University of Ljubljana
Faculty of Computer and Information Science

Andraž Kohne

Monte-Carlo Tree Search in chess endgames

UNDERGRADUATE DISSERTATION

UNDERGRADUATE INTERDISCIPLINARY UNIVERSITY STUDY
COMPUTER SCIENCE AND MATHEMATICS

Mentor: akad. prof. dr. Ivan Bratko

Ljubljana 2012

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Andraž Kohne

Monte-Carlo preiskovanje v šahovskih končnicah

DIPLOMSKO DELO

UNIVERZITETNI ŠTUDIJSKI PROGRAM PRVE STOPNJE
RAČUNALNIŠTVO IN MATEMATIKA

Mentor: akad. prof. dr. Ivan Bratko

Ljubljana 2012

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

No. of dissertation: 00005/2012
Date: 02.04.2012

University of Ljubljana, Faculty of Computer and Information Science issues the following dissertation:

Candidate: **ANDRAŽ KOHNE**

Title: **MONTE CARLO TREE SEARCH IN CHESS ENDGAMES**

Type of dissertation: Undergraduate dissertation

Topic of the thesis:

Monte Carlo tree search (MCTS) has proved to be a very successful technique for computer playing of the game of go. It is not clear, however, how successful it is when applied to other games, like chess. The task of your project is to apply MCTS to chess endgames, such as the king and rook vs. king endgame. Pay special attention to the setting of the parameters C and T of MCTS. On the basis of your experimental results, propose recommendations regarding these settings.

Mentor:

[illegible]

Acad. Prof. Ivan Bratko, PhD

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dean of Faculty of Computer and Information Science

[illegible]

Prof. Nikolaj Zimic, PhD

UNIVERZA V LJUBLJANI • Fakulteta za matematiko in fiziko • 1

Dean of Faculty of Mathematics and Physics

[illegible]

Acad. Prof. Franc Forstnerič, PhD

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00005/2012

Datum: 02.04.2012

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko ter Fakulteta za matematiko in fiziko izdaja naslednjo nalogo:

Kandidat: **ANDRAŽ KOHNE**

Naslov: **MONTE CARLO PREISKOVANJE V ŠAHOVSKIH KONČNICAH**
MONTE CARLO TREE SEARCH IN CHESS ENDGAMES

Vrsta naloge: Diplomsko delo univerzitetnega študija prve stopnje

Tematika naloge:

Monte Carlo preiskovanje se je izkazalo kot izredno uspešno pri računalniškem igranju igre go. Ni pa še jasno, kako se obnese pri drugih igrah, npr. pri šahu. V tej diplomski nalogi izvedite poskuse z uporabo tega pristopa k igranju šahovskih končnic, kot je npr. končnica kralja in trdnjave proti kralju. Posebej se posvetite nastavljanju parametrov C in T Monte Carlo preiskovanja. Na osnovi eksperimetalnih rezultatov izdelajte priporočila za nastavljanje teh parametrov.

Mentor:

[illegible]

akad. prof. dr. Ivan Bratko

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan Fakultete za računalništvo in informatiko:

prof. dr. Nikolaj Zimic [illegible]

Dekan Fakultete za matematiko in fiziko:

UNIVERZA V LJUBLJANI • Fakulteta za matematiko in fiziko • 1

akad. prof. dr. Franc Forstnerič [illegible]

Zahvaljujem se akad. prof. dr. Ivanu Bratku za mentorstvo.

Za pomoč pri lektoriranju in za mnogo nasvetov glede oblike besedila bi rad omenil tudi Mojco Miklavec. Dolgujem ji veliko zahvalo, saj je prostovoljno ponudila pomoč. Z upoštevanjem njenih nasvetov glede oblike besedila, diplomsko delo izgleda veliko lepše.

Predvsem se pa zahvaljujem dr. Mateju Guidu za vso potrpežljivost in pomoč. Zahvaljujem se mu, ker me je seznanil z Monte-Carlo preiskovanjem in ker me je vodil in mi svetoval čez celotno izdelavo te diplomske naloge. Posebej bi se mu tudi zahvalil za pomoč pri samem pisanju diplomske naloge, saj sem se s pisanjem strokovnih besedil prvič srečal in mi brez njegove pomoči nikoli ne bi uspelo spraviti besedila v obliko, kot jo ima sedaj.

Contents

Razširjeni povzetek

Monte-Carlo preiskovanje združuje splošnost naključnih simulacij s preciznostjo preiskovanja dreves. Je algoritem za preiskovanje dreves in gradnjo dreves, ki gradi delno drevo igre. Vsako vozlišče v tem drevesu predstavlja neko stanje v izbrani domeni. V naši domeni vozlišče tako predstavlja neko stanje na šahovnici (ali drugače: neko šahovsko *pozicijo*).

Monte-Carlo preiskovanje je leta 2006 vzbudilo veliko pozornosti s strani raziskovalcev zaradi uspeha v igri Go. Remi Coulom je s programom, ki za igranje igre Go uporablja Monte-Carlo preiskovanje, imenovanim Crazy Stone, dobil zlato medaljo na 11. računalniški olimpijadi.

Za igro Go ne poznamo dobre ocenjevalne funkcije. Zato z uporabo tradicionalnih algoritmov ni bilo možno ustvariti programa, ki bi lahko igral bolje od človeškega igralca. Monte-Carlo preiskovanje pa se je izkazalo za učinkovit algoritem v igri Go, saj ne potrebuje ocenjevalne funkcije. Namesto ocenjevalne funkcije uporablja naključne simulacije.

Simulacije usmerjajo preiskovanje in gradnjo drevesa v algoritmu. Za delovanje algoritma zadoščajo le informacije, ki jih Monte-Carlo preiskovanje dobi od izvajanih simulacij.

Algoritem je neodvisen od domene, v kateri je uporabljen. Za delovanje potrebuje le vse možne akcije in končna stanja. To je dovolj za izvajanje simulacij. Kar pomeni, da ga je možno implementirati v drugih domenah, z le malo spremembami.

Kljub temu ta pristop ni bil pogosto uporabljan v igri šaha. Tema te naloge je poskus uporabe Monte-Carlo preiskovanja v igranju šahovskih končnic. Cilja te naloge sta:

- ugotoviti pri kakšni konfiguraciji algoritem igra najbolje v izbrani domeni,
- najti povezave med parametri Monte-Carlo preiskovanja in s tem dobiti vpogled, ki bi lahko povečal učinkovitost algoritma, ne glede na izbrano domeno.

V ta namen sem razvil program, ki uporablja Monte-Carlo preiskovanje za igranje šaha. Program smo uporabili v šahovskih končnicah. Z njegovo pomočjo smo tudi bolj podrobno opisali delovanje algoritma.

Za šahovske končnice obstajajo zbirke podatkov, ki za vsako pozicijo podajo njeno vrednost oz. razdaljo do mata. Če primerjamo pozicijo po potezi, ki jo je odigral naš program, s pozicijo po potezi idealnega igralca, lahko ocenimo kvaliteto posamezne poteze. Kvaliteto igre v tej nalogi merimo s povprečjem kvalitete posameznih potez belega igralca (ki ga usmerja naš program).

Pokazali smo, da mera tudi določa povprečno dolžino odigrane igre. Kot domeno smo si izbrali šahovsko končnico, kjer beli poskuša s kraljem in trdnjavo matirati črnega. V tej končnici je našemu programu, ki upravlja belega igralca, uspelo matirati črnega igralca v skoraj vseh igrah. Tako je povprečna dolžina matiranja dobra mera kvalitete igre.

Z uporabo te mere smo prišli do naslednjih zaključkov.

- Pokazali smo, da je sposobnost algoritma, da predvidi možne poteze nasprotnika, zelo povezana z njegovo učinkovitostjo. Če nasprotni igralec izbere potezo, katera vodi v pozicijo, ki je ni v drevesu, pride do sesutja drevesa. V drevesu ostane le koren in vse pridobljene informacije so izgubljene.

  Katere pozicije se bodo dodale v drevo, določa kvaliteta simulacij. Sposobnost simulacije, da predvidi poteze nasprotnika, je tako zelo pomembna za učinkovitost algoritma.

- *Prag* T je eden od parametrov Monte-Carlo preiskovanja, je prag, ki določa ali bo algoritem v izbranem vozlišču izbral naslednika, ki ga oceni kot najboljšega, ali pa bo začel izvajati simulacijo. Če je bilo vozlišče obiskano manjkrat, kot določa prag T, se izbere najbolje ocenjen naslednik, sicer se začne izvajanje simulacije.

  Nova vozlišča se v drevo dodajo samo, ko igra v simulaciji pride v pozicijo, ki še ni predstavljena v drevesu. Prag T tako določa, kakšno je največje možno število vozlišč na nekem nivoju drevesa. Prag T je tako eden od pomembnejših parametrov. V tej nalogi smo prišli do zaključka, da je za učinkovitost algoritma dobro nastaviti prag T na največje število možnih potez za igralca.

  Če je prag T manjši od največjega števila možnih potez, v drevesu ne morejo biti predstavljene vse možne pozicije. Nova vozlišča pa v drevo pa dodajo naključne simulacije, zato obstaja verjetnost, da tudi pomembnejše pozicije ne bodo dodane v drevo. Kar pa seveda pomeni slabšo kvaliteto igre.

  Prav tako previsok prag T tudi slabo vpliva na kvaliteto igre. Tudi če so že dodani vsi možni nasledniki, program namreč začne izvajati naključno simulacijo, namesto da bi izbral doslej najbolje ocenjenega naslednika. Kar lahko pomeni, da bo algoritem več časa namenil veji drevesa, ki ni najboljša.

- Konstanta C je parameter, ki določa v kolikšni meri bo Monte-Carlo preiskovanje izkoriščalo že dobljeno znanje. Če je C manjši, bo algoritem bolj usmeril preiskovanje v vozlišča, ki se do tedaj zdijo bolj obetavna. Pri večjem C pa bo več časa usmeril v vozlišča, ki so bila manj raziskana.

  Pokazali smo, da če damo algoritmu na voljo več časa za izvajanje simulacij, je bolje, da tudi povečamo konstanto C. Če so bila vozlišča dovolj dobro raziskana, potem z izvajanjem novih simulacij na teh vozilščih ne pridobimo nič novega znanja. Z raziskovanjem ostalih, manj

raziskanih vozlišč pa lahko najdemo novo, bolj obetavno vozlišče.

Premajhno število simulacij, izvajanih iz nekega vozlišča, lahko pomeni, da to vozlišče ne bo pravilno ocenjeno. Zaradi tega C ne sme biti prevelik, posebej pri fiksnem številu simulacij. Saj večji kot je C, bolj bo algoritem enakomerno preiskoval vsa vozlišča, ne glede na to kako so bila do tedaj ocenjena.

S to nalogo upamo, da smo vsaj malo pripomogli k še boljšemu razumevanju Monte-Carlo preiskovanja in verjamemo, da bo naloga služila kot podlaga za nadaljnje raziskovanje.

Ključne besede: Monte-Carlo, preiskovanje, šah, tablebases, šahovska končnica

Abstract

The Monte-Carlo Tree Search (MCTS) algorithm has in recent years captured the attention of many researchers due to its notable success in the game of Go. In spite of this success, so far it has not been used much in the game of chess. In this thesis, we attempt to apply MCTS to chess endgames. The reason for this is the existence of chess tablebases, i.e. databases that provide an exact value of each chess board position in terms of distance to mate. With this information at disposal we are able to measure more objectively the quality of play, and thus assess how well the Monte-Carlo Tree Search algorithm performs. We propose some guidelines about how the algorithm should be configured to achieve good performance in chess endgames. The question remains whether our findings are applicable to other domains as well.

Keywords: Monte-Carlo, tree, chess, tablebases, endgame, search

Chapter 1

Introduction

Monte-Carlo Tree Search is a best-first tree search method guided by playing out random simulations. It is built upon two fundamental concepts: that we can approximate true value of an action by performing random simulations; and that these values may be used to adjust the policy towards a best-first policy. The algorithm uses previously played random simulations to guide the building of a partial game tree. This approach was formalized into the UCT algorithm by Kocis and Szepesvari [1, 3, 4, 5].

Monte-Carlo Tree Search caught the attention of researchers in 2006 when a computer program Crazy Stone won a gold medal at the 11th Computer Olympiad. Crazy Stone is a program written by Remi Coulon which uses Monte-Carlo Tree Search algorithm for playing GO.

Another Monte-Carlo Tree Search based program MoGo was also first computer program to defeat a professional human Go player, when it achieved a victory against Guo Juan on 2007, further increasing the interest in the algorithm [2, 5].

The main advantage of Monte-Carlo Tree Search is that it does not rely on a position evaluation function, instead it uses random simulations to evaluate a game position. This is especially important in the game of Go. Go is a traditional Japanese board game that has high branching factor and lacks good static evaluation function for non-terminal board position. Making it a very challenging game for traditional tree search approaches which require a good static evaluation function.

Monte-Carlo Tree Search is up to date still the most promising direction in achieving human-competitive computer player for Go [3].

Even though there has been at least 150 research papers written on topics related to Monte-Carlo Tree Search since its inception. This field of study is still very interesting research topic in artificial intelligence, with many opened research questions [1].

By applying Monte-Carlo Tree Search to chess endgames, where information about game play quality is available, we hoped to demonstrate how algorithm performs in relation to input parameters. We focused on examining relations between these parameters and how they influence its performance. For optimal performance, currently values of these parameters are determined empirically. By exploring these relations we hoped to achieve that at least some of the parameters could be calculated to achieve optimal performance in a given domain.

1.1 Thesis goals

The purpose of this this thesis is to better understand how Monte-Carlo Tree Search performs in relation to input parameters. To achieve this we applied it to the KRK endgame in the game of chess.

We chose KRK endgame for two reasons: first the attacking player (controlled by the algorithm) can easily achieve a checkmate against the defending player. The second reason is that for chess endgames tablebases are available.

Chess tablebases give us a way to objectively measure the quality of play by providing number of moves to deliver checkmate from any given position assuming optimal play by both players.

We set our goals to be:

1. To discover combination of parameter values where Monte-Carlo Tree Search performs best in our domain.

2. To discover possible relations between the Monte-Carlo Tree Search parameters. Thus gaining knowledge that could improve performance regardless of domain.

1.2 Results

We used data from performed experiments to:

- to show that if we give Monte-Carlo Tree Search more time for performing simulations, it is good idea to configure it so that it explores higher number of different nodes.
- to show that from a given node H, it is best to perform random simulations until that node has been visited as many times as there are possible moves. Only after that we should use information gotten from these simulations, to decide on which child of node H should we put more focus.

1.3 Structure of the thesis

- Chapter 1 introduces Monte-Carlo Tree Search and describes goals and structure of this thesis.
- Chapter 2 defines and describes some terms used in this thesis.
- Chapter 3 describes how Monte-Carlo Tree Search works and how our program for playing chess endgames was implemented.
- Chapter 4 explains how experiments were conducted and contains conclusions that we can draw from these experiments.
- Chapter 5 summarizes this thesis.

Chapter 2

Domain description

This chapter contains definitions and explanation on some terms used in this thesis.

2.1 Chess

Chess is a board game for two players. Each player has sixteens pieces and the object of the game for a player is to trap enemy king piece in a position where it is attacked regardless of which move enemy plays. This position is called a *checkmate* and it represent a loss of a player who's king was trapped in this way. Chess game consists of three phases:

- opening,
- middlegame,
- endgame.

In this thesis, we are only interested in *endgames*. Endgame in chess is a stage of the game where only a few pieces remain on board. The reason why we used chess endgames as our domain are tablebases (see Section 2.2). For more detailed description and rules of the game see [9].

2.1.1 KRK endgame

KRK endgame is a chess endgame where an attacking player has two pieces: a king and a rook and the defending player has only a king piece. In our program only this endgame was used.

Every match our computer program played used the same chessboard state as starting position. From this position if attacking player played perfect game, checkmate was achieved in 15 moves.

In our program, white player is also the attacking player. In the thesis white player is a synonym for attacking player and black player is a synonym for defending player.

2.2 Nalimov tablebases

An endgame tablebases is a database containing various precalculated information of every chess endgame position. We used it for two reasons: it enables our program to play best possible moves for defending player and to objectively determine the quality of play. Quality of play is calculated by examining the discrepancy between number of moves needed to reach a checkmate position determined by moves played by attacking player and by perfect player. [7]

2.2.1 Distance to Mate (DTM) difference

We used *Distance to Mate difference* to determine the quality of game play. It represents the deviation from optimal play. It is calculated with the help of tablebases.

Tablebases provided us with information of how many moves are needed to achieve a checkmate for the attacking player if both players play optimal. We called this *Distance to Mate value*. With the help of Distance to Mate value we defined DTM difference as:

$$DTM\ difference = DTM_{played} - DTM_{optimal} \tag{2.1}$$

where DTM_{played} represents Distance to Mate value of the position reached after played move and $DTM_{optimal}$ represents Distance to Mate value of the position after best possible legal move was played.

Best possible move minimizes Distance to Mate value, so DTM difference is non-negative. If DTM difference of played move is 0, optimal move was played. Lower DTM difference means better quality of play [8].

Chapter 3

Monte-Carlo Tree Search

3.1 What is Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) combines generality of random simulation with precision of tree search. It is a tree searching algorithm that builds a partial game tree guided by (pseudo) random simulations.

Monte-Carlo Tree Search offers many advantages over other tree search methods:

- Even though MCTS does not require evaluation function or strategic knowledge, it can make use of them thus becoming even more effective [6].
- It can function effectively knowing only the rules of the game (which can mean knowing only all legal moves and terminal positions). As a result a single implementation can be easily reused for different domains with only small modifications [1].
- MCTS algorithm visits more interesting nodes more often and focuses on searching more relevant parts of the tree. This property makes MCTS effective in games with high branching factor like the game of GO [3, 1]. Such games can usually cause problems for other algorithms, but due to its adaptive nature MCTS may handle them effectively.

- The algorithm can be stopped at any moment, to return best estimate. Game tree that was built so far can be discarded or preserved for future use [1].

However MCTS is not without drawbacks:

- In its basic form MCTS may not perform efficiently in games with medium complexity. Some domains may require too many simulations for MCTS to evaluate position properly in a reasonable amount of given time [1].
- Ramanujan argued that MCTS may not yield good results in domains with many trap states [6]. This is why good simulation strategy is very important for the algorithm to perform efficiently, so that it can avoid those trap states.
- In domains where simulations are slow or where too many iterations are required to find a good solution, performance issues may arise. This flaw may be avoided with improvements such as adding domain knowledge or detecting and removing trap states in domains where this is possible [1].

3.1.1 Basic algorithm

Monte-Carlo Tree Search is a best first algorithm tree search method. It uses random simulations (also called *playouts*) for evaluation of a game positions (also called *game states*) instead of an evaluation function. These random playouts guide construction of partial game tree.

Its aim is to select the most promising node by randomly exploring the tree space with random playouts. It is very simple in its design and domain independent.

Basic principle behind MCTS (or any algorithm built upon Monte-Carlo method) is: “From a single random game, very little can be learnt. But from simulating a multitude of random games, a good strategy can be inferred.”

[11]. So for Monte-Carlo Tree Search to be effective, many iterations need to be executed.

In MCTS, each node represents a given position. Every node must contain at least two pieces of information:

1. Current value of a node. This information is gained through previously played random simulations.

2. How many times was node visited (this is also called *visit count*).

3.1.2 Structure of MCTS

MCTS consists of four main phases executed in following order:

1. *Selection* is a recursive task with the goal of selecting the most promising child of a given node. It traverses the game tree from the root node until it reaches position E which is not yet part of the tree, selecting most promising positions along its path.

2. *Expansion* adds a node representing position E to the tree.

3. *Simulation* performs random simulation(s) (playouts) from a position E according to the Simulation Strategy.

4. *Backpropagation* updates values of previously traversed nodes with information gained from the Simulation step.

3.2 MCTS phases

In this section, we describe the four MCTS phases (Selection, Expansion, Simulation, Backpropagation) in more detail.

3.2.1 Selection

Selection traverses the tree from the root node until a position is reached that is not a part of the tree yet, selecting most promising positions according to *Selection strategy* along its path.

Selection strategy controls the balance between exploration and exploitation. This strategy consists of selecting nodes that leads to best results so far (exploitation). However due to uncertainty of the evaluation other nodes that might not appear promising should be explored (exploration).

Our Selection strategy was very simple to implement as it consists of choosing positions which maximize the following UCT formula:

$$\frac{m}{n} + C \cdot \sqrt{\frac{\ln N}{n}} \tag{3.1}$$

Where:

- n is the number of times node has been visited.
- m represents the number of checkmates achieved by the algorithm when it traversed this node.
- C is a constant which determines the level of exploitation/exploration.
- N is a visit count of the node's parent.

From a single playout very little can be learnt, however from a multitude a sound solution may arise. Which is why in most implementations selection phase is only applied to a node if visit count is higher than a constant called *threshold T*. Otherwise Simulation phase is applied. [5]

3.2.2 Expansion

Expansion adds new nodes to the tree. Usually one node per simulation is added. This is done when algorithm reaches a position which is not yet in the three.

3.2.3 Simulation

Simulation step plays the game until it runs out of time or until a terminal state is reached. During the play, moves are selected pseudo randomly according to the *Simulation strategy.*

Implementation of Simulation strategy

We implemented simulation strategy in a very basic way. First all possible legal actions are generated and then these actions are filtered with what we call (for the lack of a better term) heuristics methods. These methods act as filter functions. For example, if we use heuristic method that prevents white player from loosing any piece, this function will remove any action where white could potentially lose a piece.

For simulation strategy players relied on the following simple heuristics:

- Black player stays as near the center of chessboard as possible.
- White player shouldn't loose any pieces.
- White king should not increase distance from black king[1].
- White king should move if distance between kings is larger than 3.
- If kings are in opposition, white player tries to check the black king from the side (opposition of kings in chess is a position on the chessboard where two kings face each other on the same rank or a file being exactly one square apart).
- White player evades chess board state repetition.

3.2.4 Backpropagation

Backpropagation updates values of previously visited positions with information gained from simulation. Backpropagation is usually a very simple

[1]Whenever distance between squares or pieces in mentioned in the thesis we are actually referring to Manhattan distance.

method. In our program it updates every node's value to the value of equation 3.2.

$$\left(\frac{\text{checkmates achieved from position } A}{\text{visit count of position } A}\right) \tag{3.2}$$

Where position A is the starting position of simulation phase.

3.3 Implementation

For experiments, we implemented MCTS and the KRK rules in Java which can be downloaded from git repository:

`https://github.com/ak83/MCTS.git`.

Instructions on how to use the program can be found at

`https://github.com/ak83/MCTS/wiki`.

In the program we implemented MCTS and chess rules. For calculating DTM difference of a chess move we used a chess program Fruit (version 2.3.1).

3.3.1 Playing moves

In our implementation, the white player is the attacking player, controlled by the MCTS algorithm and trying to achieve a checkmate against the black (defending) player. White player wins the game if it manages to checkmate the black player.

White player can only play moves that are in the tree. It chooses *top level node* (a child of the root node) with the highest visit count.

Black player always plays an optimal move which puts him in a position with the greatest distance to checkmate.

The tree built by MCTS algorithm is a partial game tree. Meaning that not all positions are in the tree so black's response is not always anticipated. If black player plays a move which was not anticipated, the tree is reset to the root node and all previously gained information is lost. We call this a *tree collapse*.

Before every white player ply and at the beginning of every match many iterations of MCTS algorithm were completed (exact amount of MCTS steps performed is specified in Section 4.1).

Chapter 4

Experiments

4.1 Experimental setup

For each set of MCTS input parameters (C, threshold T, number of MCTS iterations performed before white player's ply) we ran 100 chess games. In total there were 26,000 playouts performed. All matches started from the same starting position where black player was at least 15 moves away from a checkmate.

In our experiments we focused on the following parameters:

C controls the amount of exploitation/exploration (lower C values favour exploitation, higher C values favour exploration).

Values used	0, 0.12, 0.25, 0.75, 1.0, 1.75, 2.5, 5, 10, 25, 50, 100, 500

Threshold *T* determines if simulation or selection is performed from a given position. If position has a visit count higher than the threshold T, the selection strategy is applied otherwise the simulation strategy is applied.

Values used	5, 10, 30, 50, 75, 100, 150, 300, 500, 1000

Number of MCTS simulations is a number of MCTS iterations that were performed before each white player's ply.

Values used	3000, 6000

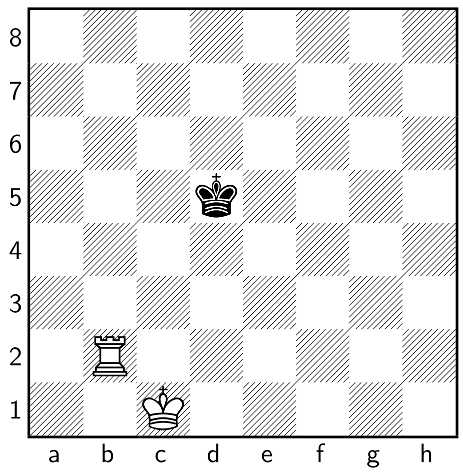

Figure 4.1: Starting position used in our experiments

In addition, before every match there were also 10,000 MCTS iterations completed.

4.2 Results

4.2.1 Measuring performance

This section explains why we used DTM difference as a method of measuring quality of play and why it is a viable measurement.

Average DTM difference may not be directly connected with winning ratio of attacking player. Especially in more difficult endgames the attacking player can make one or two wrong moves that may prevent him from achieving a checkmate, despite playing optimally up to that point.

The main reason why we chose the DTM difference for measuring playing quality are the tablebases (Section 2.2). Tablebases are a great tool that allowed us to exactly calculate the DTM difference of a played move. With average DTM difference we may accurately calculate quality of an individual move played.

Figure 4.2 shows that the average game length and the average DTM difference are related. With better performance of the attacking player (lower DTM difference) the average game length becomes shorter, meaning that the attacking player wins the game faster.

4.2.2 Basic results

All results described in this Section were expected and we use them to explain how MCTS works.

Level of exploitation/exploration of MCTS is controlled by the UCT formula, more specifically by the constant C. With higher *exploitation* MCTS focuses more on the previously gained information and does not try to explore less significant positions. In contrast, higher *exploration* means that it will explore also less significant positions. This property is important due to uncertainty of evaluation by simulations, a position which may seem good at some point, might turn out to be suboptimal later on.

In chess games where 3,000 MCTS iterations were performed, we took a sample from each combination of C and threshold T, and for each value of C we calculated standard deviation of visit count for top level nodes. These samples were taken when the attacking player played his first move at the beginning of the chess games. In figures 4.3 and 4.4 we calculated the standard deviation of top-level nodes. Figure 4.3 explains best how our algorithm works. With higher exploration (higher C values) standard deviation of visit count taken from the top-level nodes decreases. Due to heuristics used, there were usually usually 2 or 3 top-level nodes. In playouts where exploitation was high, typically only one of these nodes was explored properly. This is especially true for games where there was no exploration (if C was set to 0). If a checkmate was reached in the first iteration of MCTS, all other top-level node were ignored.

In playouts with higher exploration, all top-level nodes were explored equally. Which may not be optimal, because we want the algorithm to explore significant parts of the tree more extensively.

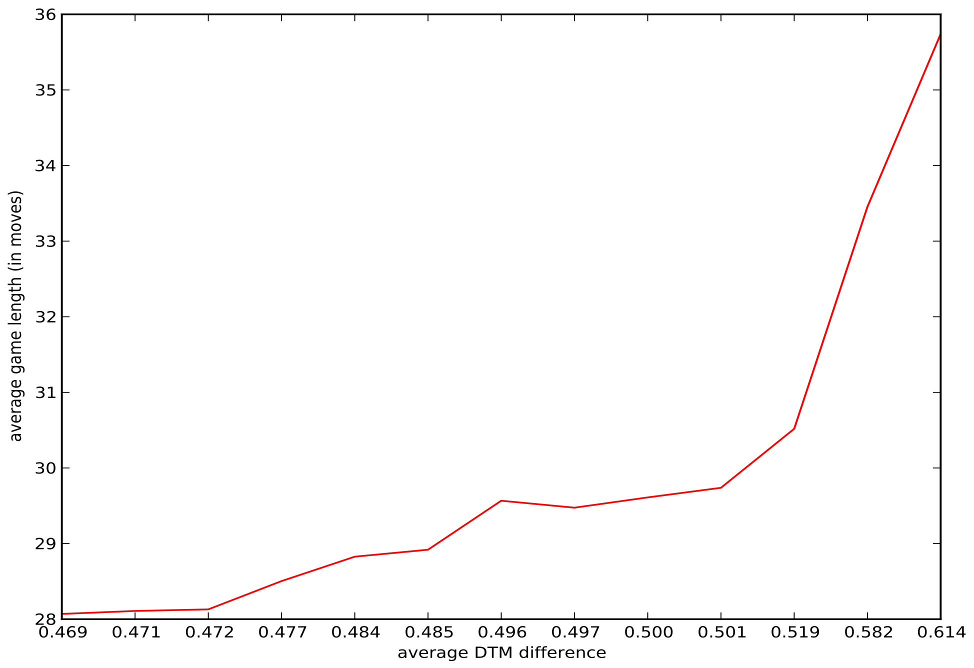

Figure 4.2: Average game length based on the DTM difference. We can see that if the DTM difference increases average game length also increases. The graph demonstrates that average DTM difference can be used as a valid measurement of a player's skill. The attacking player did not play very well, because in our tests we used starting position that is 15 moves away from checkmate if both players play optimally. However as we can see from the graph, average game lengths ranged from 28 to 36.

Setting C value for optimal performance is not an easy task. Nevertheless there are some guidelines how to find a good value (see Section 4.2.3 for more details). Also optimal C values are domain dependent and different for each set of other MCTS input parameters.

As can be seen in figure 4.3 standard deviation of times that node was visited by the algorithm decreases with increasing values of C. This demonstrates how Monte-Carlo Tree Search and UCT formula work. The C constant determines *exploitation/exploration* of the MCTS algortihm. With lower C values the MCTS algorithm will rely on exploiting knowledge that has already been found and will not explore nodes that do not seem good choice early on. With higher C values the MCTS algorithm will tend to explore more different nodes. As a consequence of this, also standard deviation of average number of checkmates achieved will decrease with increasing of C value (see Figure 4.4).

Importance of a good simulation strategy

Figure 4.6 shows that the quality of play is related to the number of tree collapses. A tree collapse occurs when the black player plays a move which is not in the tree, i.e. the move that was played was not anticipated by the algorithm. In tests where fewer tree collapses occurred, the quality of play was better (lower DTM difference).

The data for Figure 4.6 was collected in following way: for each set of C and threshold T average DTM difference and number of tree collapses were collected. Both DTM difference and number of tree collapses were then normalized to fit on the interval $[0, 1]$. By looking at the figure it is easy to see that a relation between them, since in most graphs when the number of tree collapses drops also DTM difference drops.

It is easy to conclude that inability to correctly guess black player's response in a given game state can be a mayor flaw in designing a good simulations strategy.

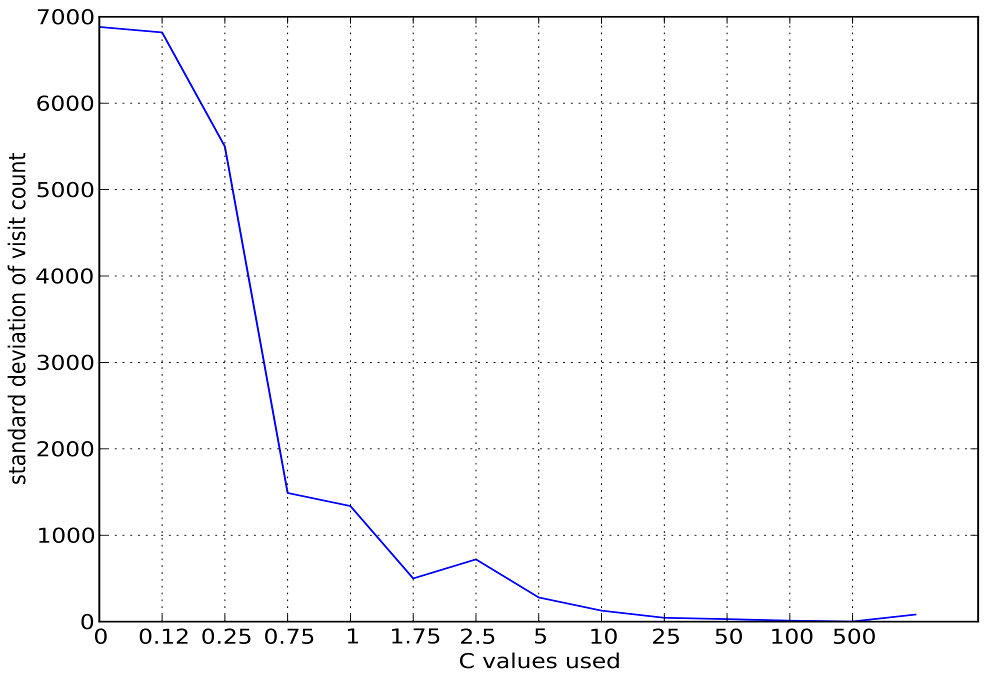

Figure 4.3: Average standard deviation of visit count at top-level nodes for different C values. With lower C values (higher exploitation and lower exploration) less top-level nodes are explored and search is more focused on positions which appear a good choice very early on. With the higher C value (increased exploration) more positions which may not appear significant will be explored. Until a point where C is high enough so that all top-level nodes will be explored equally causing tree building process to ignore information gain from simulations performed.

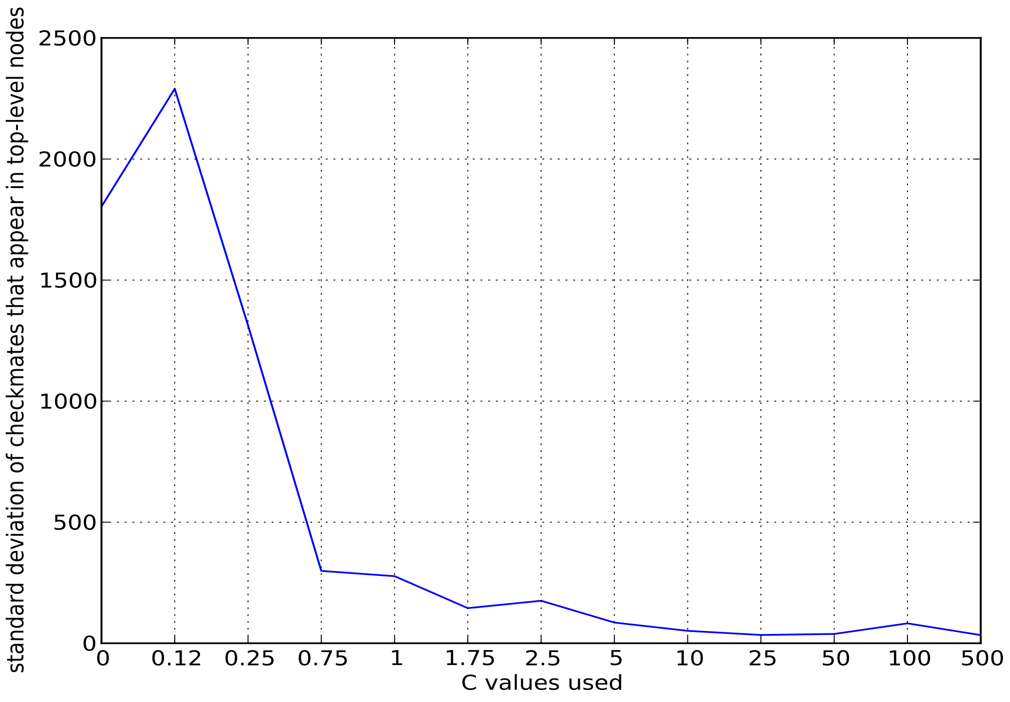

Figure 4.4: Standard deviation of checkmates achieved from top-level nodes. Checkmates are very important because they guide building of the tree. Lower exploration (low C values) cause only very few top-level nodes to be explored. Checkmates then appear only in those few nodes and cause tree building process to be sub-optimal. Because the first node from which checkmate is achieved may not be the node from which checkmate is achieved in least number of moves.

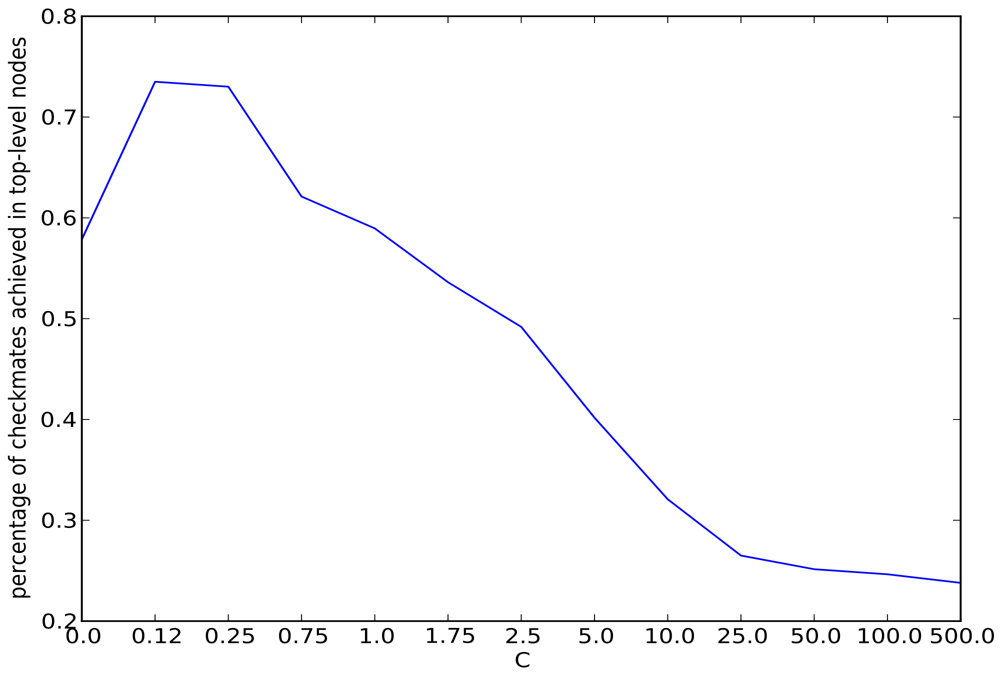

Figure 4.5: In this figure we see the ratio between top-level node's visit count and a number of checkmates achieve from it. With higher exploration (higher C values) more different nodes are explored. However, if fixed amount of simulations is available, by distributing simulations to more different positions, there are less simulations available for evaluation of each node. If less simulations are performed from a position, MCTS might gain too little information for proper evaluation. Meaning that quality of MCTS guided play might drop if we simply keep increasing exploration. However if we set exploration too low, then not all significant positions would be explored.

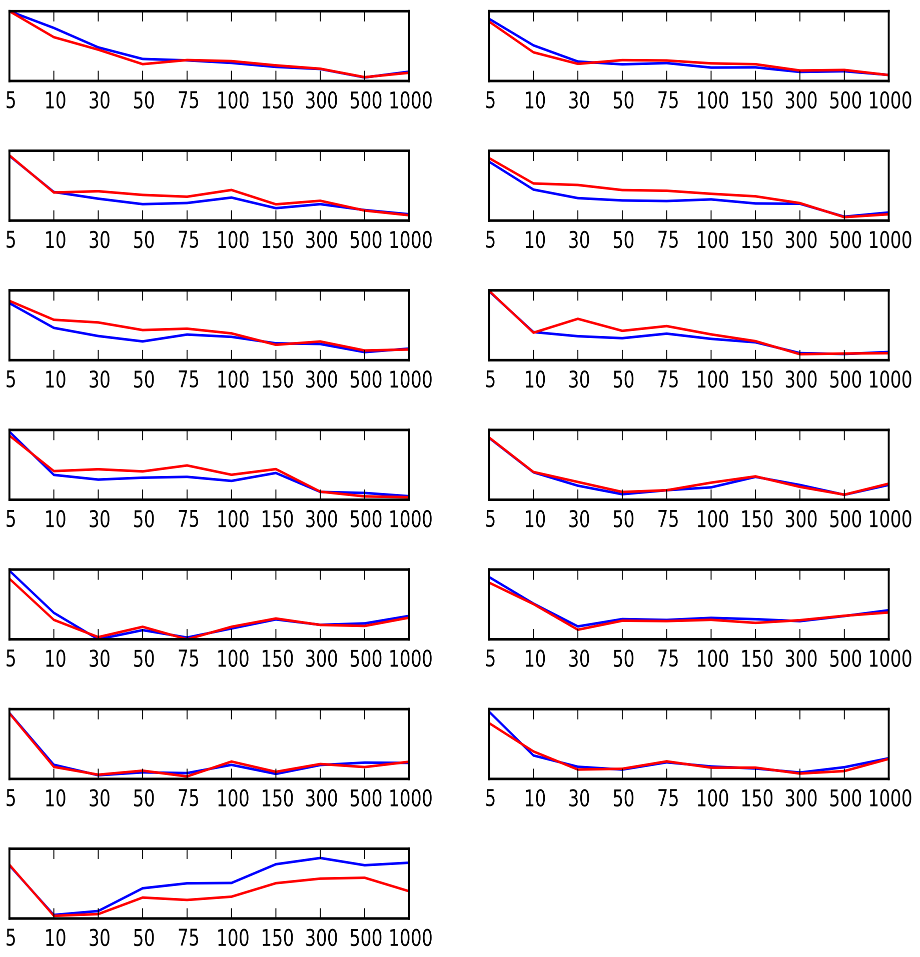

Figure 4.6: The Figure shows that the quality of play and the number of tree collapses are related. Both DTM difference (y axis) and the number of tree collapses have been normalized to fit on the interval $[0, 1]$. Blue lines represents DTM difference and red line number of MCTS tree collapses. The x axis of each graph represents threshold T values used in tests.

4.2.3 Relation between MCTS parameters

Relation between *C* and the number of MCTS iterations

In Figure 4.7, we can see an average DTM difference based on the value of C. What is the most important is for which C value algorithm's performance is optimal (where DTM difference is minimum). This value is different for chess games with different number of MCTS iterations made.

Our claim is that with higher number of MCTS iterations performed, it is better to increase exploration.

With higher C values standard deviation of visit count decreases (Figure 4.3), because a certain amount of simulations must be divided between a fixed amount of nodes. If we increase the number of simulations and if we don't increase exploration, simulations will begin mostly from nodes that have already been evaluated. If those nodes were already properly evaluated, we gain nothing from increased number of simulations. However if we increase exploration, new simulations can better evaluate a node that may not have been properly evaluated before.

Finding optimal C value is not an easy task. If we just increase the exploration, MCTS performance would not be optimal. If we have a fixed amount of simulations available and we increase exploration too much, the algorithm will explore more different positions. Even though more positions would get explored, from these positions fewer simulations would be performed. With fewer simulations, evaluation of those positions may not be up to par.

Relation between *C* and threshold *T*

For each value of C used, Figure 4.8 shows DTM difference based on different threshold T values. We can observe that for lower exploration the quality of play increases with threshold T.

If exploration is set too low, the MCTS focuses too much on certain positions, before enough different nodes have been explored to determine which position is a good choice. In such case higher threshold T provides

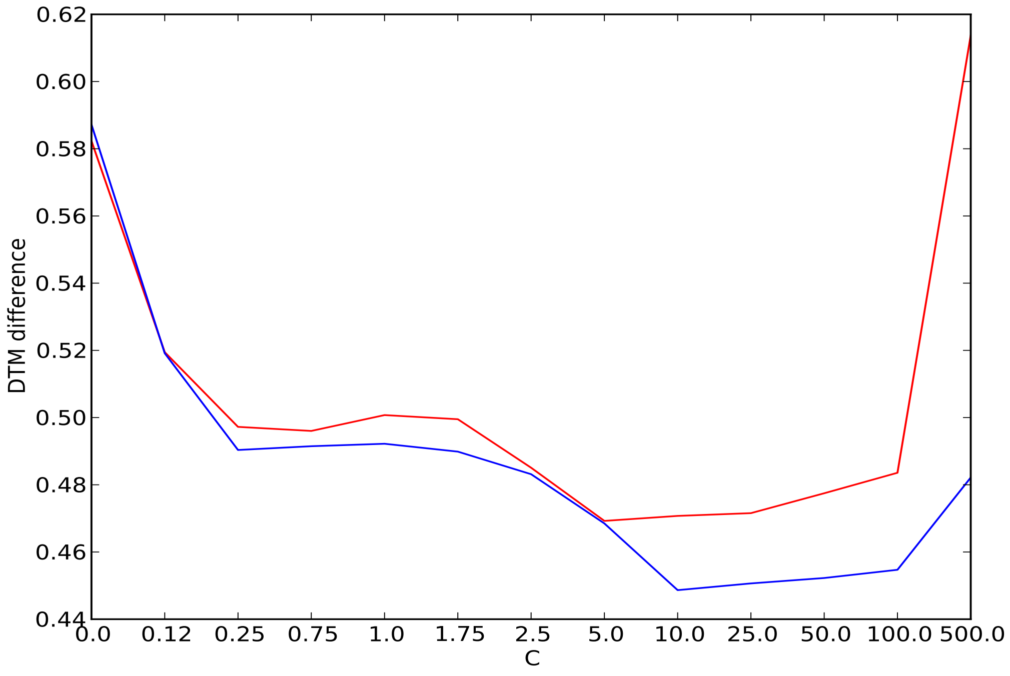

Figure 4.7: Average DTM difference for C values used in games with 6000 (blue line) and 3000 (red line) MCTS iterations performed per white player's move. Better quality of play is represented with lower values of DTM difference. This Figure suggests that it is better to choose higher exploration where more time for performing simulations is available. Because the attacking player's skill is the best at C value 5 in games with 3000 MCTS iterations made and 10 in games with 6000 iterations made. In matches where the attacking player made 6000 MCTS iterations, the attacking player played with more skill than in those with 3000 MCTS iterations available.

extra exploration by performing random simulations that start closer to the top of the MCTS tree.

This is however not true for games performed with higher exploration. In these games the white player plays best (or close to best) when threshold T is set to 30. We believe this is because in the KRK endgame there are at most 22 moves possible for attacking player. If threshold T is set to a value that is lower, not all possible move may be added to the tree and crucial moves may not be played. However if threshold T is set to a value higher than that, MCTS should add all possible moves to the tree.

Although when all of possible moves are added with threshold T set higher, instead of using the information gained, the algorithm performs random playouts. Which is not optimal.

In our tests, for games with sufficient exploration level algorithm performed best at threshold T with the value of 30 incidentally, this is the closest value that is also higher than the maximum number of legal moves possible.

The claim about this relation can be further confirmed by the fact that it has been empirically determined that for the game of Go played on 9×9 board, it is best to set threshold T to the value of 81. Which is also the maximum number of possible moves for a player [10] in that game.

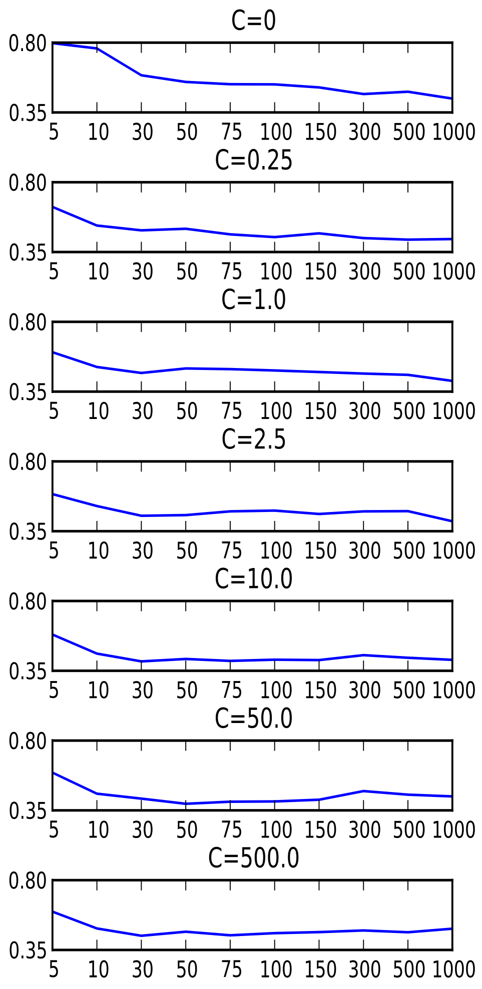

Figure 4.8: This picture shows how threshold T and C constant are related. The x-axis represents the threshold T values used and the y-axis represents the DTM difference. For chess games where C was set to 2.5 or lower DTM difference decreases with increasing threshold T. For chess games where C was set to 5 or higher DTM difference reaches minimum or is near minimum in chess games where threshold T was set to 30.

Conclusions

In this thesis, we applied the Monte-Carlo Tree Search to chess endgames. We chose this domain because with the help of tablebases, giving an exact evaluation of each algorithm's decision in terms of distance to mate, we could assess more objectively the algorithm's performance. For this purpose we used *DTM difference*. With the DTM difference we could assess the quality of each move and consequently the quality of the algorithm's decisions.

The DTM difference offers a better way to measure game quality (compared to average game length, for example). It represents a deviation from optimal play, and could be assigned to each particular move. Thus using the DTM difference we could assess the quality of each individual move played (and not only of a game as a whole), making it more feasible to analyze MCTS performance.

It is worth noting that MCTS performed rather poorly in our domain of choice – chess endgames. While our MCTS-based program won almost every game in the KRK endgame, it rarely achieved a checkmate when tested in more difficult endgames, even with help of additional heuristics. The main reason is that during simulations too few checkmates were achieved to properly guide the tree search.

Although we applied MCTS to chess endgames, the purpose of this thesis was not to create a chess-playing agent. Our goal was to understand Monte-Carlo Tree Search better.

By using the DTM difference we came to following conclusions.

- By exploring the relations between various parameters, we showed that by increasing the computational time to perform simulations it is also a good strategy to also increase exploration. By running simulations from nodes that were already evaluated properly hardly any information can be gained. However, using the extra time to explore previously less explored nodes, a better solution may be found.

- We argued that by setting the threshold T to a value equal to the maximum of possible actions in a given domain, the Monte-Carlo Tree Search will perform best. If the threshold T is set to a value lower than that, then not all moves may be explored. On the other hand, if the threshold T is set much higher than the maximum number of possible moves, then instead of relying on information gained from the MCTS tree, random simulations are performed, causing MCTS performance to be less than optimal.

- By demonstrating that the number of tree collapses being very much related to the algorithm's performance, we also explained that if the algorithm does not anticipate opponent's moves properly, its performance will drop.

Another interesting domain for studying the MCTS performance may be Checkers, where tablebases also exist. Future work can be associated with obtaining more experimental results in order to find out how domain independent our findings are.

Bibliography

[1] Multiple authors. “Webpage dedicated to Monte Carlo tree search”, accessible at :
http://www.mcts.ai

[2] Multiple authors. “Webage for MoGo computer program”, Accessible at:
http://www.lri.fr/~teytaud/mogo.html

[3] Cameron Browne, Edward Powley, Member, Daniel Whitehouse, Simon Lucas, Senior, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis and Simon Colton, “A Survey of Monte Carlo Tree Search Methods”, Transactions on Computational Intelligenceand AI in games, Vol. 4, No. 1, 2012

[4] Chaslot, Winands, Uiterwijk, van der Herik, Bouzy, “Progressive Strategies for Monte-Carlo Tree Search”, New Math. Nat. Comput., 4(3):343-357, 2008.

[5] G.M.J-B. Chaslot, “Monte-Carlo Tree Search” (PhD thesis), 2010

[6] Raghuram Ramanujan and Bart Selman, “Trade-Offs in Sampling-Based Adversarial Planning”, In Proc. 21st Int. Conf. Automat. Plan. Sched., pages 202-209, Freiburg, Germany, 2011.

[7] Article about endgame tablebases on wikipedia, article is accessible at:
http://en.wikipedia.org/wiki/Endgame_tablebase

[8] Matej Guid, Ivan Bratko, "An Experiment in Students Acquisition of Problem Solving Skill from Goal-Oriented Instructions", The Fourth International Conference on Advanced Cognitive Technologies and Applications, page 4, 2012

[9] Multiple authors, "Article about chess on wikipedia", article is accessible at:
`http://en.wikipedia.org/wiki/Chess`

[10] Remi Coulom, "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search", In Proc. 5th Int. Conf. Comput. and Games, pages 72-83, Turin, Italy, 2006.

[11] Chaslot, Bakkes, Szita and Pieter Spronck, "Monte-Carlo Tree Search: A new Framework for Game AI", In Proc. Artif. Intell. Interact. Digital Entert. Conf., pages 216-217, Stanford Univ., California, 2008.