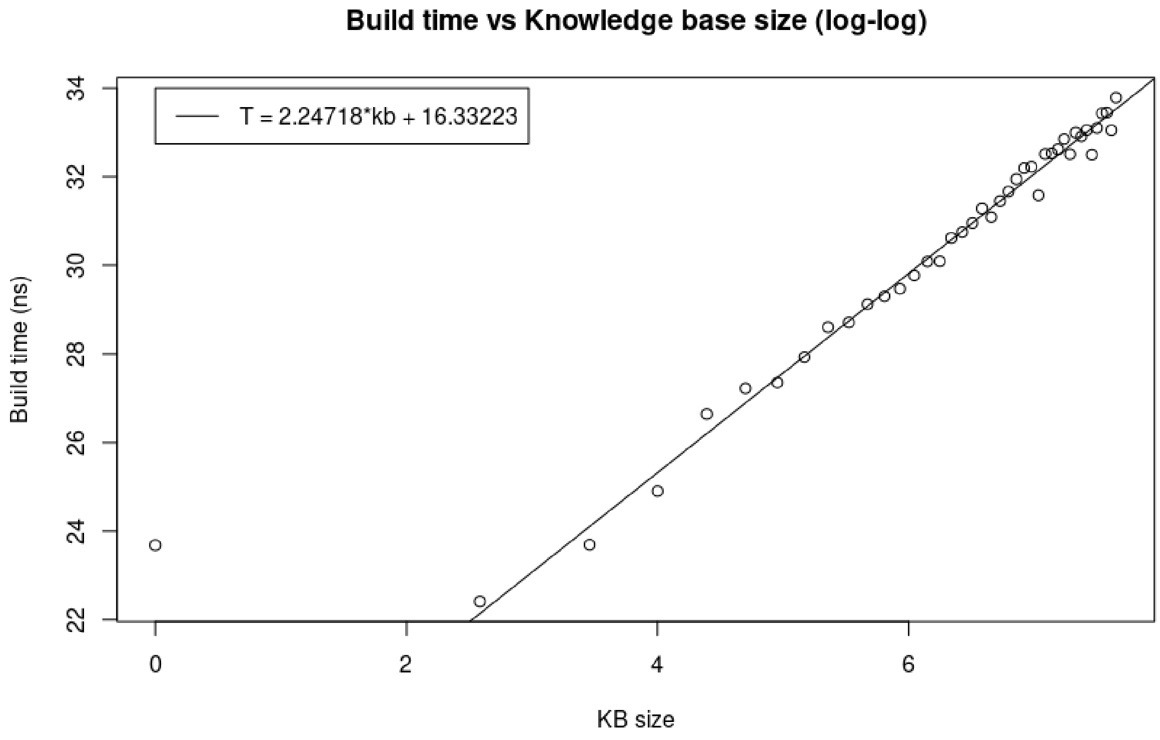


Figure 20: Build time against knowledge base size plotted on log-log scale so as to examine any polynomial asymptotic behaviour.

So, on the log-log plane each monomial and, eventually, as x approaches infinity, each polynomial, is represented by a straight line with slope equal to the degree of the monomial. Observe that the above is independent of the logarithm's base as well as that we can, given a straight line $v = b + mu$ on the $u - v$ plane, get back to the $x - y$ plane as follows:

$$v = b + mu \Leftrightarrow c^v = c^{b+mu} \Leftrightarrow c^{\log y} = c^b (c^u)^m \Leftrightarrow y = c^b x^m,$$

where c is the chosen logarithm's base.

Based on the above, we conducted a linear regression over the our dataset as represented on the log-log plane, omitting however the very first entry – i.e. the case of knowledge base size being 1. The latter was done for two reasons. To begin with, by omitting the first point in our dataset we minimise the effect of caching, which is irrelevant to the asymptotic behaviour of our algorithm we are aiming to measure. Secondly, distances between points on the log-log plane are distorted since only the upper right quartile of the Cartesian plane is mapped with u and v axis corresponding to the straight lines $x = 1$ and $y = 1$ on the Cartesian plane[79].

The resulting straight line may be seen in Figure 20 – p-value for both constants is less than $2 \cdot 10^{-6}$. So, we may safely conclude that computation time as far as the construction of the dual representation of a grounded model is concerned is indeed polynomial in terms of the size of the knowledge base. For completeness, the corresponding polynomial curve as well as the aforementioned dataset plotted on the Cartesian plane are shown in Figure 21.

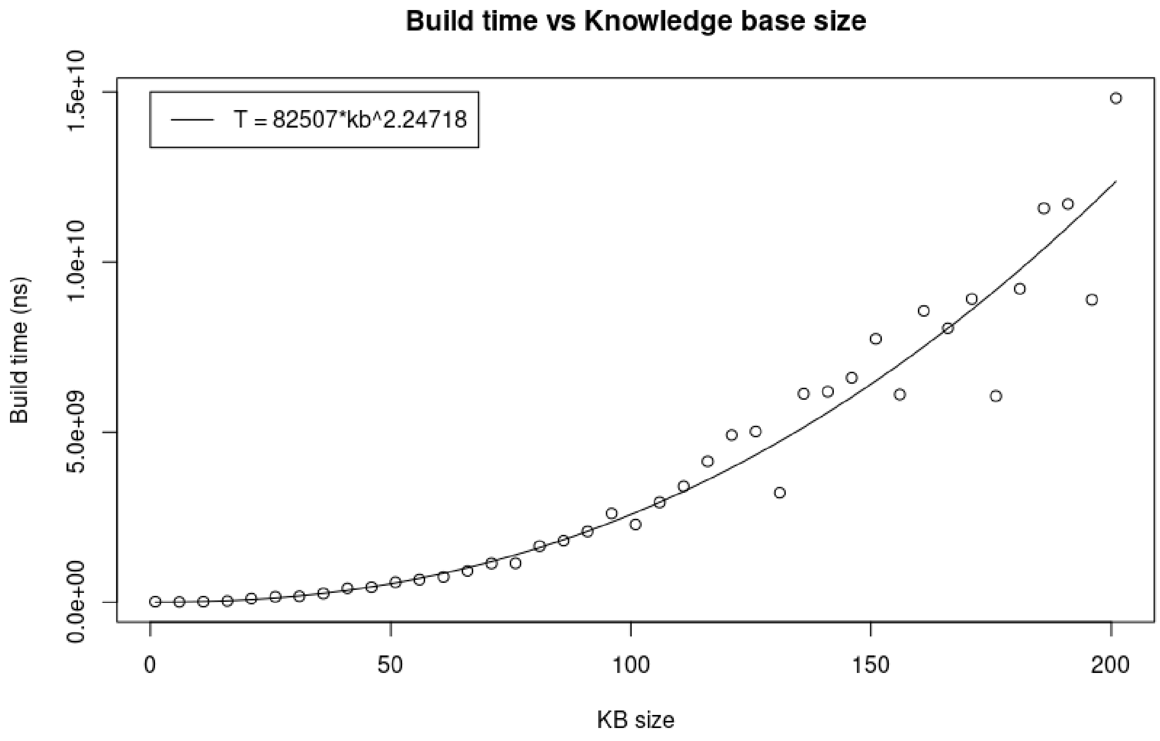


Figure 21: The polynomially growing fitting curve describing build time against knowledge base size.

5.2 Experts' feedback

In this section we will present and discuss the feedback we received form professional chess players as well as chess coaches regarding our work. Our aim was to gather useful information about how chess is viewed by them and, most importantly, how chess

[79] More precisely, as one may observe, while distances between points lying on the $(1, +\infty) \times (1, +\infty)$ part of the Cartesian plane are brought together while the rest of the Cartesian plane is magnified – e.g. $(0,1 \times (0,1)$ is mapped to $(-\infty, 0) \times (-\infty, 0)$ on the log-log plane.

coaching takes place in practice, as well as the implications of the above to our work so far. Moreover, we also sought to get domain expertise regarding chess itself as a game as well as how they would think of our approach in terms of efficiency, plausibility and so on.

5.2.1 The interviewing process

In this subsection we will present the way in which interviews with chess players/coaches were conducted for the purposes of this thesis.

At first, chess players and coaches from local chess clubs in Athens and its suburbs were contacted so as to start building up a network of people related to chess. Next, so as to diversify our sample of chess coaches and players as well as to receive more extensive feedback from the community, we contacted chess clubs from all over Greece. In total, more than 60 chess clubs were reached, however 17 chess players and coaches responded to our call for an interview while as of the time this thesis is written, 16 of them have been interviewed.

Our initial call for chess players and coaches consisted of a brief description of the purpose of our work as well as the why we needed their assistance to it. Furthermore, a fifteen (15) page demo of our application was included so as to facilitate a better understanding of our aims as well as our work.

After the first round of calls for chess players and coaches had been completed, we arranged online meetings with them which had, in general, the following structure:

1. At first, we presented to them our work live by starting a live coaching session with our bot – see chapter 4, section 4.4 – so as to elaborate on what had been presented in the already sent demo. Apart from live coaching, we also demonstrated all other capabilities of our app, including study mode for the purposes of which we utilised games we have previously played against a trained bot[80].
2. After our presentation had been completed, we asked for further questions regarding the functionality of our application and/or the interaction protocol or anything that was unclear during the presentation.

[80] The bot had been trained with game opening mostly in mind using rules based on principles found in (Lasker, 1946: 1-8).

3. Having clarified all points raised at the previous stage, we proceeded by asking: (i) for their views on the topic; (ii) whether and at what extent they considered it feasible that a bot could be trained in the presented way so as to play chess adequately in the first place; (iii) which analogies could they see between our interaction protocol and human player coaching; (iv) any other question that had occurred throughout the interview.
4. Having completed the above discussion part, we concluded by summarising what had been discussed so as to verify that there were no misconceptions as well as asked anything we intended to but was not brought up during our conversation[81].

At this point we should mention that we did not focus in the means by which interaction is conducted - i.e. the language of Machine Coaching - but merely on interaction as a process by which knowledge is transferred from a human coach to a machine trainee and backwards, in the form of explanations about the machine’s actions.

5.2.2 Feedback from the chess community

In this subsection we will discuss the feedback we received from the chess community regarding our work. During the interviewing process we have had several discussions and received feedback regarding various aspects of chess not only as a game but also about chess coaching. The latter helped us a lot in making analogies with human-to-human interaction in this context and led us to many thoughts about possible directions towards which we could conduct further research in the future – for more, see chapter 6, section 6.2.

To begin with, our overall methodology was in general considered to be plausible and applicable to the game of chess, since it resembles that of human-to-human coaching and interaction between a trainer and their trainees. Indeed, as we were told by all experts we contacted, there are, as per their words, “no absolute rules in chess”. On the contrary, rules are only contextually superior to others, given certain characteristics of the position on the board as well as the player’s experience. For instance, while it is

[81] In general, during the interviews, it was preferred from some point and after to let the expert, be it a player or a coach, to lead to conversation to any points they wished to – always regarding chess and within our purposes – so as to minimise the effect of our own views on the game and, hence, our own bias, as well as to broaden our view of the game and its strategic features.

generally not advised to move one's king during the early or middle stages of the game, the king's role is of an increasing importance as material for both sides becomes less and the game approaches its end.

Additionally, the coaching modes allowed by our application - i.e. live and asynchronous coaching - were both considered as necessary during coaching since, as described thoroughly, the coaching process may contain both sessions where the trainee plays against their coach in order to receive live feedback on their moves as well as sessions, e.g. after official games, where played games are reviewed so as to find strong and weak moves as well as any other feature that may be relevant.

We shall now continue by discussing a possible issue which was unanimously considered to be the most important by all the experts we came into contact and which we should overcome in order to build an efficiently playing chess bot. That is, the distinction made between strategy and tactics in the game of chess and how this could be effectively captured by our current approach. Actually, the aforementioned distinction proved to be more complicated than we had initially expected, so we shall present one by one all the remarks made by the experts[82].

To begin with, as per the words of one of the experts we have contacted, "the best strategic player would not be able to win should they not be capable of recognising a mate in two". To elaborate more on this, all experts agreed that a language as the one we have defined seems capable to capture most if not all the strategic that may arise in any chess position – hence, our bot could potentially be a good "strategic player". Nevertheless, it was by all of them considered dubious whether features regarding combinatorial aspects of the game, such as move counting, could be captured by if-then rules based on positional features – i.e. strategic attributes as well as purely positional ones – see chapter 3, section 3.2 as well as Appendix A for more details on what information regarding a position can be accessed as of the time this thesis is being written[83].

[82] For possible solutions on the issues which the expert's comments and thoughts are unveiling, see chapter 6, section 6.2.

[83] At this point, it would be useful to remind that the only predicates of our language which directly facilitated move counting – up to one move depth – are `controls` and `is_checkmate`.

An example of counting which would, almost surely, not be captured by a completely strategic player is shown in Figure[84] 22. There, even if it seems improbable, the white plays and has a forced mate in 290 moves! Indeed, the sequence of white's moves begins as follows:

> 1. Rd1+ Bd4 2. c4+ Kd6 3. Rxg1 Bc3 4. Rd1 Bd4 5. Ka5 Bb7 6. Ka4 Ba8 7. Ka3 Bb7 8. Ka2 Ba8 9. Kb1 Bb7 10. Kc2 Ba8 11. Kd3 Bb7 12. Re1 Ba8 13. Rf1 Bb7 14. Rd1 Ba8 15. Kc2 Bb7 16. Kb1 Ba8 17. Ka2 Bb7 18. Ka3 Ba8 19. Ka4 Bb7 20. Ka5 Ba8 21. Kb6

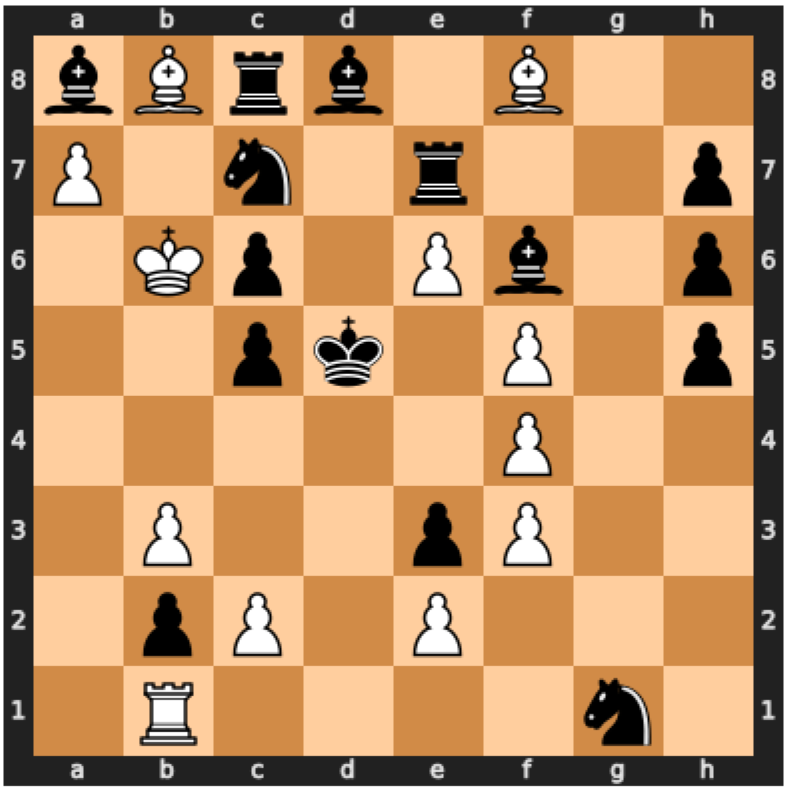


Figure 22: White to play and mate in 290.

This position, as one may observe – see Figure 23 –, is the same as right before white's move 5. Ka5 with one subtle difference: it is now the black's turn to play and not white's. This was exactly the reason behind the white king's "trip" from a5 to d3 and back to a5, i.e. to return to the same position but this time not having to play. In this case, the black have only one plausible move that postpones their mate, which is 21. … h4. Then, the white king starts again in a similar way his trip to d3 and, by repeatedly doing the same

[84] This position is attributed to O. Bláthy (possibly, 1929).

"trick", the black will be gradually forced to push all their pawns on the h file down to rank 1, were the white rook will capture them. So, we arrive at a position where the white play 282. Kb6 and now the black have no pawn to move other than b2, which would be either way sacrificed sooner or later, while their next best option is 282. ... Bb7. Regardless of which of the two moves the black prefer to play first, the white has now a mate in 8 as follows:

> 282. ... Bb7 283. Kxb7 b1=Q 284. Rxb1 Be5 285. Rd1+ Bd4 286. Rxd4 cxd4 287. Kb6 d3 288. a8=Q Rxb8 289. Qxb8 dxe2 290. Qxd8#

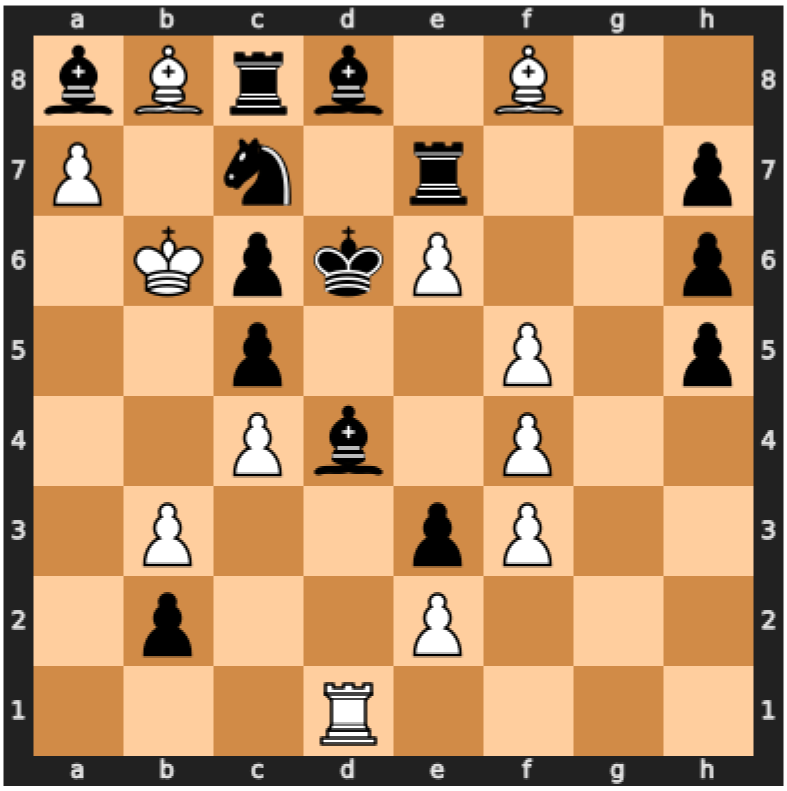


Figure 23: Towards a zugzwang position.

As one may observe, the major strategic feature that appears in the above position is that of *zugzwang*, i.e. a position in which one side is to move, however, any move it has at its disposal will make its position weaker[85]. Nevertheless, as one also may observe, it is only by means of move counting – i.e. game tactics – that one finds the right move

[85] As per the words of one coach with whom we discussed about zugzwang, "it is the fact that tempo in chess may also be of negative value - i.e. it would be better *not* to play in your turn – that makes the game so complicated".

sequence so as to properly take advantage of this strategic feature that appears (potentially) on the board after 17 accurate moves have been played.

One may object that the above position is not a legal one[86] and that, either way, it is an extreme case which should not affect our methodology in general – this is at least what we did in several conversations. However, we were provided with a plethora of positions in which tactics should come in the first place in a different way, each time. We decided to present three of them, the ones that seem to be the most accurate representatives of most cases in which tactics come to the foreground, overriding the importance of strategy.

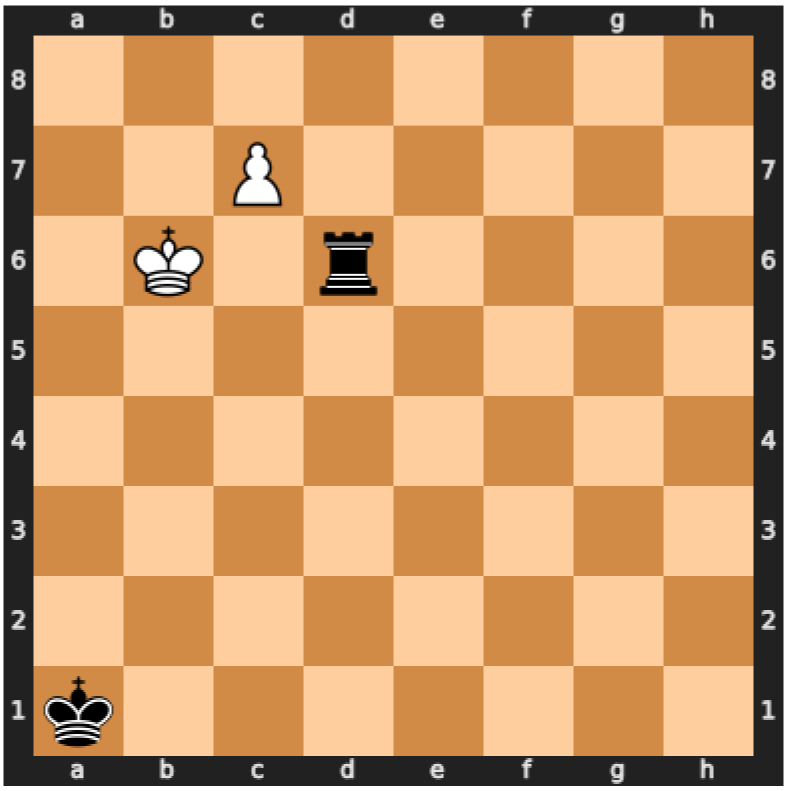


Figure 24: Fewer pieces, yet the same ideas.

At first, consider the position shown in Figure 24 – white to move (Seirawan, 1999: 66). In contrast to the previous one, this is quite minimalistic. While, at a first glance, this position seems to be a draw – the white does not seem to be capable of taking advantage of their pawn at c7 since they are seemingly vulnerable to black rook's checks while the black rook needs to be capable at any time to either check the white king or control the

[86] Indeed, observe that the white has two same-coloured bishops which could happen only if some white pawn had been promoted to one of the two bishops. However, all eight white pawns are on the board.

white pawns threat to promote to c8. However, there exists a winning sequence for white, namely:

1. Kb5! Rd5+ 2. Kb4! Rd4+ 3. Kb3! Rd3+ 4. Kc2!

At this point, see Figure 25, the white seems to have won since the black rook can give no more check nor threaten with a fork or so. However, black has 4. ... Rd4!! which, in case of 5. c8=Q gives the black the possibility to play 4. ... Rc4!! which forces the white to capture the black rook and leads to stalemate. However, white can tackle this threat as follows: 5. c8=R!! which avoids stalemate in case of 5. ... Rc4+? and leads to a forced win for white.

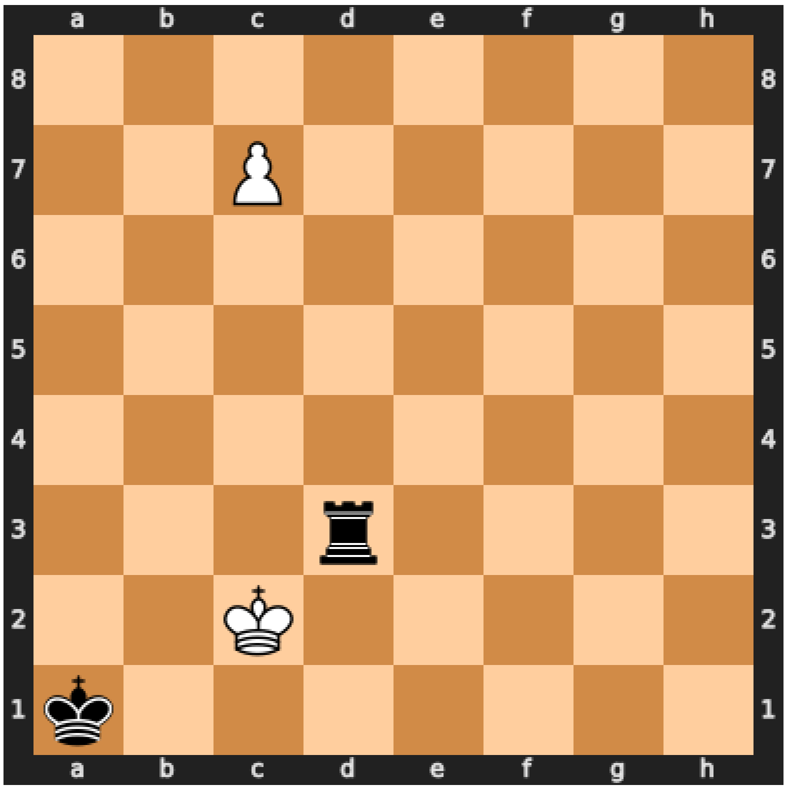


Figure 25: An almost winning position for white.

Again, in the above position we see how the path to victory for the white was not dependent on the strategic attributes of their position – i.e. the fact that they had a pawn ready to be promoted on the seventh line – but on move counting.

Our next position comes from an actual game, where, quoting again one of our interviewees, "strategy can lead you to a winning position but it is up to move counting and tactics to make a win out of it". As shown in Figure 26 (Vuković, 1993: 240), the white has a clear strategic feature they would like to take advantage of, which is the pressure on black's pawn at h7. As one may also observe, in order to further increase pressure on this pawn, the white has several options – strategically equivalent as far as that specific feature is concerned. Namely, 1. Nf6, 1. Neg5 and 1. Nfg5 all lead to some piece of the white putting more pressure on h7. However, as we shall see next, it is the white bishop at g2 that will be the right choice. Namely, the game went as follows:

1. Nfg5! fxg5 2. Nf6! Bxf6 3. Be4 1-0,

Since, in any case, the white mates at h7.

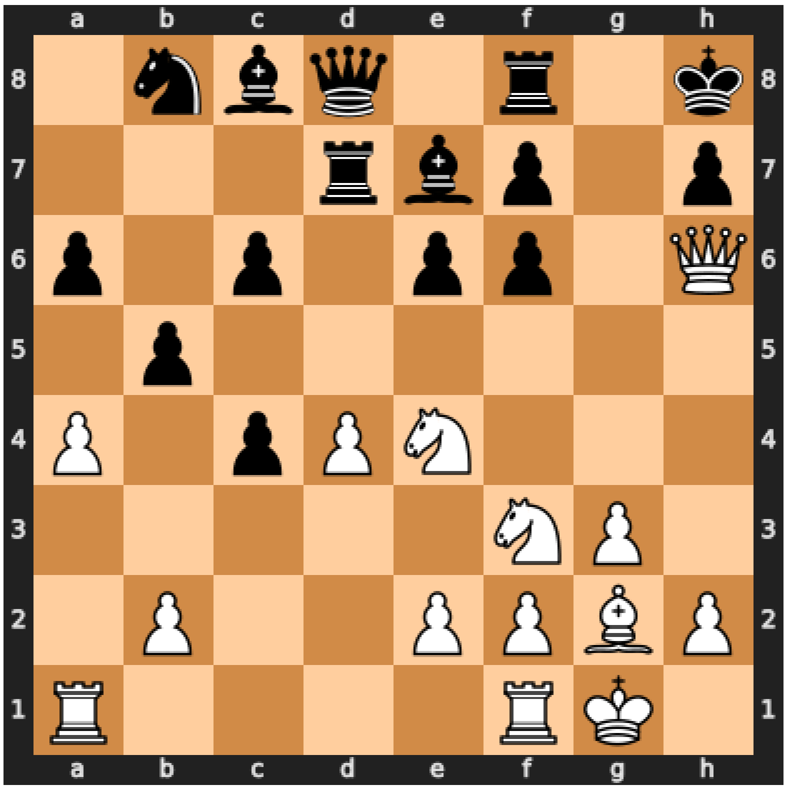


Figure 26: Capablanca - Nimzowich, 1928, white to move and win.

What is of high importance here is the move sequence itself. Starting with 1. Nf6 would not work since then, trying to mimic the above, we would have 1. ... Bxf6 2. Ng5 Bxf5 and now the black can defend white's 3. Be4 with f5. Also, even which of the two knights moves to g5 initially matters, since, should the white play 1. Neg5 then they would not be capable of playing 2. Nf6.

As for now, we have examined positions at which strategic superiority by itself does not suffice so as to ensure victory and precise move counting is needed. Our last example will present the previous idea in an extreme case, showing how a strategically sound move leads to defeat due ignoring the position's tactics. So, let us consider the position shown in Figure 27 - black to move (Seirawan, 2003: 246). There, according to the coach that proposed us this position as well as Seirawan himself, the black's strategic plan should involve blocking white's game on the queen's side so as to shift the game towards the king's side and the centre, where the black seem to be well-positioned. Under this perspective, black's move in the game, 1. ... a5 is sound and actually blocks white pawns in the queen's side[87].

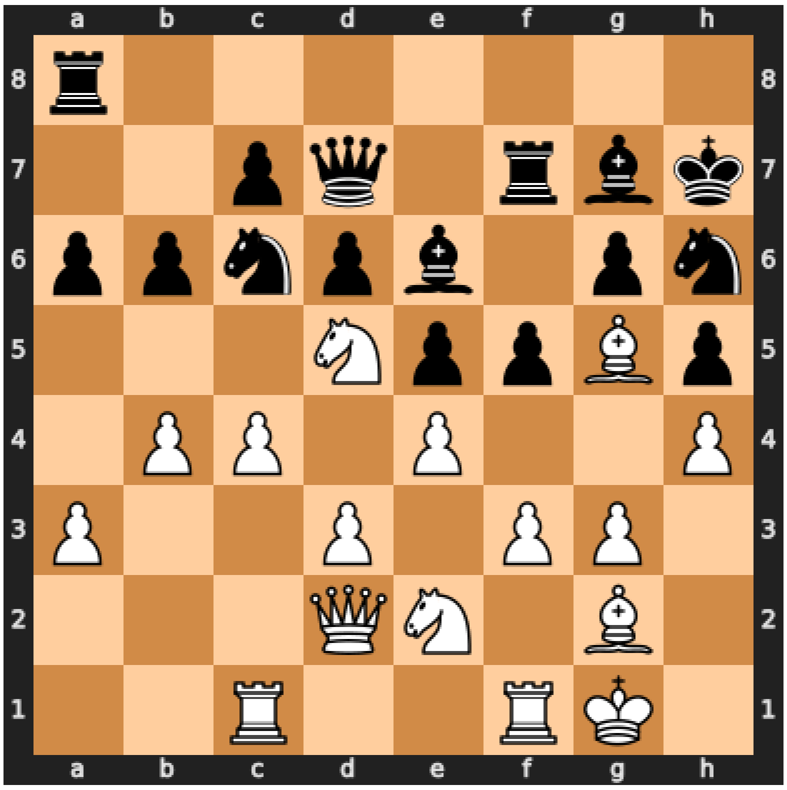


Figure 27: Seirawan - van Wely, 1992, black to move.

However, black's move ignore the position's special tactical features, which allow for the following move sequence:

2. b5 Nd8 3. exf5! Bxf5 4. g4! hxg4 5. fxg4 Nxg4 6. Nxb6 cxb6 7. Bxa8 Qc8 8. Bf3

Now, the white have managed to get an advantage in material which proved enough to lead them to victory.

[87] In case the white play 2. bxa5 then the black can create substantial counter-play on the queen's side by 2. ... Nxa5 which leaves the white with a useless semi-open b rank and a hanging pawn at a3.

While black’s move was not mistaken in terms of strategy, it was highly erroneous with respect to the position’s tactical attributes. The white, taking advantage of their white bishop on the h1-a8 diagonal – i.e., a strategic feature on the board - found an appropriate move sequence so as to capture the black’s rook at a8 by offering their knight.

As explained by many experts we contacted, such positions are characteristic of the interplay between strategic and tactical playing in chess. As a conclusion, strategy is viewed as a high level heuristic while it remains for tactical game and move counting to actually make any strategic advantage an actual advantage capable of leading to win or draw - depending on the side playing.

As we have mentioned above, since our learning methodology strongly relies on defeasible rules to draw inferences given a certain position on the chessboard, we expected that, by continuous coaching and interaction with a human chess coach, the bot would eventually capture a significant part of its coach’s theory or, at least, a reasonably sufficient part that would allow it to play at an acceptable level.

Discussing the above idea with various experts one of them raised an interesting point. As we were told, a casual part of chess training involves studying hundreds of other games - either with or without a coach, depending on the player’s level - so as to get accustomed to as many positions on the board as possible and, thus, be capable of recognising appropriate conditions under which certain (strategic) features are prior to other. While this actually coincides with core ideas of our methodology, it also implicitly poses a question of high significance to the efficiency of our work: Is it plausible to expect that a human coach will be capable - in terms of time as well as cognitive resources - of describing enough positions so as to allow the trained bot to play at an acceptable level?

The view of most of the experts was this does not seem to be possible, at least not in an efficient way. For instance, a formerly International Master and now chess coach of the Greek Coach Federation informed us that, when it comes to human players that start having little or no knowledge about strategic chess, it takes them about two (2) years in order to study enough positions with their trainees so as to demonstrate efficiently the most usual strategic patterns of the game.

One may argue that when it comes to human coaching the process is more time demanding since a human may be prone to errors a machine is not - e.g. omitting a rule during the evaluation of a position. Indeed, should two positions P_1 and P_2 be equivalent with respect to our theory, in the sense that they trigger the same rules, then a machine would not need to be exposed to both of them during its coaching, while a human player might need to be repeatedly exposed to strategically equivalent positions so as to minimise the chance that in a future equivalent position rule omission or similar errors will be avoided.

Bearing in mind the above, the question posed above could be reformulated as follows: How many, in terms of order of magnitude, are the estimated positions a bot following the above learning methodology should be exposed to in order to play at an acceptable level? To this question, the answers we received varied, with the general line being, however, that the absence of explicit move counting in our design would imply that the final number of positions our bot should study with a coach would be insufficiently many.

All in all, we could say that, while our approach as designed and presented to chess experts seemed plausible up to some certain extent, what troubled them most was the absence of an explicit move counting mechanism as well as whether it is possible, under the current methodology, to capture all, or at least the majority of, the tactical aspects of the game[88].

[88] We also received some feedback from one chess coach about transparency and how this could facilitate human-to-human chess coaching. Since it is a suggestion for a future extension of our work we will discuss it in the next chapter, in section 6.1, were possible future steps are presented.

Chapter 6
Conclusion

In this chapter we will discuss how the results we obtained from our scalability experiments as well as the feedback we received from the chess community will determine any future work. Additionally, we present a synopsis of our work as well as a brief summary of its evaluation as a whole. More precisely, this chapter is structured as follows: (i) in section 6.1 possible future steps that could be taken based on the results and views presented in chapter 5 are discussed; (ii) in section 6.2 we conclude.

6.1 Future Steps

Based on what was presented in chapter 5, in this section we shall discuss several directions towards which we could extend our current work.

6.1.1 Technical aspects and scalability

In terms of the technical part of our work, as it is presented in chapter 4, sections 4.1 and 4.2, we intend to thoroughly re-examine build time against all the already examined parameters as well as exception depth and any other parameter that may be considered reasonable to examine. On the one hand, we are aiming in generating more realistic synthetic datasets since the ones used for the purposes of this thesis, while reasonably diverse, also had several weak points. For instance, each knowledge base contained only rules of a certain body length, which is quite unrealistic to occur in knowledge bases generated by human users.

On the other hand, we also aim to examine the consistency of any results we have already found as well as any that will be found using synthetic data with respect to data generated by users in the context of chess. As our discussions with domain experts have revealed, chess is of such complexity that allows for quite diverse knowledge bases to emerge, given that one's goal is to express high strategic concepts - e.g. zugzwang. As a result, we expect that the emerging knowledge bases will allow for us to manipulate effectively any parameter of the game we wish to.

6.1.2 Extensions of the current methodology

Based on the feedback we have received from the chess community, several directions towards which we could search for extensions of the current Machine Coaching interaction protocol have emerged. To begin with, as shown in (Michael, 2019: 85), learning is guaranteed to be efficient under certain conditions which, among other, also require for the rules contained in a knowledge base $k = (\rho, \prec)$ to be *linearly* ordered with respect to the knowledge base's priority relation. However, with chess this may not be the case. So, it would be interesting as well as useful to explore whether more relaxed conditions regarding $\prec$ ceteris paribus would also lead to efficient learning as well as, in case linearity is needed, what other changes, e.g. in the complexity of the current interaction protocol would lead to efficient learning using non-linear priority relations under the current learning semantics.

Another direction would be that of examining in which ways could our current choices from the ASPIC+ semantics be extended and/or altered as well what the effects of such changes are to the efficiency of reasoning and argumentation within them. For instance, in the domain of chess we may need, as indicated in chapter 3, section 3.3, some rules to be declared as strict. In this case, we would need to examine how this introduction of rules of different structure would lead to deviations from the already declared semantics. Moreover, it would also be necessary to examine up to what extent this affects the efficiency of argumentation and, consequently, that of learning.

6.1.3 User Interface and Interaction

As far as the existing user interface is concerned, it serves more as a way to demonstrate our work rather than an application that could be used by chess players and coaches directly, without at first providing some sort of training. On the one hand, we could enrich the existing GUI with more functionalities related to chess based on suggestions we received about how our GUI could be improved e.g. by including more graphic features when it comes to moving a piece - such as arrows indicating possible moves and so on.

On the other hand, we should also focus on how explanations are presented to human users, given that, among others, explainability and interpretability are among our work's core goals. As for now, as one may see in chapter 4, section 4.4, the user has access to the entire argument that has led to the execution of a move, however it is presented as a list of rules expressed in the first order language we have defined in chapter 3, section 3.2. Evidently, this format affects the quality of interaction since it increases the cognitive load of the user, who has to first "translate" the machine's output to natural language and then proceed in understanding its actual meaning. As a result, it seems mandatory to seek for ways in which arguments could be presented in a more user-friendly way, allowing for the users to allocate their cognitive resources in the coaching process itself and not its technical aspects.

However, even if an argument is presented in natural language, in a user-friendly way, it may still provide unnecessary cognitive load to a user. Imagine, for instance, an argument with a crown rule of the form:

"Since this is a move that leads the opponent to a zugzwang, I preferred it".

The same argument is also expected to include very "primitive" rules, expressing low-level relations between entities, such as:

"If I play with white and a move moves a white piece then tag this move as mine".

Evidently, chess experts coaching a bot, having themselves defined notions such as *my move* or *zugzwang* would not need to constantly be informed about how they are defined in each of the machine's explanations. As a result, in later versions of our application we may allow for arguments to be presented gradually, as per the user's request. Namely, we may adopt the following methodology:

1. Once a move is played by the bot, it provides as an explanation only the crown rule of the argument that led to that move being suggested.
2. On condition that the user requests some further explanation about some of the crown rule's antecedents, they are presented with the rules that led to them. In other words, they are presented with the crown rules of each sub-argument that supports the antecedents they requested further explanation for.
3. Repeat step 2 while the user requests explanations at a deeper level and until the premises of an argument are returned.

In the above context, the user is allowed to manipulate the amount of information they wish to include in an explanation, exploring the full argument from higher to lower level rules.

6.1.4 Introducing tactics

The aspect of move counting as well as general tactical manipulations throughout a game was, as mentioned in chapter 5, section 5.2, the major concern regarding the efficiency of our methodology. However, given that the designed system has not yet been systematically evaluated in terms of being coached by experts and then playing games against human or even bot players, we cannot but be modest in any assertion we make about how tactical features are treated. What we can say, for sure, is that, indeed, there do exist cases in which our approach is expected to fail but for the case the user had previously instructed the bot to play in some specific way in that position[89]. And, as the feedback we received from the chess community unveiled, such positions are not uncommon. Even if we do not have clear evidence about our system's behaviour with respect to tactical aspects of chess, we can still consider extensions of our current methodology that could allow for tactics to be introduced more actively.

At first, we could substitute random selection of moves among the ones suggested by the bot by a more sophisticated selection process, e.g. based on alpha-beta pruning or any other known methodology for adversarial AI agents that we consider suitable for our purposes. Thus, we would also remove any randomness from our agent's behaviour which could also result in hybrid explanations consisting of two parts were:

1. The first part will be an argument that led to the suggestion of a set of moves M to be played by the bot.
2. The second part will consist of the evaluation function's score or any other metric we consider appropriate, given the methodology we have selected.

However, even if the above approach is proved to lead to some improvements in our system's performance as far as its chess playing level is concerned, it also reduces our model's transparency and interpretability – as defined in (Arrieta et al., 2020: 84) –, given the fact that some component of its algorithm is neither known to the end user nor efficiently describable.

Another way to introduce move counting explicitly in our system while at the same time we do not deviate from our initial aim of designing an interpretable system would be, at

[89] For instance, one way to achieve this is by describing a specific position on the chessboard through a rule and then, using a more imperative approach, overriding any previous knowledge given that position appears on the board and explicitly asking for certain moves to be played.

first, to extend the built-in predicates our language gives access to so as to include more tactical attributes of the board. These attributes could include, but not restrict to forks, pins, x-ray attacks and, in general any tactical feature of the game chess experts may suggest us. By doing so, we provide the chance to the user to also explicitly introduce chess tactics in rules, thus allowing for a wider set of behaviours as well as game positions to be described.

Moreover, we could also allow for the machine to count moves in the way described above but also further extend our language with the necessary predicates so as to build rules that could modify and control the extent to which the machine utilises its move counting. Namely, as a chess coach may instruct a player to count up to three main variants in any position while not exceeding the depth of 5 double moves during the opening stage of a game, in the same way we could introduce two new built-in predicates, let search_depth(·) and variants_count(·) which would capture the above parameters of move search. Similarly, we could allow for any other predicate which seems plausible to chess experts and which also resonates with common instructions a coach would provide to their trainees about move counting.

Should we adopt the above or any similar methodology, it would be interesting to measure whether and, if so, by what extent is that "hybrid" bot superior in terms of chess playing against our current approach as well as at what level does its behaviour resemble that of a human trainee.

6.1.5 Utilising our work in chess education and human coaching

During the interviewing process one expert came up with a suggestion – in the form of a query of whether such a thing could be possible – about utilising our system, possibly after making its GUI more user-friendly, as discussed above, in human chess coaching. More precisely, what was suggested was to use the developed system as a cognitive coaching assistant in the following way:

1. At first, a human coach trains the bot up to some desired level – possibly not equivalent to the coach's one.
2. Then, the bot is used in coaching human players by letting them play either against it or other players and on each move played by a human player, it also provides in the form of a suggestion the move that it would have played in that position alongside with the explanation.

The rationale behind the above proposal was that, should a bot be capable of capturing a coach's theory about the game, even up to some certain extent, then this would allow for it to substitute the coach in simple tasks within a chess classroom or outside the it, as an assistant for students with whim they could study along and who (i.e. the assistant) would have a theory about the game which resonates with that of the human coach.

6.2 Conclusions

In this thesis we presented a complete methodology as well as its implementation that accommodate a transparent and interpretable way of coaching a computer to play chess. Our motivation was the fact that most of the current as well as past approaches on computer chess were designed with performance as their main principle while at the same time they did not allow for almost no fragment of human knowledge and expertise on the game to assist the machine's effort. Furthermore, most contemporary approaches have adopted black-box machine learning methodologies which, consequently, lead to the behaviour of the bot being almost impossible to be interpreted by humans, at least without introducing any external tools.

In order to address these issues appearing in black-box approaches, we adopted the Machine Coaching paradigm (Michael, 2019: 82-85). Under that, a machine is learning taking advantage of human knowledge about some certain domain of application by learning from it. Namely, the human coach, according to the machine's actions as well as the corresponding explanations it returns about them, provides contextual advice based on their own knowledge which are accordingly stored in the machine's knowledge base and alter accordingly its behaviour.

The above learning paradigm was implemented in java by first implementing a generic first order language through which a user can interact with the machine. After that, all the necessary algorithms were implemented utilising the above language as well as extending it up to some point so as to allow for several desirable additional functionalities. The implementation of all the necessary learning functionalities was followed by the development of a domain specific user interface which allowed for a machine to be trained to play chess using the Machine Coaching methodology.

Having implemented all the above, we assessed them on two orthogonal directions. At first, as far as the learning mechanism is concerned, we measured its efficiency with respect to several parameters and found results conformant with the corresponding predictions of the theory of Machine Coaching. Secondly, we came into contact with numerous chess experts – either professional players or chess coaches – in order to explore how plausible and applicable our approach was as far as the specific domain of chess is concerned. The feedback we received from chess experts on the one hand indicated that, indeed, our approach was applicable to the game of chess while, on the other hand, it also provided material upon which we should reflect so as to further improve our designed system as well as possibly extend the currently used learning semantics accordingly.

Appendix A
Built-in Predicates

In this Appendix we present all built in predicates as of the time this thesis was written.

`is_at(Piece,Colour,Square)`

It means that a piece of type `Piece` and of colour `Colour` lies on square `Square`.

For instance, in the initial board position, `is_at(king,white,e1)` describes the initial position of the white King.

Meaningful constants for each variable are:

- `Piece: pawn, knight, bishop, rook, queen, king.`
- `Colour: black, white.`
- `Square: a1, a2,..., h7, h8.`

`from_square(Move,Square)`

It means that a move `Move` starts from square `Square`.

For instance, in the initial board position, `move_starts_from(e2e4,e2)` describes that move `e2e4` starts from `e2`.

Meaningful constants for each variable are:

- `Move: any move in uci form (e.g. g1f3).`
- `Square: a1, a2,..., h7, h8.`

`to_square(Move,Square)`

It means that a move `Move` ends to square `Square`.

For instance, in the initial board position, `move_ends_to(e2e4,e4)` describes that move `e2e4` ends to `e4`.

Meaningful constants for each variable are:

- `Move: any move in uci form (e.g. g1f3).`
- `Square: a1, a2,..., h7, h8.`

`move_played_by(Move,Colour)`

It means that a move `Move` is played by a player with colour `Colour`.

For instance, in the initial board position white's move `e4` is described by `move_played_by(e2e4,white)`.

Meaningful constants for each variable are:

- `Move: any move in uci form (e.g. g1f3)`
- `Colour: black, white.`

`moves(Move,Piece)`

It means that a move `Move` moves piece of type `Piece`.

For instance, in the initial board position, `moves(g1f3,knight)` describes the fact that move `g1f3` moves a knight.

Meaningful constants for each variable are:

- `Move: any move in uci form (e.g. g1f3).`
- `Piece: pawn, knight, bishop, rook, queen, king.`

`plays_as(Colour)`

It means that the bot plays with colour `Colour`.

For instance, the fact that the bot plays as black is described by `plays_as(black)`.

Meaningful constants for each variable are:

- `Colour: black, white.",`

`has_kingside_castling_rights(Colour)`

It means that player of colour `Colour` has kingside castling rights.

For instance, after: 1. e4 e5 2. Nf3 Nc6 3. Bc4 white has kingside castling rights, which is described by `has_kingside_castling_rights(white)`

Meaningful constants for each variable are:

- `Colour: black, white.`

`has_queenside_castling_rights(Colour)`

It means that player of colour `Colour` has queenside castling rights.

For instance, after: 1. d4 d5 2. c4 e6 3. Nc3 Nf6 4. Bg5 Bb4 5. Qc2 white has queenside castling rights, which is described by `has_queenside_castling_rights(white)`.

Meaningful constants for each variable are:

- `Colour: black, white.`

`is_kingside_castling(Move)`

It means that move `Move` is a kingside castling.

Meaningful constants for each variable are:

- `Move: any move in uci form (e.g. g1f3).`

`is_queenside_castling(Move)`

It means that move `Move` is a queenside castling.

Meaningful constants for each variable are:

- `Move: any move in uci form (e.g. g1f3).`

`move_count(Integer)`

It means that the current move count (i.e. the number of pairs of white-black moves) equals `Integer`.

Meaningful constants for each variable are:

- `Integer: 1,2,3,…`

`is_capture(Move)`

It means that move `Move` is a capture move (en passant capture included).

Meaningful constants for each variable are:

- `Move: any move in uci form (e.g. g1f3).`

`pinned(Square,Colour)`

It means that the square `Square` is pinned to its king of colour `Colour`.

For instance, after 1. e4 e5 2. Nf3 Nc6 3. Bb5 d6 the black knight at c6 is pinned by the white bishop at b5.

Meaningful constants for each variable are:

- `Square: a1, a2,..., h7, h8.`
- `Colour: black, white.`

`is_promotion_to(Move,Piece)`

It means that move `Move` is a promotion to `Piece`.

For instance, `is_promotion_to(c7c8,queen)` denotes the promotion of a pawn to a queen by moving from c7 to c8.

Meaningful constants for each variable are:

- `Move: any move in uci form (e.g. g1f3).`
- `Piece: pawn, knight, bishop, rook, queen, king.`

`is_attacked_by(Colour,Square,Piece)`

It means that a square `Square` is attacked by a piece of type `Piece` and colour `Colour`.

Meaningful constants for each variable are:

- `Piece: pawn, knight, bishop, rook, queen, king.`
- `Square: a1, a2,..., h7, h8.`
- `Colour: black, white.`

`is_check(Move)`

It means that move `Move` is a check move.

Meaningful constants for each variable are:

- `Move: any move in uci form (e.g. g1f3).`

`is_checker(Piece,Square)`

It means that piece `Piece` at square `Square` is currently giving a check.

For instance, after: 1. e4 d5 2. Bb5+ it holds that `is_checker(bishop,b5)`.

Meaningful constants for each variable are:

- `Piece: pawn, knight, bishop, rook, queen, king.`
- `Square: a1, a2,..., h7, h8.`

`square_file(Square,File)`

It means that square `Square` lies on file `File`.

For instance, `square_file(e4,e)` describes the fact that square `e4` lies on file `e`.

Meaningful constants for each variable are:

- `Square: a1, a2,..., h7, h8.`
- `File: a, b,..., h.`

`square_rank(Square,Rank)`

It means that square `Square` lies on rank `Rank`.

For instance, `square_rank(e4,4)` describes the fact that square `e4` lies on rank `4`.

Meaningful constants for each variable are:

- `Square: a1, a2,..., h7, h8.`
- `Rank: 1, 2,..., 8.`

`is_en_passant(Move)`

It means that move `Move` is an en passant capture.

Meaningful constants for each variable are:

- `Move: any move in uci form (e.g. g1f3).`

`?<(X,Y)`

Built-in predicate that is interpreted as $X < Y$.

For instance, `?<(3,4)`.

Meaningful constants for both `X` and `Y` are any integers or double precision numbers.

`?=(X,Y)`

Built-in predicate that is interpreted as $X = Y$.

For instance, `?=(5,5)`.

Meaningful constants for both X and Y are any integers or double precision numbers.

`suggest(Move)`

Built-in predicate that informs the system that move `Move` should be suggested (or should not be suggested, in case it appears negated). Typically, this predicate is used in a rule's head.

References

Anantharaman, T., Campbell, M., Hsu, F. (1988), Singular Extensions: Adding Selectivity to Brute Force Searching, *AAAI Spring Symposium on Computer Game Playing.*

Anderson, H., L. (1986), Metropolis, Monte Carlo, and the MANIAC*, Los Alamos Science* (14).

Arrieta, A., B., et al. (2020), Explainable Artificial Intelligence (XAI): Concepts, Taxonomies, Opportunities and Challenges toward responsible AI, *Information Fusion* (58), 82-115.

Atkin, L., R., Slate, D., J. (1983), Chess 4.5: the Northwestern University Chess Program, *Chess Skill in Man and Machine,* (2nd edition), 82-118.

Atkinson, G., W. (1998), Chess and machine intuition, *Intellect Books*.

Bauer, F., L., Wössner, H. (1972), The "Plankalkül" of Konrad Zuse: a Forerunner of Today's Programming Languages, *Communications of the ACM 15* (7), 678-685.

Berliner, H. (1989), Hitech Chess: From Master to Senior Master with no Hardware Change, *International Workshop on Industrial Applications of Machine Intelligence and Vision*, 12-21.

Bernstein, A., Roberts, M., de V. (1958), Computer vs. Chess-Player, *Scientific American 26* (7).

Bowden, B., V. (1957), Faster Than Thought, *Pitman, Virginia*.

Campbell, M., Joseph Hoane Jr., A., Hsu, F. (2002), Deep Blue, *Atrificial Intelligence 134*, 57-83.

Champernowne, D., Obituary (2000), "David Champernowne (1912-2000), *ICGA Journal 23* (4), 262.

Copeland, B., J. (2004), The Essential Turing: Seminal Writings in Computing, Logic, Philosophy, Artificial Intelligence and Artificial Life plus the Secrets of Enigma, *Oxford University Press*, *Oxford*.

Douglas, J., R. (1978), Chess 4.7 versus David Levy, *BYTE Magazine 3* (12), 84-91.

Douglas, J., R. (1979), Grandmaster Walter Browne versus Chess 4.6, *BYTE Magazine 4* (1), 110-115.

Dung, P., M. (1995), On the Acceptability of Arguments and its Fundamental Role in Nonmonotonic Reasoning, Logic Programming and *n*-Person Games, *Artificial Intelligence 77*, 321-357.

Enderton, H., B. (2012), A Mathematical Introduction to Logic, 2nd edition, *Cretan University Press*, translation, Papadoggonas, I., editors, Kyrousis, E., Pheidas A.

Fishburn, J., P. (1980), An Optimization of Alpha-Beta Search, *ACM SIGART Bulletin*, 29-30.

Frey, P., W. Atkin, L., R. (1978), Creating a Chess Player, *BYTE Magazine 3* (10), 182-191.

Gardnder, H. (2005), Oral History of Richard Greenblatt, *Interview, Recorded: January 12, 2005. Boston, MA.*

Greenblatt, R., Eastlake III, D., E., Crocker, S., D. (1969), The Greenblatt Chess Program, *Artificial Intelligence 174,* Massachusetts Institute of Technology, Cambridge, Massachusetts, 801-810.

Gumpel, C., G. (1889), "Memphisto", the Marvelous Automaton, Exhibited at the International Theatre, Exposition Universelle, Paris, 1889, *T. Pettit & Co.*

Harding, T. (2012), Eminent Victorian Chess Players: Ten Biographies, *McFarland & Company, Inc. London.*

Hapgood, F. (1982), Computer chess bad – human chess worse, *New Scientist 23* (30), 827-830.

Hsu, F., Anantharaman, T., Campbell, M., Nowatzyk, A. (1990), A Grandmaster Chess Machine, *The Scientific American*, 44-50.

Jennings, P., R. (1978), The Second World Computer Chess Championships, *BYTE Magazine 3* (1), 108-119.

Klein, M. (2014), Stockfish Outlasts "Rybkamura", retrieved at 16/09/2020: https://www.chess.com/news/view/stockfish-outlasts-nakamura-3634

Knuth, D., E., Pardo, L., T. (1976), The Early Development of Programming Languages, *Encyclopedia of Computer Science and Technology*.

Kotok, A. (1962), A Chess Playing Program for the IBM 7090 Computer, *Thesis (B.Sc.) Massachusetts Institute of Technology, Dept. of Electrical Engineering*.

Lasker, E. (1946), Common Sense in Chess, *David McKay Company Inc,* New York.

Levitt, G., M. (2000), The Turk, Chess Automaton, *Jefferson, N.C.: McFarland*.

Levy, D., N., L. (2013), Computer Chess Compendium, *Springer Science and Business Media*.

Lipton, Z., C. (2018), The Mythos of Model Interpretability, *ACM Queue 16*, 31-57.

Mashey, J., (2005), Oral History of Ken Thompson, *Interview, Recorded: February 8, 2005, Mountain View, California.*

McCorduck, P. (2004), Machines Who Think: A Personal Inquiry into the Historry and Prospects of Artificial Intelligence*, A. K. Peters*.

Michael, L. (2019), Machine Coaching, *IJCAI 2019 Workshop on Explainable Artificial Intelligence (XAI @ IJCAI 2019)*, 80-86.

Mittman, B. (1980), ICCA Newsletter, *ICCA Journal 3* (1), 1-12.

Montavon, G., Samek, W., Müller, K., R., (2018), Methods for Interpreting and Understanding Deep Neural Networks, *Digital Signal Processing 73*, 1-15.

Montfort, N. (2003), Twisty Little Passages: An approach to interactive friction, *MIT Press*.

Newborn, M. (1977), PEASANT: An Endgame Program for Kings and Pawns, *Chess Skill and Man and Machine*, Springer-Verlag, New York, 119-129.

Prakken, H. (2010), An Abstract Framework for Argumentation with Structured Arguments, *Argument & Computation*, 1:2, 93-124.

Pritchard, D., B. (1994), The Encyclopedia of Chess Variants, *Games and Puzzles Publications*.

Rojas, R. (1997), Konrad Zuse's Legacy: The Architecture of the Z1 and Z3, *IEEE Annals of the History of Computing 19* (2).

Romstad, T., Costalba, M., Kiiski, J. (2008), Stockfish, retrieved on 22/09/2020: https://github.com/ddugovic/Stockfish

Schaeffer, J. (1997), One Jump Ahead. *Springer*.

Seirawan, J. (1999), Winning Chess Endings, *Glouchester Publishers plc,* London.

Seirawan, J. (2003), Winning Chess Strategies, *Glouchester Publishers plc,* London.

Shannon, C. (1950), Programming a Computer for Playing Chess, *Philosophical Magazine 41* (314).

Silver, D., et al. (2017), Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm, arXiv:1712.01815v1.

Standage, T. (2002), The Turk: The Life and Times of the Famous Eighteenth-Century Chess Playing Machine, *Walker,* New York.

Valiant, L., G. (1984), A Theory of the Learnable, *Communications of the ACM 27* (11), 1134-1142.

Vuković, V. (1993), Art of Attack in Chess, *Cadogan Books plc,* London.

Wiener, N. (1948), Cybernetics, *Wiley John*.