University of Ljubljana
Faculty of computer and information science

Simon Stoiljkovikj

Computer-based estimation of the difficulty of chess tactical problems

bachelor's thesis

undergraduate university study programme
computer and information science

Mentor: Matej Guid, PhD

Ljubljana 2015

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Simon Stoiljkovikj

Računalniško ocenjevanje težavnosti taktičnih problemov pri šahu

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in informatika

Mentor: doc. dr. Matej Guid

Ljubljana 2015

The text is formatted with the text editor LaTeX.

Faculty of Computer and Information Science issues the following thesis:

In intelligent tutoring systems, it is important for the system to understand how difficult a given problem is for the student. It is an open question how to automatically assess such difficulty. Develop and implement a computational approach to estimating the difficulty of problems for a human. Use a computer heuristic search for building search trees that are meaningful from a human problem solver's point of view. Focus on analyzing such trees by using machine-learning techniques. Choose chess tactical problems for the experimental domain. Evaluate your approach to estimating the difficulty of problems for a human, and present your findings.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Pri inteligentnih tutorskih sistemih je pomembno, da sistem razume, kako težak je določen problem za učenca. Kako samodejno oceniti tovrstno težavnost, ostaja odprto vprašanje. V svojem delu razvijte in implementirajte algoritmičen pristop k ugotavljanju težavnosti problemov za človeka. Posebej se posvetite uporabi računalniškega hevrističnega preiskovanja za gradnjo preiskovalnih dreves, ki so smiselna z vidika osebe, ki problem rešuje. Osredotočite se na računalniško analizo tovrstnih dreves, pri tem pa uporabite tehnike strojnega učenja. Za raziskovalno domeno izberite taktične probleme pri šahu. Izbrani pristop k ocenjevanju težavnosti problemov za človeka eksperimentalno ovrednotite in predstavite ugotovitve.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Simon Stoiljkovikj, z vpisno številko **63110393**, sem avtor diplomskega dela z naslovom:

Računalniško ocenjevanje težavnosti taktičnih problemov pri šahu

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Mateja Guida,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 16. februarja 2015 Podpis avtorja:

Hvala moji družini, da me je v letih študija in pred tem podpirala moralno, finančno in z iskreno ljubeznijo.

Hvala tudi mojim najboljšim prijateljem, saj veste, kdo ste.

Najlepša hvala mentorju, doc. dr. Mateju Guidu za njegov čas, ideje in potrpežljivost pri izdelavi diplomske naloge.

Thanks to my family that, in the years of this study and even prior to, supported me morally, financially and with sincere love.

Also, thanks to my best friends, you know who you are.

Special thanks go to my mentor, Matej Guid, PhD, for his time, ideas and patience during the writing of the thesis.

Contents

Table of acronyms

kratica	angleško	slovensko
CP	centipawns	stotinko kmeta
CA	classification accuracy	klasifikacijska točnost
AUC	area under curve	površina pod krivuljo
ROC	receiver operating characteristic	značilnost delovanja sprejemnika
ITS	intelligent tutoring system	inteligentni tutorski sistem

Abstract

In intelligent tutoring systems, it is important for the system to understand how difficult a given problem is for the student; assessing difficulty is also very challenging for human experts. However, it is an open question how to automatically assess such difficulty. The aim of the research presented in this thesis is to find formalized measures of difficulty. Those measures could be used in automated assessment of the difficulty of a mental task for a human. We present a computational approach to estimating the difficulty of problems in which the difficulty arises from the combinatorial complexity of problems where a search among alternatives is required. Our approach is based on a computer heuristic search for building search trees that are "meaningful" from a human problem solver's point of view. This approach rests on the assumption that computer-extracted "meaningful" search trees approximate well to the search carried out by a human using a large amount of his or her pattern-based knowledge. We demonstrate that by analyzing properties of such trees, the program is capable to automatically predict how difficult it would be for a human to solve the problem. In the experiments with chess tactical problems, supplemented with statistic-based difficulty ratings obtained on the ChessTempo website, our program was able to differentiate between easy and difficult problems with a high level of accuracy.

Keywords: task difficulty, human problem solving, heuristic search, search trees, chess tactical problems.

Povzetek

Pri inteligentnih tutorskih sistemih je pomembno, da sistem razume, kako težak je določen problem za učenca. Ocenjevanje težavnosti problemov predstavlja izziv tudi domenskim strokovnjakom. Kako samodejno oceniti tovrstno težavnost, ostaja odprto vprašanje. Namen raziskave, predstavljene v tem diplomskem delu, je razviti algoritmičen pristop k ugotavljanju težavnosti, ki bi ga lahko uporabljali pri avtomatiziranem ocenjevanju težavnosti problemov za človeka. Osredotočili se bomo na ocenjevanje težavnosti problemov, pri katerih težavnost izvira iz kombinatorične kompleksnosti in kjer je potrebno preiskovanje med alternativami. Pristop temelji na uporabi hevrističnega računalniškega preiskovanja za gradnjo preiskovalnih dreves, ki so "smiselna" z vidika osebe, ki problem rešuje. Ta pristop predvideva, da se računalniško pridobljena "smiselna" preiskovalna drevesa relativno dobro ujemajo s preiskovanjem, ki ga pri istih obravnavanih problemih izvedejo ljudje. Le-ti pri tem uporabljajo predvsem znanje, ki tipično temelji na pomnjenju številnih naučenih vzorcev. Pokazali bomo, da je s pomočjo analize lastnosti tovrstnih "smiselnih" dreves računalniški program sposoben samodejno napovedati, kako težak za reševanje je določen problem. Za eksperimentalno domeno smo izbrali šahovske taktične probleme. Uporabili smo taktične probleme, kjer smo imeli na voljo statistično utemeljene ocene težavnosti, pridobljene na spletni strani Chess Tempo. Naš program je bil sposoben z visoko stopnjo natančnosti ločevati med enostavnimi in težkimi problemi.

Ključne besede: težavnost problema, človeško reševanje problemov, hevri-

stično preiskovanje, preiskovalna drevesa, šahovski taktični problemi.

Razširjeni povzetek

Eden od perečih raziskovalnih izzivov je modeliranje težavnosti problemov za človeka, npr. z uporabo tehnik strojnega učenja. V tej diplomski nalogi se bomo osredotočili na šahovsko igro oz. bolj natančno, na taktične probleme pri šahu. Kdorkoli je kdaj reševal tovrstne probleme, bodisi iz šahovske knjige bodisi na namenski spletni igralni platformi, bo takoj razumel zakaj je pomembno, da igralec dobiva probleme ustrezne težavnosti glede na njegovo predznanje. Gre za podoben problem kot npr. pri inteligentnih tutorskih sistemih, torej za oceno težavnosti problema in primerjavo te težavnosti s učenčevo sposobnostjo reševanja problemov, še preden mu dani problem ponudimo v reševanje. Čeprav smo se osredotočili na le eno domeno (šah), radi bi razvili algoritmični pristop k razumevanju, kaj pri problemih predstavlja težavo za reševanje pri ljudeh. Razvoj računalniškega modela težavnosti za taktične šahovske probleme (oziroma tudi za kakršno koli drugo reševanje problemov, ki vključuje drevesa iger), je lahko v pomoč na področjih, kot je npr. razvoj inteligentnih sistemov za poučevanje. Še zlasti, ker je razvoj tovrstnih sistemov drag zaradi odsotnosti posplošenega pristopa za njihovo izdelavo. Priprava izpitov za učence je še eno izmed področij, ki bi imele koristi od tovrstnega modela, saj bi bilo za učitelje pri pripravi izpitov lažje, če bi razumeli kaj je zanje težko. Skratka, korist od avtomatiziranega ocenjevanja težavnosti problemov bi lahko našli povsod, kjer je vključeno poučevanje učencev in še zlasti v manj uveljavljenih domenah, kjer še vedno ne vemo, kaj je pri reševanju problemov predstavlja težave za ljudi in kjer hkrati tudi nimamo dovolj re-

sursov, da bi težavnost ugotavljali "ročno" (brez pomoči strojnega učenja). Poleg tega se je izkazalo, da tudi ljudje sami niso tako dobri pri modeliranju težavnosti [13], torej je avtomatizirano ocenjevanja težavnosti potrebno ne le s finančnega vidika, ampak tudi z vidika zanesljivosti ocen.

Zgolj računski pristop (brez uporabe hevristik) k ugotavljanju težavnosti problemov za ljudi ne bi dal želenih rezultatov. Razlog za to je, da računalniški šahovski programi rešijo taktične probleme pri šahu zelo hitro, navadno že pri zelo nizkih globinah iskanja. Računalnik bi tako preprosto prepoznal večino šahovskih taktičnih problemov za lahke in ne bi znal dobro razlikovati med pozicijami z različnimi stopnjami težavnosti (kot jih dojemajo ljudje). Ocenjevanje težavnosti tovrstnih problemov zato zahteva drugačen pristop in druge algoritme.

Naš pristop temelji na uporabi računalniškega hevrističnega preiskovanja za izgradnjo "smiselnega" drevesa preiskovanja z vidika človeka, ki rešuje dani problem. Želimo pokazati, da je model, pridobljen z analizo lastnosti tovrstnih "smiselnih" dreves preiskovanja, sposoben samodejno napovedovati težavnost problema za ljudi (v izbrani domeni, na katero se model nanaša). Naj poudarimo, da je analizira lastnosti "smiselnih" dreves vodila k bistveno boljšim rezultatom od uporabe strojnega učenja z atributi, temelječih zgolj na uporabi specifičnega domenskega znanja.

V naši raziskavi smo zajeli tip reševanja problemov, pri katerih mora igralec predvideti, razumeti in izničiti dejanja nasprotnikov. Tipična področja, kjer se zahteva tak način reševanja problemov, vključujejo vojaško strategijo, poslovanje in igranje iger. Pravimo, da je šahovski problem taktičen, če rešitev dosežemo predvsem z izračunom konkretnih variant v dani šahovski poziciji in ne s pomočjo dolgoročnih pozicijskih presoj. V diplomski nalogi nas ne zanima sam proces dejanskega reševanja šahovskih taktičnih problemov, ampak predvsem vprašanje, kako težavno je reševanje problema za človeka. Kot osnovo smo vzeli statistično utemeljene ocene težavnosti šahovskih taktičnih problemov, pridobljene na spletni šahovski platformi Chess Tempo (dostopna

na `http://www.chesstempo.com`). Le-te so predstavljale objektivne ocene težavnosti problemov.

Pri umetni inteligenci tipični način predstavljanja problemov imenujemo prostor stanj. Prostor stanj je graf, katerega vozlišča ustrezajo problemskim situacijam, dani problem pa reduciramo na iskanje poti v tem grafu. Prisotnost nasprotnika v veliki meri otežuje iskanje. Namesto, da bi iskali linearno zaporedje akcij v problemskem prostoru, dokler ne dosežemo ciljnega stanja, nam prisotnost nasprotnika bistveno širi nabor možnosti. Pri reševanje problemov, kjer obstaja tudi nasprotnik, je prostor stanj običajno predstavljen kot drevo igre. Pri reševanju problemov s pomočjo računalnika tipično zgradimo le del celotnega drevesa igre, ki se imenuje drevo iskanja, in uporabimo hevristično ocenjevalno funkcijo za vrednotenje končnih stanj (vozlišč) v drevesu iskanja.

Drevesa iger so tudi primeren način predstavljanja šahovskih taktičnih problemov. Pri tipih težav, v katerih težavnost izhaja iz kombinatorične kompleksnosti iskanja med alternativami, je navadno nemogoče za človeka, da bi upošteval vse možne poti, ki bi lahko vodile k rešitvi problema. Človeški igralci zato hevristično zavrnejo možnosti (poteze), ki niso pomembne pri iskanju rešitve določenega problema. Pri tem se opirajo predvsem na svoje znanje in izkušnje. Pravzaprav človeški reševalci problemov (v mislih) pri reševanju problemov zgradijo svoja lastna drevesa preiskovanja (oz. drevesa iskanja). Ta drevesa preiskovanja pa so bistveno različna od tistih, pridobljenih pri računalniškem hevurističnem preiskovanju. Zato smo uvedli t.i. "smiselna" drevesa preiskovanja (človeška drevesa iskanja), ki so običajno bistveno manjša. In kar je najpomembneje, ta drevesa so v glavnem sestavljena iz smiselnih stanj in akcij, ki naj bi ljudi vodila do rešitve problema.

Z namenom, da bi omogočili samodejno ocenjevanje težavnosti problemov za človeka, smo se osredotočili na izgradnjo preiskovalnih dreves, ki so smiselna s stališča reševalca problema. Takšna drevesa bi morala biti sestavljena predvsem iz dejanj, ki naj bi jih človeški reševalec pri reševanju problema vzel v obzir. Implicitna predpostavka pri našem pristopu je, da težavnost šahovskega

taktičnega problema korelira z velikostjo in drugimi lastnostmi "smiselnega" drevesa preiskovanja za dani problem. Pokazali smo, da lahko za pridobitev vrednosti posameznih vozlišč v "smiselnem" drevesu igre za dani problem uporabimo računalniško hevristično preiskovanje. Pri tem ohranimo le tista vozlišča, ki izpolnjujejo določene pogoje (kot so npr. arbitrarno določene mejne vrednosti ocen vozlišč). V diplomskem delu smo pokazali, da je z analizo lastnosti tovrstnih dreves mogoče pridobiti koristne informacije o težavnosti problema za človeka.

Chapter 1

Introduction

1.1 Motivation

One of the current research challenges is using machine learning for modeling the difficulty of problems for a human. In this thesis, we focused on the domain of chess and, more precisely, on tactical chess problems. Whoever tried to solve such a problem, being an example from a book or an online chess playing platform, can understand why it is important for the player to receive a problem with a suitable difficulty level. The problem here is, just like in intelligent tutoring systems (for example), to assess the difficulty of the problem and to compare it with the student's problem solving skill before showing it to the student to solve it. Although we focused on a single domain (chess), we would like to come up with an algorithmic approach for determining the difficulty of a problem for a human, in order to obtain a more general understanding what is difficult for humans to solve. This is a fairly complex question to answer, particularly with limited resources available: a database of ratings for tactical chess problems acquired from the website for solving such problems – Chess Tempo (available at `http://www.chesstempo.com`), chess playing program (or rather, a selection of them, since we experimented with three chess engines: Houdini, Rybka and Stockfish), and Orange,

a visual tool for data mining [9].

Developing a computational model of difficulty for chess tactical problems (or, as we will discuss later, for any problem solving that involves a game tree) would help in areas such as easing the development process of intelligent tutoring systems. These systems are currently expensive to develop due to the lack of a general approach to creating them. Student exam preparation is another topic that would benefit from such a model, since it would be easier for teachers to understand what is difficult for their students and prepare the exams accordingly. In short, anything that involves student learning on a less than well-established basis, where it is unknown what is difficult for humans to solve, and where we also don't have the resources to let humans research this question manually, can benefit from an automated assessment of problem difficulty. Furthermore, it turns out that humans are not that great at modeling the difficulty themselves [13], so a method for determining the difficulty of problems for a human is needed not only from the financial aspect, but also from the aspect of reliability.

1.2 Our approach and contributions

A pure computational-based approach (without the use of heuristics) to determining the difficulty of problems would yield poor results. The reason for this is that computer chess programs tend to solve tactical chess problems very quickly, usually already at the shallowest depths of search. Thus the computer simply "recognizes" most of the chess tactical problems to be rather easy and does not distinguish well between positions of different difficulties (as perceived by humans) [13]. Estimating difficulty of chess tactical problems therefore requires a different approach, and different algorithms. Our approach is based on using computer heuristic search for building meaningful search trees from a human problem solver's point of view. We intend to demonstrate that by analyzing properties of such trees, the model is capable to automatically

predict how difficult the problem will be to solve by humans. It is noteworthy that we got better results from analyzing the game tree properties rather than by analyzing specific chess domain attributes.

1.3 Related work

Relatively little research has been devoted to the issue of problem difficulty, although it has been addressed within the context of several domains, including Tower of Hanoi [17], Chinese rings [10], 15-puzzle [11], Traveling Salesperson Problem [12], Sokoban puzzle [1], and Sudoku [2]. Guid and Bratko [3] proposed an algorithm for estimating the difficulty of chess positions in ordinary chess games. Their work was also founded on using heuristic-search based methods for determining how difficult the problem will be for a human. However, they found that this algorithm does not perform well when faced with chess tactical problems in particular. Hristova, Guid and Bratko [13] undertook a cognitive approach to the problem, namely, will a player's expertise (Elo rating [4]) in the given domain of chess be any indication of whether that player will be able to classify problems into different difficulty categories. They demonstrated that assessing difficulty is also very difficult for human experts, and that the correlation between a player's expertise and his or her perception of a problem's difficulty to be rather low.

1.4 Structure

The thesis is organized as follows. In Chapter 2, we introduce the domain of chess tactical problems and the concept of meaningful search trees. We also describe features that could be computed from such trees, and present our experimental design. Results of the experiments are presented in Chapter 3. We conclude the thesis in Chapter 4.

A note to the reader

Parts of the contents in this bachelor's thesis are also contained in the research paper submitted to the 17th International Conference on Artificial Intelligence in Education (AIED 2015), titled "A Computational Approach to Estimating the Difficulty of a Mental Task for a Human," co-authored with professors Matej Guid, PhD, and Ivan Bratko, PhD, from the Faculty of Computer and Information Science, University of Ljubljana, Slovenia.

Chapter 2

Methods Used

2.1 Domain Description

In our study, we consider adversarial problem solving, in which one must anticipate, understand and counteract the actions of an opponent. Typical domains where this type of problem solving is required include military strategy, business, and game playing. We use chess as an experimental domain. In our case, a problem is always defined as: given a chess position that is won by one of the two sides (White or Black), find the winning move. A chess problem is said to be tactical if the solution is reached mainly by calculating possible variations in the given position, rather than by long term positional judgment. In this thesis, we are not primarily interested in the process of actually solving a tactical chess problem, but in the question, how difficult it is for a human to solve the problem. A recent study has shown that even chess experts have limited abilities to assess the difficulty of a chess tactical problem [13]. We have adopted the difficulty ratings of Chess Tempo (an online chess platform available at www.chesstempo.com) as a reference. The Chess Tempo rating system for chess tactical problems is based on the Glicko Rating System [14]. Problems and users (that is humans that solve the problems) are both given ratings, and the user and problem rating are updated in a manner similar to

the updates made after two chess players have played a game against each other, as in the Elo rating system [4]. If the user solves a problem correctly, the problem's rating goes down, and the users rating goes up. And vice versa: the problems rating goes up in the case of incorrect solution. The Chess Tempo ratings of chess problems provide a basis from which we estimate the difficulty of a problem.

2.2 Meaningful Search Trees

A person is confronted with a problem when he wants something and does not know immediately what series of actions he can perform to get it [5]. In artificial intelligence, a typical general scheme for representing problems is called state space. A state space is a graph whose nodes correspond to problem situations, and a given problem is reduced to finding a path in this graph. The presence of an adversary complicates that search to a great extent. Instead of finding a linear sequence of actions through the problem space until the goal state is reached, adversarial problem solving confronts us with an expanding set of possibilities. Our opponent can make several replies to our action, we can respond to these replies, each response will face a further set of replies etc. [6]. Thus, in adversarial problem solving, the state space is usually represented by a game tree. In computer problem solving, only a part of complete game tree is generated, called a search tree, and a heuristic evaluation function is applied to terminal positions of the search tree. The heuristic evaluations of non-terminal positions are obtained by applying the minimax principle: the estimates propagate up the search tree, determining the position values in the non-leaf nodes of the tree.

Game trees are also a suitable way of representing chess tactical problems. In Fig. 2.1, a portion of a problem's game tree is displayed. Circles represent chess positions (states), and arrows represent chess moves (actions). Throughout the article, we will use the following terms: the player (i.e., the problem

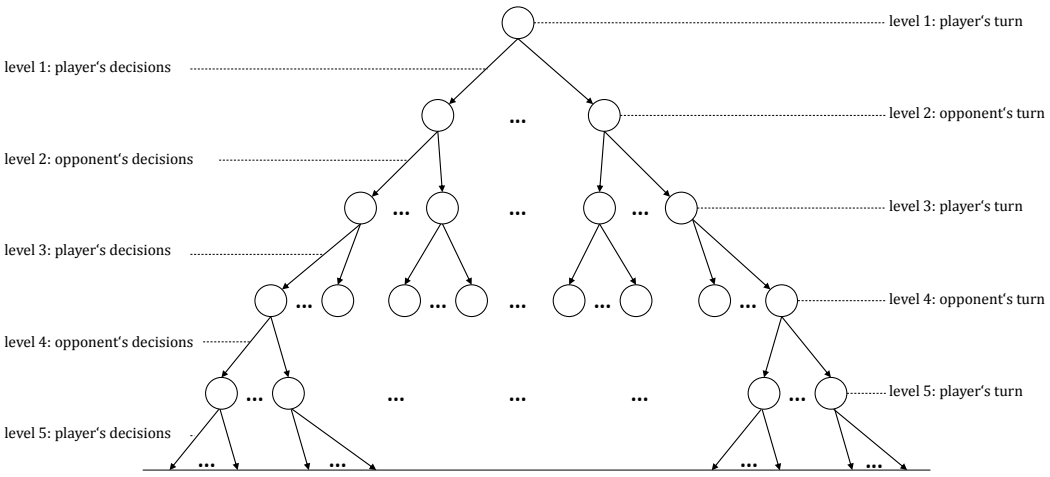

Figure 2.1: A part of a game tree, representing a problem in adversarial problem solving.

solver) makes his decisions at odd levels in the tree, while the opponent makes his decisions at even levels. The size of a game tree may vary considerably for different problems, as well as the length of particular paths from the top to the bottom of the tree. For example, a terminal state in the tree may occur as early as after the player's level-1 move, if the problem has a checkmate-in-one-move solution. In type of problems in which the difficulty arises from the combinatorial complexity of searching among alternatives, it is typically infeasible for a human to consider all possible paths that might lead to the solution of the problem. Human players therefore heuristically discard possibilities (moves) that are of no importance for finding the solution of a particular problem. In doing so, they are mainly relying on their knowledge and experience.

In fact, human problem solvers "construct" (mentally) their own search trees while solving a problem, and these search trees are essentially different than the ones obtained by computer heuristic search engines. The search trees of humans, in the sequel called "meaningful trees," are typically much

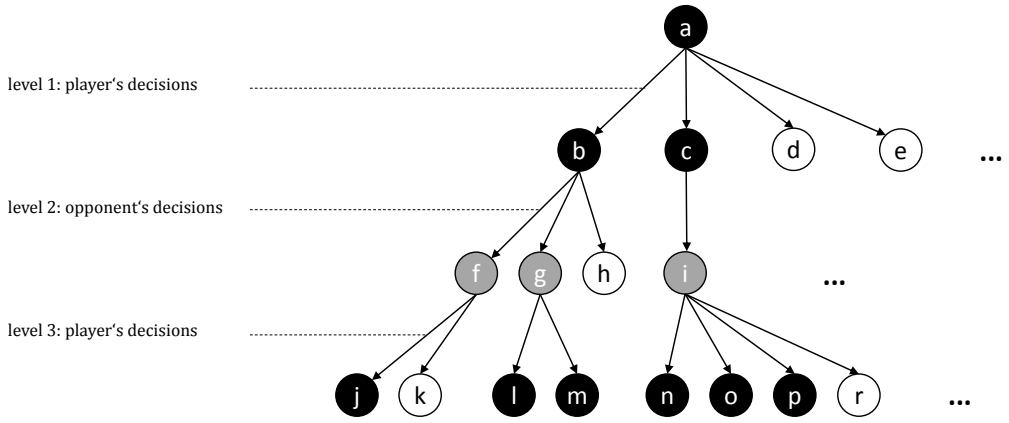

Figure 2.2: The concept of a meaningful search tree.

smaller, and, most importantly, they mainly consist of what represents meaningful (from a human problem solver's point of view) states and actions in order to solve the problem. A natural assumption is that the difficulty of a chess problem depends on the size and other properties of the chess position's meaningful tree. In order to enable automated assessment of the difficulty of a problem for a human, we therefore focused on constructing search trees that are meaningful from a human problem solver's point of view. Such trees should, above all, consist of actions that a human problem solver would consider. The basic idea goes as follows. Computer heuristic search engines can be used to estimate the values of particular nodes in the game tree of a specific problem. Only those nodes and actions that meet certain criteria are then kept in what we call a meaningful search tree. By analyzing properties of such a tree, we should be able to infer certain information about the difficulty of the problem for a human.

The concept of a meaningful search tree is demonstrated in Fig. 2.2. Black nodes represent states (positions) that are won from the perspective of the player, and grey nodes represent states that are relatively good for the oppo-

nent, as their evaluation is the same or similar to the evaluation of his best alternative. White nodes are the ones that can be discarded during the search, as they are not winning (as in the case of the nodes labeled as d, e, h, k, and r), or they are just too bad for the opponent (h). If the meaningful search tree in Fig. 2.2 represented a particular problem, the initial problem state a would be presented to the problem solver. Out of several moves (at level 1), two moves lead to the solution of the problem: a–b and a–c. However, from state c the opponent only has one answer: c–i (leading to state i), after which three out of four possible alternatives (i–n, i–o, and i–p) are winning.

The other path to the solution of the problem, through state b, is likely to be more difficult: the opponent has three possible answers, and two of them are reasonable from his point of view. Still, the existence of multiple solution paths, and very limited options for the opponent suggest that the problem (from state a!) is not difficult. Meaningful trees are subtrees of complete game trees. The extraction of a meaningful tree from a complete game tree is based on heuristic evaluations of each particular node, obtained by a heuristic-search engine searching to some arbitrary depth d. In addition to d, there are two other parameters that are chess engine specific, and are given in centipawns, i.e. the unit of measure used in chess as a measure of advantage, a centipawn being equal to 1/100 of a pawn. These two parameters are:

w The minimal heuristic value that is supposed to indicate a won position.

m The margin by which the opponent's move value V may differ from his best move value $BestV$. All the moves evaluated less than $BestV - m$ are not worth considering, so they do not appear in the meaningful tree.

It is important to note that domain-specific pattern-based information (e.g., the relative value of the pieces on the chess board, king safety etc.) is not available from the meaningful search trees. Moreover, as it is suggested in Fig. 2.2, it may also be useful to consider whether a particular level of the tree is odd or even.

2.3 Illustrative Example (Hard)

In Fig. 2.3, a fairly difficult Chess Tempo tactical problem is shown. Superficially it may seem that the low number of pieces implies that the problem should be easy (at least for most players). However, a rather high Chess Tempo rating (2015.9 points calculated from 1656 problem-solving attempts) suggests that the problem is fairly difficult.

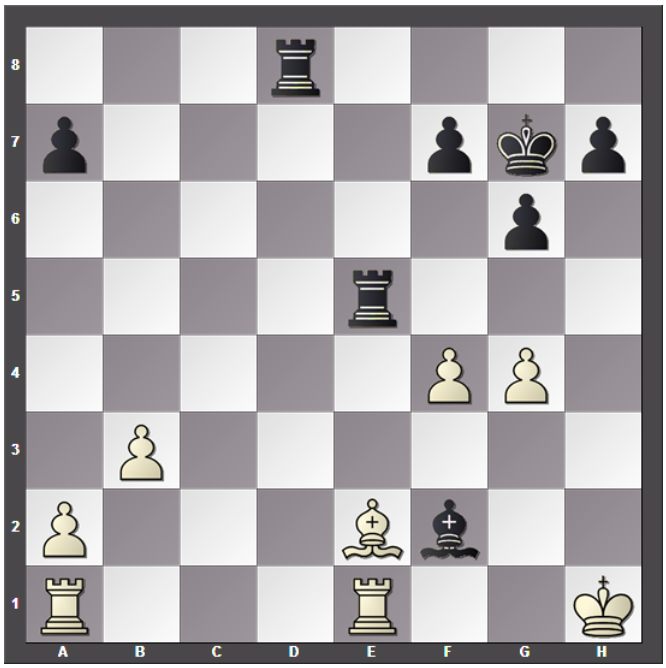

Figure 2.3: An example of a chess tactical problem: Black to move wins.

What makes this particular chess tactical problem difficult? In order to understand it, we must first get acquainted with the solution. In Fig. 2.3, Black threatens to win the Rook for the Bishop with the move 1... Bf2xe1 (that is, Black bishop captures White rook on square e1; we are using standard chess notation). And if White Rook moves from e1, the Bishop on e2 is *en prise*. However, first the Black Rook must move from e5, otherwise the White Pawn on f4 will capture it. So the question related to the problem is: what is

the best place for the attacked Black Rook? Clearly it must stay on e-file, in order to keep attacking the White Bishop. At first sight, any square on e-file seems to be equally good for this purpose. However, this exactly may be the reason why many people fail to find the right solution. In fact, only one move wins: 1... Re5-e8 (protecting the Black Rook on d8!).

It turns out that after any other Rook move, White plays 2.Re1-d1, saving the Bishop on e2, since after 2... Rd8xd1 3.Be2xd1(!) the Bishop is no longer attacked. Moreover, even after the right move 1... Re5-e8, Black must find another sole winning move after White's 2.Re1-d1: moving the Bishop from f2 to d4, attacking simultaneously the Rook on a1 and the Bishop on e2.

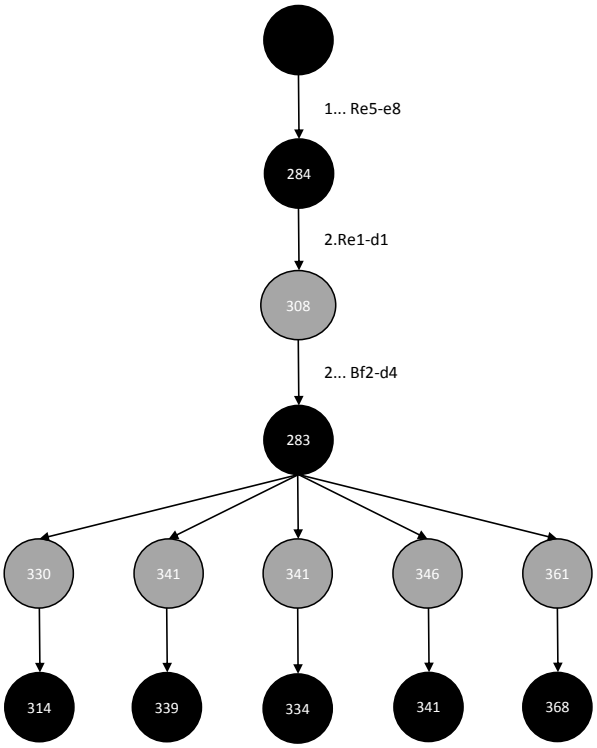

Figure 2.4: The meaningful search tree for the hard example in Fig. 2.3.

Fig. 2.4 shows the meaningful tree for the above example. Chess engine STOCKFISH (one of the best computer chess programs currently available) at 10-ply depth of search was used to obtain the evaluations of the nodes in the game tree up to level 5. The parameters w and m were set to 200 centipawns and 50 centipawns, respectively. The value in each node gives the engine's evaluation (in centipawns) of the corresponding chess position.

In the present case, the tree suggests that the player has to find a unique winning move after every single sensible response by the opponent. This implies that the problem is not easy to solve by a human.

2.4 Illustrative Example (Easy)

In Fig. 2.5, a fairly easy Chess Tempo tactical problem is shown. Superficially it may seem that the high number of pieces implies that the problem should be hard (at least for most players). However, a rather low Chess Tempo rating (996.5 points calculated from 323 problem-solving attempts) suggests that the problem is fairly easy.

What makes this particular chess tactical problem easy? Again, in order to understand it, we must first get acquainted with the solution. In Fig. 2.5, we can see that aside from *1...Rf6xf1*, which is a double attack of some sorts, because Black will than be attacking both White's king at *c1* and queen at *g5*, there aren't any other meaningful moves for Black. Why that was the right move is revealed at the next level, when the opponent has to come up with a solution. Since his king is in check, White can only do two things: (1) either move his king or, (2) capture the piece that is attacking the king. Of course, White can also try to block the attack with his rook on *d2* (*2.Rd2-d1*), but that would only result in White losing material, since Black can just capture White's queen (*2...Qe7xg5*), while simultaneously checking the opponent's king, and winning *a lot* material, since after White next move (which will be moving the king, in the best scenario), Black will be ahead, and it will be his turn to move

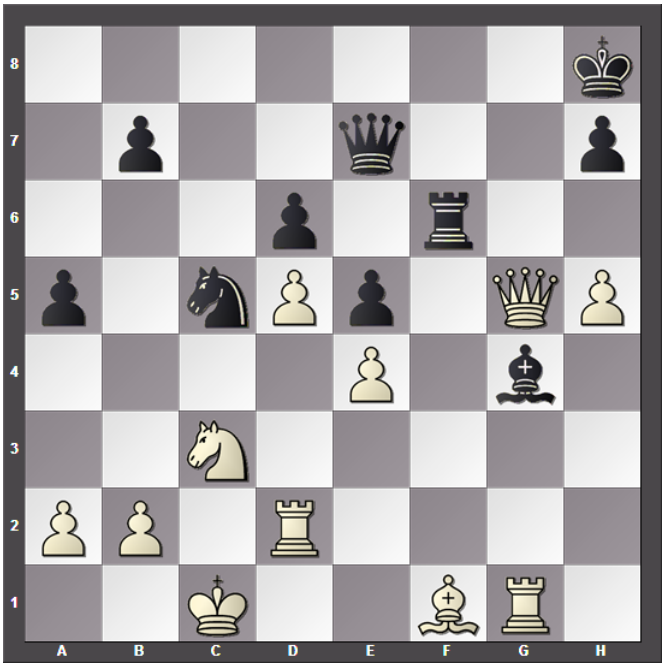

Figure 2.5: An example of a easy chess tactical problem: Black to move.

(the rook at *f1*).

Getting back to the two "meaningful" things White can do, if he chooses to move his king, he has only one valid square to go to, namely *c2*. After *2.Kc1-c2*, two of Black's pieces are being attacked, i.e. the queen on *e7* and the rook on *f1*. Luckly, Black is also attacking the pieces that are attacking his pieces, so he has the choice of capturing the rook on *g1* (*2...Rf1xg1*), or capturing the queen on *g5* (*2...Qe7xg5*). Capturing White's queen in this situation is a much better choice, since it clearly yields better material gain. At his next turn, White recapturing the lost Bishop on *f1* with his rook on *g1* (*3.Rg1xf1*) is the only viable option, kind of a "forced" move, since his own rook is been attacked by Black's rook on *f1*, so he has to do something about it. After capturing it, Black's window of opportunities got wide open, as we see in Fig. 2.6, witch shows the number of meaningful moves Black has at this point.

Now, if we consider the second "meaningful" thing White can do after Black

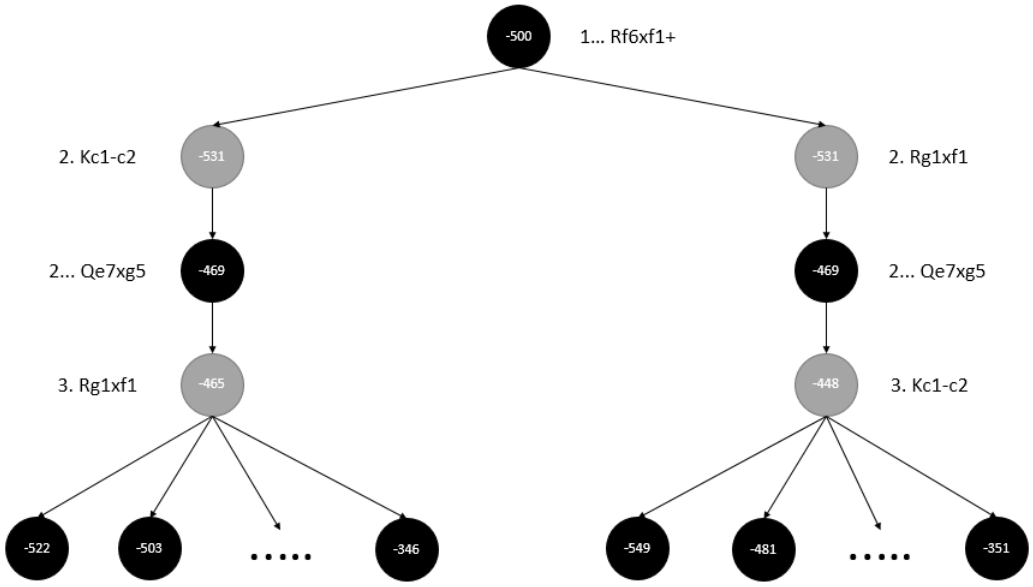

Figure 2.6: The meaningful search tree for the example in Fig. 2.5.

played *1...Rf6xf1*, is to capture the rook on *f1* (*2.Rg1xf1*). This time, it is Black who has one forced move, namely capturing the queen on *g5* (*2...Qe7xg5*). The next best thing for White here is to move his king from *c1*, so that "unpins" rook on *d2* (currently he cannot move his rook from *d2*, because that would lead to Black's queen attacking the king while being Black's turn, and that is against the rules of chess). So, after White moves his king (*3.Kc1-c2*), once more we see the situation (in Fig. 2.6) where Black is so far ahead, that he has tons of meaningful moves available to him at his next move.

This is a common phenomenon we discovered for the tactical chess positions that were deemed easy. It the example above we saw that once the player got to his third move (level 5 in our meaningful tree), he had a lot of options. That is because he made such good choices on the previous turns, that once he got to level 5, he was so far ahead, that he had a lot of meaningful moves. That is why we can see (in Fig. 2.6) the branching factor of our tree at level 5

increasing so greatly (from 1 at level 4 to 5 at level 8) if the given problem is easy. As explained before, the meaningful tree is supposed to contain moves that an experienced chess player will consider in order to find the solution of the problem. In this sense, the chess engine-computed meaningful tree approximates the actual meaningful tree of a human player.

On the other hand, we have no (pattern-based) information about the cognitive difficulty of these moves for a human problem solver. An alternative to chess engine's approximation of human's meaningful tree would be to model complete human player's pattern-based knowledge sufficient. However, that would be a formidable task that has never be en accomplished in existing research.

2.5 Attribute Description

2.5.1 A Quick Overview

As a reminder of what we explained in the previous chapters, our search trees can be up to 5 levels deep (they can be shallower, in the example of a mating position in less than 5 moves, and there are no other nodes to explore, because the game ends there). The player makes his move at odd levels (L = 1, 3 or 5), while his opponent at even levels (L = 2 or 4).

Table 2.1 shows the attributes that were used in the experiments.

#	**Attribute**	**Description**
1	Meaningful(L)	Number of moves in the meaningful search tree at level L
2	PossibleMoves(L)	Number of all legal moves at level L
3	AllPossibleMoves	Number of all legal moves at all levels
4	Branching(L)	Branching factor at each level L of the meaningful search tree
5	AverageBranching	Average branching factor for the meaningful search tree
6	NarrowSolution(L)	Number of moves that only have one meaningful answer, at level L
7	AllNarrowSolutions	Sum of NarrowSolution(L) for all levels L
8	TreeSize	Number of nodes in the meaningful search tree
9	MoveRatio(L)	Ratio between meaningful moves and all possible moves, at level L
10	SeeminglyGood	Number of non-winning first moves that only have one good answer
11	Distance(L)	Distance between start and end square for each move at level L
12	SumDistance	Sum of Distance(L) for all levels L
13	AverageDistance	Average distance of all the moves in the meaningful search tree
14	Pieces(L)	Number of different pieces that move at level L
15	AllPiecesInvolved	Number of different pieces that move in the meaningful search tree
16	PieceValueRatio	Ratio of material on the board, player versus opponent
17	WinningNoCheckmate	Number first moves that win, but do not lead to checkmate

Table 2.1: A brief description of the attritubes

2.5.2 Meaningful(L)

Description

We define a meaningful move as a move that wins material worth at least 2 pawns for the player (or, as it is defined in chess programming for better accuracy, as 200 centipawns, or CP for short), and includes the best move for the opponent, as well as all moves that are in the 0.5 pawns (50 CP) boundary of the best one. Centipawn is a score unit that conform to one hundredth of a pawn unit. According to Robert Hyatt [18], having experimented with decipawns (1/10 of a pawn), and millipawns (1/1000 of a pawn), he found out that centipawns are the most reasonable one. The argument against the decipawns is that is too coarse, so we will have rounding errors, and millipawns is too fine, and the search quickly becomes less efficient. We understand that there is no such thing as "meaningful" move, but we used the term to refer to the moves that lead to better tactical advantage. We specified the boundaries of this attribute just as a human problem solver would see the options given to him: a move is meaningful if it puts the player in a better tactical advantage than before playing the move. This attribute counts the number of meaningful moves at given level L in the search tree.

We can see what our program considers "meaningful" in Algorithm 1.

Example

We will use the list of possible moves from 2.5.3 to show witch of the moves are meaningful. If we inspect the list, we can see that only the first possible moves satisfies the boundaries we have set for a "meaningful" move. So in this case, there is only one meaningful move - $Meaningful(1) = 1$.

Algorithm 1 Get all meaningful moves

```
for each move in possibleMoves[L] do
  if type of move.score is CP then
    if move.score >= 200 then
      append move to meaningful[L]
      let noneMeaningfulFlag be False
    else if noneMeaningfulFlag == True then
      if abs(bestMoveScore – move.score) <= 50 then
        append move to meaningful[L]
      end if
    end if
  else if type of move.score is Mate then
    if move.score > 0 then
      append move to meaningful[L]
      let noneMeaningfulFlag be False
    else if noneMeaningfulFlag == True then
      append move to meaningful[L]
    end if
  end if
end for
```

2.5.3 PossibleMoves(L)

Description

Every time a chess engine searches for the answers in a given game, it considers, before using heuristics for pruning the less then winning moves, all the possible moves. Sometimes the number can be as low as one (if our king is in check, so can move only the king), but other times this number can reach as high as two hundred eighteen [19]. In our research case though, most of the positions have about 40 valid moves. We need this data more as a calculation basis than a machine learning attribute, because from this list we get our meaningful moves. That's why we keep track of the number of possible moves at each level L in PossibleMoves(L).

We can see the general approach to getting all the possible moves from the chess engine output, in our case it is from STOCKFISH, in Algorithm 2.

Algorithm 2 Get all possible moves

```
lastline ← regex for end of output
while True do
  read line from stockfishOutput
  if line == lastline then
    break
  else
    append line to possibleMoves[L]
  end if
end while
```

Example

As an example, we will take the tactical chess problem in Fig. 2.5, and we will show how we calculated all the possible moves. The following is a list of possible moves that we got from STOCKFISH (this list was abbreviated, the

actual output has more information in it, but it is not relevant in this context):

- info score cp 478 multipv 1 pv f6f1
- info score cp 5 multipv 2 pv c5b3
- info score cp -637 multipv 3 pv e7f8
- info score cp -637 multipv 4 pv e7f7
- info score cp -693 multipv 5 pv c5d3
- info score cp -884 multipv 6 pv e7g7
- info score cp -923 multipv 7 pv b7b6
- ...
- info score mate -1 multipv 35 pv g4f5
- info score mate -1 multipv 36 pv g4h5
- info score mate -1 multipv 37 pv g4d7
- info score mate -1 multipv 38 pv g4c8

We have shortened the list for readability purposes, but we can still see from the "multipv" value that there are 38 possible moves at the start of this position, i.e. at level 1, for Black. So, we can say that $PossibleMoves(1) = 38$.

2.5.4 AllPossibleMoves

Description

Similar to the attribute Tree Size, but whereas that one counts the meaningful moves, this attribute counts all the valid moves. AllPossibleMoves shows the size of the search tree that the player need to take into consideration before playing the move, at each level. In short, it shows all the possible moves in

the search tree.

We can calculate it by simply summing up all the possible moves in the search tree, for each of the levels. The pseudocode is shown in Algorithm 3.

Algorithm 3 All Possible Moves

```
for level in searchTree.depth do
    add possibleMoves[level] to allPossibleMoves
end for
```

Example

Corresponding to the pseudocode in Algorithm 3, we can easily compute this attribute if we sum up all the PossibleMoves(L), for L from 1 to the depth of the tree (we mentioned before that the depth of the tree can vary from 1 to 5, depending on the problem, with the depth at 5 being the most common). But, since we haven't yet shown an example where we would calculate all the possible moves at each of the levels, we will do so now, from our data. Accordingly, our logs show that aside from PossibleMoves(1) being 8, as shown in 2.5.3, PossibleMoves from level 2 to level 5 are: 4, 73, 45, 70. For the curious ones, that wonder how can PossibleMoves(2) can be such a low number (as 4), remember that after Black captures the bishop with his rook, White's king is in check, hence White has limited mobility. Moving forward with the calculation, $AllPossibleMoves = 38 + 4 + 73 + 45 + 70 = 230$.

2.5.5 Branching(L)

Description

Giving us the number of ratio child nodes over father nodes for each depth, this attribute shows the branching factor at each given depth in our search tree. This only captures the meaningful moves, so the default 40 moves per position doesn't apply here; considering that we search the game tree with

a MultiPV (multiple principal variation) value of 8. Usually when people examine chess problems, they would only like the winning variation, i.e. single principal variation. But in our case, we would like to get as many variations, to get all the meaningful moves, not just the best one.

We define the branching factor somewhere in the lines of Algorithm 4.

Algorithm 4 Calculate the branching factor

```
let nodes be items from searchTree
for each level in searchTree.depth do
    let fathers[level] be items from nodes[level]
    let sons[level] be items from nodes[level+1]
    branching[level] ← sons[level] / fathers[level]
end for
```

Example

In this example, we will talk about the hard illustrative example we shown previously in 2.3, and calculate the branching factor at all the levels, by hand. As we can see in Fig. 2.4, after the tactical chess problem starts (the topmost black circle) the player only has one meaningful answer *1...Re5-e8*; $Branching(1) = 1$. After the player has made his move, it's the opponents turn to look at his possibilities. The gray circle represents his meaningful answer, giving us $Branching(2) = 1$. After that, the player also has only one meaningful move which makes $Branching(3) = 1$. But, the second time that the opponent gets to play, we see more possibilities. Five meaningful answers to the players only one move quintuples the branching, i.e. $Branching(4) = 5$. In the bottom row we see five answers to five moves, which, after we divide them, we get $Branching(5) = 1$.

2.5.6 AverageBranching

Description

Since the branching factor at each given depth is not uniform, we also include the average branching factor in our search tree. Defined as the sum the branching factors from all depths, divided by the depth of the search tree, it gives us an idea of the growing size of our search tree. In chess the average branching factor in the middle game is considered to be about 35 moves [7], but since we are only counting meaningful moves, our average branching factor is considerably smaller.

The average branching factor is calculated by following Algorithm 5.

Algorithm 5 Average branching in our search tree

```
for level in searchTree.depth do
    add branching[level] branchingSum
end for
averageBranching ← branchingSum / searchTree.depth
```

Example

The average branching is nothing much, but the sum of the attribute Branching(L), for L ranging from 1 to the depth of the meaningful tree, divided by the depth of that same tree. Since we calculated that attribute for the hard illustrative example in 2.3, we will reuse those values. $AverageBranching = (1 + 1 + 1 + 5 + 1)/5 = 1.8$

2.5.7 NarrowSolutions(L)

Description

One of the principles in chess is the concept of a "forcing move". A forcing move is one that limits the ways in which the opponent can reply. A capture of

a piece that it is best to be recaptured, a threat of mating (or forking, etc.) or a check (where the rules force the player to respond in a certain way) all count as forcing moves. This type of move shows up in our search tree as a parent node with only one child (in chess term, as a move with only one meaningful answer). The narrow solutions attribute counts the number of this type of moves at each level.

We calculate the number of narrow solutions as shown in Algorithm 6.

Algorithm 6 Calculate the narrow solutions

```
for each level in searchTree.depth do
  let meaningfulAnswers be items from stockfishOutput[level]
  if meaningfulAnswers.length == 1 then
    increment narrowSolutions[level]
  end if
end for
```

Example

To explain what a narrow solution is, we will use, once again, the hard illustrative example. As seen in Fig. 2.4, after the player makes his move, the opponent only has one meaningful answer; $NarrowSolutions(2) = 1$. The same can be said about the player's options the second he needs to make a move: $NarrowSolutions(3) = 1$. But after that move, the opponent has a lot of meaningful answers for that one answer, hence $NarrowSolutions(4) = 0$, because there are no "forced moves" at this level. On the fifth level of the meaningful tree, however, we see five moves that can be answered only by one meaningful move. That's why we have $NarrowSolutions(5) = 5$.

2.5.8 AllNarrowSolutions

Description

With this attribute, we would like to see how much narrow solutions (forced moves) appear in our search tree. It is more likely that most of the narrow solutions will appear at levels 3 and 5, meaning the opponent is limiting the options of the player, but we would still like to see the magnitude of narrow solutions from the whole meaningful tree.

This attribute is also fairly simple to calculate, since it's just summing up the NarrowSolutions(L) attribute from each of the levels, just like we describe it in Algorithm 7.

Algorithm 7 All Narrow Solutions

```
for level in searchTree.depth do
    add narrowSolutions[level] to allNarrowSolutions
end for
```

Example

Because this attribute is similar to the other ones that are other attribute summed up, we already seen this formula (Algorithm 7) for calculating this kind of attribute. If we would to take the meaningful tree gotten from the hard illustrative example 2.4 and the data from the example from the NarrowSolutions attribute 2.5.7, we would get $AllNarrowSolutions = 1 + 1 + 1 + 0 + 5 = 8$. There is no value for NarrowSolutions(1) in the example, but from the meaningful tree in the hard illustrative example we can see that $NarrowSolutions(1) = 1$.

2.5.9 TreeSize

Description

Our search tree, which is similar to a game tree in chess (a directed graph with nodes and edges) consists of positions (nodes) and moves (edges). It differs in a way that the root is not the default starting position in chess, but a start of a tactical problem (whole games in chess are considered a strategic problem, while choosing a winning variation in a pre-given position setup is considered a tactical problem). Also, as opposed to a game tree, our search tree has a fixed maximum depth (set to 5, for computational purposes) which means, that not all leafs (end nodes) in the tree are usual chess game endings, such as a checkmate or a draw (we don't incorporate time control in our search, just a fixed depth, and there is no option to draw, since the computer plays against itself). Important thing to note here is that at any one given level (or depth) in the search tree, only one side (Black or White) can move.

Simply put, TreeSize is the size (measured in number of meaningful moves from each level) of our search tree, as shown in Algorithm 8.

Algorithm 8 Tree size

```
for level in searchTree.depth do
    add meaningful[level] to treeSize
end for
```

Example

The TreeSize, as see in the algorithm 8 can be computed from the attribute Meaningful(L) from $L = 1$ to the depth of our meaningful tree. If we take the tree from Fig. 2.6, Meaningful(L) for each L = 1, 2, 3 is 1. Meaningful(4) and Meaningful(5) are both 5. Consequently, we have $TreeSize = 1+1+1+5+5 = 13$.

2.5.10 MoveRatio(L)

Description

For any given position, there are a number of valid moves, also referred to as possible moves, as seen in the attribute PossibleMoves(L), but only a few sensible ones, also referred to as meaningful moves, as seen in the attribute Meaningful(L). If we would like to calculate the proportion of meaningful moves out of all possible moves, which would give us somewhat of an idea of difficulty, since different values tell different stories. A really low value would mean that, either there are a lot of possible moves or there are very little meaningful moves. A high value might suggest that almost all the moves are meaningful, or it may even mean that the opponent is forcing us moves, so we quickly run out of possibilities. This attribute shows the ratio between those two, out of all the possible moves how many of them should the player consider playing.

To calculate this value there isn't much of complexity involved, we just divide the number of meaningful moves with the number of possible moves at each level, as documented in Algorithm 9.

Algorithm 9 Proportion of meaningful moves out of all possible ones

```
for level in searchTree.depth do
    moveRatio[level] ← meaningful[level] / possibleMoves[level]
end for
```

Example

This computation will be quite simple, since we already shown an example calculating the meaningful moves in 2.5.2 for the easy illustrative example in Fig. 2.5, and the number of possible moves in 2.5.3. Thus, we have $MoveRatio(1) = 1/38 = 0.026$.

2.5.11 SeeminglyGood

Description

Some (even many) positions have a high rating mainly because there is seemingly attractive variation in them that doesn't actually lead to victory. Since most of the difficult problems have only one winning variation, all the alternatives are either seen as losing lines, and ignored immediately by the player, or in some cases, when the opponent has only one good answer that the player overlooks, are also seen as winning variations (by the player). Our reasoning is, it's easier for the player to get carried away by this alternatives, if a lot of them exist. We call these deceptive alternatives seeming good moves, since they are not really a good move for the player (in most cases they worsen the tactical advantage of the player).

To get all the seemingly good moves the player would encounter, we need to search the non-meaningful alternatives that have only good one answer by the opponent, just like in Algorithm 10.

Example

As previously mentioned, SeeminglyGood is one attribute that cannot be extracted from the meaningful search tree. That is because of the nature of the attribute, which is counting the non-meaningful moves which only have one meaningful answers from the opponent to get an idea of answers that we could have missed while searching for the right move. In Fig. 2.7 we can see such tactical position, where White can make a move that will cost him his rook. The winning (meaningful) move here would be to move the queen to *b5* (*1.Qb3-b5*), which attacks Black's rook at *e8* that is undefended. After that move, Black can safely move his rook to *d8* (*1...Re8-d8*), at the same time White can capture Black's bishop on *b7* (2.Be4xb7). We explain this reason behind moving the queen before capturing the bishop in more detail next.

If White overlooks the crushing answer from Black, although he gets to cap-

Algorithm 10 Extract the seemingly good moves from all the possible ones

```
if level == 1 then
  for move in possibleMoves[level] do
    if type of move.score is CP then
      if move.score < 200 then
        append move to badMoves
      end if
    end if
  end for
  for move in badMoves do
    answers ← stockfishOutput(move)
    if answers[1].score < 200 then
      let onlyOneAnswer be True
    end if
    if answers[2].score < 200 then
      let onlyOneAnswer be False
    end if
    if onlyOneAnswer == True then
      increment seeminglyGood
    end if
  end for
end if
```

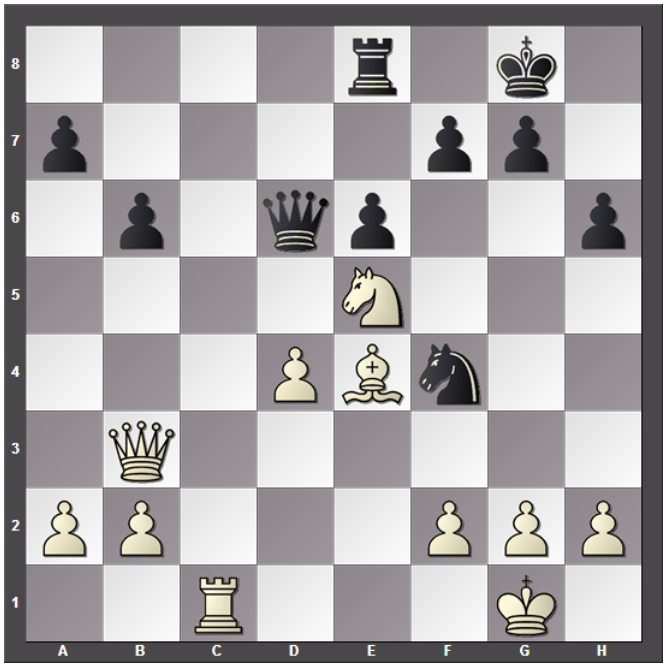

Figure 2.7: An example of a tactical chess problem where there is a seemingly good move that leads to tactical disadvantage: White to move.

ture Black's bishop at *b7*, he loses his rook at *c1*. That is because after White plays the "non-meaningful" (seemingly good) move *1.e4xb7*, Black can respond by "forking" the rook at *c1* by moving his knight to *e2* (1...Nf4-e2+) checking White's king while also attacking White's rook. White has no other option, but to move his king and lose the rook in Black's next move. White could have avoided this move if *1.Qb3-b5* was played first (before moving the bishop), because the queen would be defending the square *e2*, hence Black would have never been able to fork White's rook. From this we get: $SeeminglyGood = 1$, since in this position there is only one such move.

2.5.12 Distance(L)

Description

This attribute gives us a representation of how far the player (or his opponent, depending on the level) has to move his chess pieces. The sum of all distances between the start square and the end square for each of the meaningful moves at a given depth in our search tree. Calculated according to the rules of Chebyshev distance, the larger value of the two absolute differences between the start file and end file, and the start rank and the end rank.

We can see the Chebyshev distance being calculated, right before statement that adds the variable distance to the the dictionary for storing the distances at each level, in Algorithm 11.

Algorithm 11 Distance between the square on which the piece is, and where it would be, after the player makes a move

```
for level in searchTree.depth do
    for move in meaningful[level] do
        startSquare ← move.startSquare
        endSquare ← move.endSquare
        startFile ← startSquare.file
        endFile ← endSquare.file
        startRank ← startSquare.rank
        endRank ← endSquare.rank
        distance ← MAX (ABS (startFile - endFile), ABS (startRank - endRank))
        add distance to distance[level]
    end for
end for
```

Example

We will use the list of meaningful moves, for our easy illustrative example, shown in 2.5.3, more specifically, just one item from it (the first one, which is the only one that contains a meaningful move). The combination of letters and numbers shown here, *f6f1*, means that we will move the piece on *f6* (Black's rook) to *f1*, capturing White's bishop. By the definition of measuring distance, we need both ranks and files where the pieces rests, and where needs to move. From *f6f1* we extract the information: the start rank is *6*, while the end rank is *1*, a difference of *5*. Likewise, we can extract the start file, which is *f*, and in this case is the same as the end file, *f*, so the numeric difference between the two files is 0.

From the formula in 11: $MAX(ABS(f-f), ABS(6-1))$, where we can substitute the file (in our case *f*) with a number $a = 1, b = 2, ...h = 6$, $f - f$ would be $5-5$ which gives us zero, and since we are looking for a maximum, we should look at the other result, absolute value of $(6-1)$, which is 5. So we can conclude the calculation of this attribute (at level 1) with $Distance(1) = 5$. Important thing to note here is that we only had one meaningful move, so our attribute Distance(1) included a distance from only one move. If we had more meaningful moves at level 1, Distance(1) would be the sum of all their calculated distances.

2.5.13 SumDistance

Description

This is a measure of how far the involved players (the player and the opponent) has to move the chess pieces, if they would play all the meaningful variations. Once we calculate the summed distance at each depth, we move on to all distances from all the meaningful moves from every depth.

SumDistance, like the other attributes that sum up the minor attributes from each of the levels, can be simply calculated by adding Distance(L) from

each of the levels, in Algorithm 12.

Algorithm 12 Summed distance from all the levels

```
for level in searchTree.depth do
    add distance[level] to sumDistance
end for
```

Example

This attribute takes into account all Distance(L) attributes from $L = 1$ to the depth of the meaningful tree. We will use the meaningful tree in the hard illustrative example in 2.3. It has 13 nodes (meaningful moves), which means we need to calculate 13 distance to compute the $SumDistance$ attribute for this three. From the Fig. 2.4 we can see that the first three moves are (in STOCKFISH output format, for easier calculation): *e5e8*, *e1d1* and *f2d4*. The first one involves a piece that is moving on the same file, so only the ranks matter: $ABS(5-8) = 3$; the second move involves a piece moving on the same rank, so we only take the start and end file into consideration: $ABS(e-d) = ABS(5-4) = 1$. The third one features different rank as well as different file: $MAX(ABS(f-d), ABS(2-4)) = MAX(ABS(6-4), 2) = MAX(2,2) = 2$. So far we got $SumDistance = 3+1+2 = 6$. But, we still got 10 more distances from moves to compute.

At level 4, we see 5 meaningful moves, and from our gathered data logs, we can see : *e2c4*, *e2f3*, *e2b5*, *e2a6*, *d1d4*. To speed up the proces, we will skip a couple of steps while computing the distances. That said, they are as follows: computeDistance(e2c4) = 2, computeDistance(e2f3) = 1, computeDistance(e2b5) = 3, computeDistance(e2a6) = 4, computeDistance(d1d4) = 3. That makes $SumDistance = 6 + (2 + 1 + 3 + 4 + 3) = 19$. At level 5, again we see 5 meaningful moves, but this time the calculation step is a little easier to do by hand, since 4 of the 5 moves are the same, i.e. for any one of the 4 moves that the opponent makes, the player has the same answer. Again,

from our logs, one of the moves is *d4a1* with a distance of 3, and *d8d4* with a distance of 4.

Now that we know all the distances in our meaningful tree, we can work out the SumDistance attribute to be $SumDistance = 19 + (3 + 4) = 26$.

2.5.14 AverageDistance

Description

Just like with AverageBranching, we would like to get an overview of the minor attribute, from witch the distance is calculated, Distance(L), and that is why the AverageDistance shows how much the players would have to move their pieces on average. AverageDistance is the arithmetic mean of the distance, and since we have the sum of the distances already calculated in SumDistance, we just divide it with the depth of our search tree. Again, it will not always be 5, the fixed depth we predefined, because sometimes the tree can be smaller (but not larger). We can see this formula in Algorithm 13.

Algorithm 13 Average distance the players would need to move the chess pieces on the board

averageDistance ← *sumDistance* / *searchTree.depth*

Example

We can expect an example for this attribute to be as brief as the algorithm by which the calculation abides, and would be right. Looking back to the hard illustrative example in 2.3, we can see that our meaningful tree is 5 levels deep, and looking at the computed SumDistance attribute to 26 in 2.5.13, we get $AverageDistance = 26/5 = 5.2$.

2.5.15 Pieces(L)

Description

The number of meaningful moves only shows us a number of valid moves that we can take, that are sensible. But those moves can have something in common, the piece that is moving. That's why we are introducing this attribute – Pieces(L): Number of different pieces involved in the meaningful moves at each level.

We check for every move if the involved piece has not yet appeared between the other meaningful answers to the opponent's (or player's) move in Algorithm 14.

Algorithm 14 Number of different pieces involved

```
count ← 0
let pieces be empty
for move in meaningfulAnswers do
  if pieces[move.piece] != True then
    increment count
    let pieces[move.piece] be True
  end if
end for
```

Example

This attribute can sometimes be the same with $Meaningful(L)$, if we only have one meaningful move at a given level, that means that we will move only one piece. But when there are more than one meaningful answers at any given level, like at level 1 in the example in Fig. 2.8, Pieces(1) can differ from Meaningful(1), if there is a specific piece occurring in more than one move. The list in 2.5.18 shows all the possible moves, from which only the first three are meaningful, but we can see the queen at $g7$ appearing in two of the three

meaningful moves. That means, although we have three meaningful moves, we only have two different pieces present at level 1; $Pieces(1) = 2$.

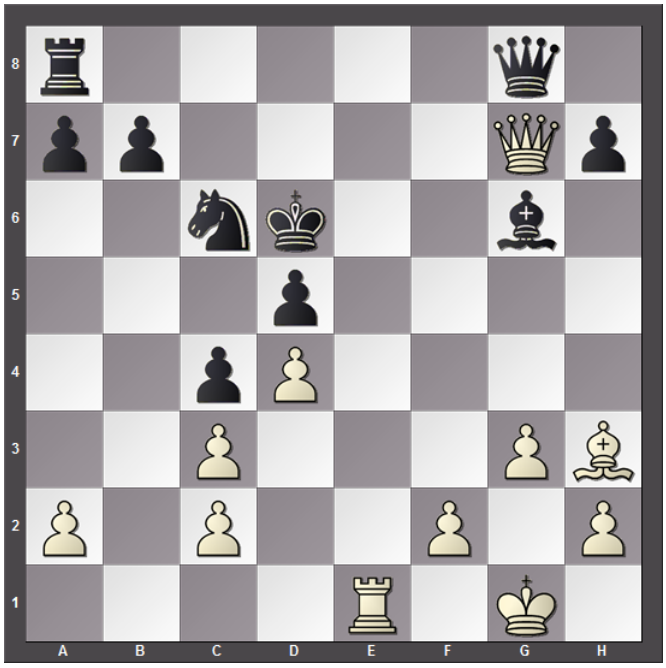

Figure 2.8: An example of a tactical chess problem where there are meaningful moves that are not checkmate: White to move.

2.5.16 AllPiecesInvolved

Description

This attributes shows how much pieces has been moved, while we were building the search tree, or rather, the number of all different pieces involved in the search tree. Important thing to point out here is: the "different" restriction applies only on the same level. Once we go up a level (or down, depending on the representation of the tree), the set that keeps track of the unique pieces resets.

We check for every move if the involved piece has not yet appeared between the other meaningful answers to the opponent's (or player's) move in Algorithm 15.

Algorithm 15 Number of all pieces involved

```
for level in searchTree.depth do
    add pieces[level] to allPiecesInvolved
end for
```

Example

For the purpose of showing how we calculated the number of all involved pieces in the meaningful search tree, we will use the tactical chess problem we saw in the hard illustrative example in 2.3. The problem had 13 meaningful moves overall, so we know that the number of all pieces involved can be 13 or less. Best way to calculate it is to go from level 1 to the depth of the meaningful tree (which is not always 5, but it is 5 in our case), and count the different pieces that occur. From Fig. 2.4 we can see that there is only one meaningful move at the first three levels. So Pieces(L) for $L = [1, 2, 3]$ is 1. At level 4, there are 5 meaningful moves, but only 2 pieces moving, i.e. White's bishop from *d1* is involved in four of the meaningful moves, and the white rook (now at *d1*, after *2.Re1-d1*) is involved involved in the last one. That makes $Pieces(4) = 2$.

At level 5, we observe a similar situation, where out of 5 meaningful moves, there is only 2 pieces moving: Black's bishop (at this point placed at *d4*) accounts for four of the meaningful moves, and Black's rook at *d8* is responsible for the fifth meaningful move. If we sum up the pieces from all the levels, we get $AllInvolvedPieces = 1 + 1 + 1 + 2 + 2 = 7$.

2.5.17 PieceValueRatio

Description

In chess, the relative piece value system assigns a value to each piece when assessing its strength in potential exchanges. Here we use the system to determine which player has larger relative piece value on the board, by taking the ratio between the player's calculated value and the value of the opponent's pieces. Common values for the different chess pieces involved in the game are, also proposed by Claude Shannon, one point for each of the pawns, three points for each of the minor pieces (knights and bishops), five points for each rook, and nine points for the queen [15, 8, 20, 16]. To avoid king captures, he is often assigned a large value, such as 10000, but we don't include this value in our equations, to avoid undermining the values of the rest of the pieces on the board, giving us a more realistic ratio.

At the start of each game, we divide the player's relative piece value with the opponent's piece value, as shown in Algorithm 16.

Algorithm 16 Relative piece value ratio

```
for piece in board.playerPieces do
    add piece.value to playerPieceValue
end for
for piece in board.opponentPieces do
    add piece.value to opponentPieceValue
end for
pieceValueRatio ← playerPieceValue / opponentPieceValue
```

Example

Calculating the relative piece value ration is pretty straight forward, once you are familiar with the rules. Like we described the attribute, we don't consider the king's value, whatever it is, because there will always be a white

and black king on the chessboard. The values of the other pieces are relative to the pawn (value of one). The knight and the bishop are each worth three pawns, the rook is worth five pawns, while the queen is worth nine. The calculation that follows is from looking at the tactical chess problem in Fig. 2.8. First, we go over the player's pieces, adding the value of the piece to the total value for the player: There are seven white pawns (add 7 to total value for White), a white bishop (add 3), a white rook (add 5) and a queen (add 9). That make the total for the player equal to 24. Second, we go over the opponent's pieces, adding the value of the piece to the total value for the opponent: There are five black pawns (add 5 to total value for Black), a black knight (add 3), a black bishop (add 3), a black rook (add 5) and a black queen (add 9). That make the total for the player equal to 25.

2.5.18 WinningNoCheckmate

Description

Number of moves that would lead to a winning position, but they are not mate. There is a difference between the easy and the hard positions in the tactical problems we are looking at, namely the easy problems can have a lot of mating possibilities, while the hard ones are more about winning material or getting in a better tactical position.

At the start of each game, check to see how many moves have the opportunity to gain material, but not checkmate the king, as shown in Algorithm 17.

Algorithm 17 Winning moves, that aren't checkmate

```
for move in possibleMoves[1] do
  if type of move.score == CP AND value of move.score >= 200 then
    add move to winningNoCheckmate
  end if
end for
```

Example

This time around, we need an example where there will be more than one meaningful move at level 1, at least one of them will be measured in mate in x moves, and the others will be measured in centipawns. The example below is a shortened list of all the possible moves White can make at the start of the problem. A chessboard illustration of the problem given can be seen in Fig. 2.8.

- info score mate 1 multipv 1 pv g7d7
- info score cp 370 multipv 2 pv e1e6
- info score cp 370 multipv 3 pv g7b7
- info score cp 116 multipv 4 pv g7f6
- info score cp -61 multipv 5 pv g7h6
- info score cp -184 multipv 6 pv g7g8
- info score cp -241 multipv 7 pv e1e7
- info score cp -861 multipv 8 pv g7g6
- info score cp -901 multipv 9 pv g7e5
- info score cp -924 multipv 10 pv e1e8
- ...

We can see that moving the queen to *d7* results in a move in 1 move. But if the player misses that move, there are other possibilities that are still considered as meaningful moves, e.g. taking the pawn on *b7* with the queen or checking the king with the rook (*1.Re1-e6+*). In this case, we say that the attribute $WinningNoCheckmate = 2$, since there are three meaningful moves, but one of them leads to imminent mate.

2.6 Experimental Design

The aim of the experiments was to assess the utility of the meaningful search trees for differentiating between easy, medium hard and hard tactical chess problems. We used 900 problems from the Chess Tempo website: 1/3 of them were labeled as "easy" (with average Chess Tempo Standard Rating $\varnothing$ = 1254.6, standard deviation $\sigma = 96.4$), 1/3 of them as "medium hard" ($\varnothing$ = 1669.3, $\sigma = 79.6$), and 1/3 of them as "hard" ($\varnothing = 2088.8$, $\sigma = 74.3$). Chess engine Stockfish at 10-ply depth of search was used to build the meaningful search tree up to level 5. The parameters w and m were set to 200 centipawns and 50 centipawns, respectively. Several attributes were derived from the trees. They are presented in Table 2.1. Attributes 1–10 represent features that mainly reflect properties of the meaningful search trees, while attributes 11–17 are more closely related to the properties of the chosen domain – chess in our case. We used five standard machine learning classifiers (and 10-fold cross validation) to estimate performance of (all) the attributes, and the performance of each aforementioned groups of attributes.

Chapter 3

Experimental Results

The methods we used we explained in the previous section, but we need to compare them with the results on our data set of tactical chess games played by humans. We present the experiment in this chapter.

Having experimented with different parameters for the lists of algorithms (see 3.1) that we used (leftmost column), we found the ones that work for our methods of work:

1. **Neural Networks** - Inspired by the central nervous system of animals, artificial neural networks are a statistical learning algorithm that are used to estimate functions.

   (a) Hidden layer neurons: 20

   (b) Regularization factor: 1.0

   (c) Maximum iterations: 300

   (d) The data was normalized

2. **Logistic Regression** - A type of probabilistic statistical classification model, mostly used for binary prediction, but here we used for three values.

   (a) L2 (squared weight) for regularization, with training error cost (C) 1.0

   (b) We normalized the data

3. **Classification Tree** - A decision tree where the target variable (in our case difficulty) can take a finite set of values.

   (a) Gain ratio was used as a criterion attribute selection

   (b) Exhaustive search for optimal split as binarization

   (c) Minimum 10 instances in the leaves for pre-pruning

   (d) Stopped splitting nodes when the majority class was 85% or higher

   (e) m-estimate for post-pruning was set to 10

   (f) Recursively merged the leaves with the same majority class

4. **Random Forest** - An ensemble learning method for classifications that operates by constructing multitude of decision trees.

   (a) 10 trees in the forest

   (b) Seed for random generator was 7

   (c) Stopped splitting nodes with 10 or fewer instances

5. **CN2 Rules** - a rule induction learning algorithm that can create set of rules, while being able to handle noisy (imperfect) data.

   (a) Laplace was used as a rule for quality estimation

   (b) The default rule alpha for pre-pruning was set to 0.05

   (c) The parent rule stopping alpha was set to 0.2

   (d) There were no restrictions on minimum coverage

   (e) The beam width was set to 5

   (f) We used the exclusive covering algorithm

To measure the correctness of our approach we used classification accuracy (higher is better), area under ROC curve (higher is better) and Brier score (less is better), gathered from the experiments for all the attributes (see Section 2.5.1) is shown in Table 3.1.

	All Attributes		
Classifier	**CA**	**AUC**	**Brier**
Neural Network	**0.81**	**0.94**	**0.25**
Logistic regression	**0.81**	**0.94**	0.28
Classification Tree	**0.81**	0.90	0.33
Random Forest	0.74	0.89	0.41
CN2 rules	0.73	0.89	0.37

Table 3.1: Results of experiments with the all attributes.

Classification accuracy (CA), Area under ROC curve (AUC), and Brier score are given for each of the five classifiers. All classifiers were able to differentiate between easy, medium hard, and hard problems with a high level of accuracy when both the tree and domain attributes (from Table 2.1) were used. 17 attributes total, each of them derived from letting a chess engine play chess matches against himself, while logging data that we later used for machine learning.

However, it is interesting to observe that their performance remained almost the same when only attributes 1–10 (see Table 3.2) were used. This only includes attributes like the number of nodes in our tree (the tree size), the number of all considered nodes before pruning, the branching factor, the number of parent nodes with only one child node, etc. This is the important because the whole premise of our research was to find automated assessment of difficulty, without going too much in depth in a given domain. These attributes are computable from the structure of meaningful search trees, and contain no domain-specific knowledge.

On the other hand, the performance of the classifiers dropped significantly

	Attributes 1-10		
Classifier	**CA**	**AUC**	**Brier**
Neural Network	**0.80**	**0.94**	**0.26**
Logistic regression	0.79	**0.94**	0.29
Classification Tree	**0.80**	0.89	0.33
Random Forest	0.74	0.89	0.41
CN2 rules	0.74	0.90	0.36

Table 3.2: Results of experiments with the tree attributes.

when attributes 11–17 (see Table 3.3) were used. These attributes were still derived from the meaningful search trees, however, they do contain chess-related knowledge such as information about piece movements, piece values, and checkmates, but do not concern the structure of the trees at all. For example, the more distant moves, that the player might overlook, the sheer number of pieces and their roles on the board that the player needs to consider before making a move, the value ratio of pieces on the board between the player and the opponent, etc.

	Attributes 11-17		
Classifier	**CA**	**AUC**	**Brier**
Neural Network	0.46	0.64	0.61
Logistic regression	0.47	**0.69**	**0.60**
Classification Tree	0.46	0.63	0.73
Random Forest	**0.49**	**0.69**	**0.60**
CN2 rules	0.41	0.61	0.86

Table 3.3: Results of experiments with the domain attributes.

The results support the idea on not dwelling too deep into a specific domain to extract information of problem difficulty, but to look at the built search tree generated from domain knowledge, while still being generalized enough that can be used to estimate difficulty in a wide selection of problem areas.

Chapter 4

Conclusions

We tackled the problem of how to assess automatically the difficulty of a mental problem for a human, which depends on the human's expertise in the problem domain. We focused in our experiments on assessing the difficulty of tactical chess problems for chess players of various chess strengths. The difficulty depends on the amount of search required to be carried out by the player before he or she finds the solution. An experienced player only has to search a small part, here called "meaningful tree," of the complete search tree. The rest of the search tree is pruned away by a large amount of pattern-based chess knowledge that the player has accumulated through experience. A direct approach to assessing the difficulty of the problem for the player would be to automatically reconstruct the player's meaningful tree. This would however require a computer implementation of the player's pattern knowledge which would be extremely hard due to the complexity of this knowledge and has never been done to a realistic degree. The idea put forward in this thesis is to approximate the player's meaningful tree with the help of another type of "meaningful" tree.

We showed how such an approximation to the human's meaningful tree can be extracted from the complete search tree automatically by a chess playing program. The extracted subtree only contains critical chess variations, that

is those that only contain best or "reasonable" moves which is decided by the chess program's evaluations of positions. To turn this idea to work as difficulty estimator of chess problems, we constructed a set of attributes and performed classification learning for the task of classifying positions into hard or easy. The attributes were of two types: (1) those that are derived from the structure of the meaningful tree only (just formal mathematical properties of the tree, ignoring the chess-specific contents of nodes and moves in the tree), and (2) attributes that reflect chess-specific contents of the nodes and arcs. The important findings from the experimental results are:

1. The classification accuracy so obtained was rather high, about 80%, which indicates the viability of the proposed approach.

2. Using tree structure attributes only, achieved practically the same accuracy as using all the attributes.

3. Using chess-specific attributes only, produced accuracy inferior to using all the attributes, as well as using tree structure attributes only. This is interesting because it indicates that the difficulty can be determined from the structure of the meaningful tree, but less so from domain-specific contents.

We set the aim of the research to be finding formalized measures of difficulty that could be used in automated assessment of the difficulty of a mental task for a human. As said before, this would be really convenient for subjects such as ITS or student's exams. We mention intelligent tutoring systems because of the complexity and work involved in the process of making them. If we would have a formalized approach that takes into account only a simple game tree and its properties, we would be simplifying the process of making an ITS.

Bibliography

[1] P. Jarušek, R. Pelánek, "Difficulty rating of sokoban puzzle," in: Proc. of the Fifth Starting AI Researchers' Symposium (STAIRS 2010), IOS Press, 2010, pp. 140–150.

[2] R. Pelánek, "Difficulty rating of sudoku puzzles by a computational model," in: Proc. of Florida Artificial Intelligence Research Society Conference (FLAIRS 2011), AAAI Press, 2011, pp. 434–439.

[3] Guid M., Bratko I., "Search-based estimation of problem difficulty for humans," in Artificial Intelligence in Education, ser. Lecture Notes in Computer Science, H. Lane, K. Yacef, J. Mostow, P. Pavlik, Eds. Springer, vol. 7926, 2013, pp. 860–863.

[4] A.E. Elo, "The rating of chessplayers, past and present," in Arco Pub., 1978.

[5] A. Newell, H. Simon, "Human Problem Solving," Prentice-Hall, 1972.

[6] D.H. Holding, "Adversary problem solving by humans," in Human and machine problem solving. Springer, 1989, pp. 83–122.

[7] V. Allis, "Searching for Solutions in Games and Artificial Intelligence," in Ph.D. Thesis, University of Limburg, pdf, 6.3.9 Chess, 1994, pp. 171.

[8] H. S. M. Coxeter, "Mathematical Recreations and Essays," from the original by W. W. Rouse Ball, Macmillan, 1940.

[9] J. Demšar, T. Curk, A. Erjavec, Č. Gorup, T. Hočevar, M. Milutinović, M. Možina, M. Polajnar, M. Toplak, A. Starič, M, Štajdohar, L. Umek L. Žagar, J. Žbontar, M, Žitnik, B. Zupan, "Orange: Data Mining Toolbox in Python," in Journal of Machine Learning Research, vol. XIV, pp. 2349-2353, 2013.

[10] K. Kotovsky, H. A. Simon, "What makes some problems really hard: Explorations in the problem space of difficulty," in Cognitive Psychology 22(2), pp. 143–183, 1990.

[11] Z. Pizlo, Z. Li, "Solving combinatorial problems: The 15-puzzle," in Memory and Cognition 33(6), pp. 1068–1084, 2005.

[12] M. Dry, M. Lee, D. Vickers, P. Hughes, "Human performance on visually presented traveling salesperson problems with varying numbers of nodes," in Journal of Problem Solving 1(1), pp. 20–32, 2006.

[13] D. Hristova, M. Guid, and I. Bratko, "Assessing the Difficulty of Chess Tactical Problems," in International Journal on Advances in Intelligent Systems, vol. VII, no. 3 & 4, 2014.

[14] M.E. Glickman, "Parameter estimation in large dynamic paired comparison experiments," in Applied Statistics 48, pp. 377-394, 1999.

[15] J. Good, "A Five-Year Plan for Automatic Chess - Appendix F," The Value of the Pieces and Squares. Machine Intelligence Vol. 2, 1968.

[16] H. M. Taylor, "On the Relative Values of the Pieces," in Chess. Philosophical Magazine, Series 5, Vol. 1, pp. 221-229, 1876.

[17] K. Kotovsky, J. R. Hayes, H. A. Simon, "Why are some problems hard? Evidence from tower of Hanoi," in Cognitive Psychology 17(2), pp. 248–294, 1985

[18] (September 09, 2009) Centipawns and Millipawns by Robert Hyatt, CCC. Available at: http://www.talkchess.com/forum/viewtopic.php?t=29694&start=14

[19] (June 11, 2011) Max[imum] amount of moves from a position? by Árpád Rusz, CCC. Available at: http://www.talkchess.com/forum/viewtopic.php?topic_view=threads&p=410026&t=39332

[20] Influence Quantity of Pieces - Fibonacci Spiral. Available at: https://chessprogramming.wikispaces.com/Influence+Quantity+of+Pieces#FibonacciSpiral