Empirischer Vergleich von Alpha-Beta Suche und Playout Policy Adaptation Algorithmen anhand des Spiels Xiangqi

Bachelorarbeit

Jelena Moesus
468081

02. März 2026

Betreuer: Prof. Dr. Benjamin Blankertz
Dr.-Ing. Stefan Fricke

Technische Universität Berlin
Fakultät IV – Elektrotechnik und Informatik
Institut für Softwaretechnik und Theoretische Informatik
Fachgebiet Neurotechnologie

Zur Nutzung von Hilfsmitteln

Für diese Arbeit wurden kleinere Codeabschnitte (i.d.R. nur Hilfsfunktionen) von KI [1] generiert. Diese Abschnitte wurden im entsprechenden Code mit Kommentaren gekennzeichnet.

Andere Quellen und Inspirationen, sei es Bild oder Text, finden sich im Literaturverzeichnis und Fußnoten und sind an den entsprechenden Stellen verlinkt.

[1] Microsoft Copilot. Für die verwendete Online-Version liegt keine wohldefinierte Softwareversionsnummer vor, aber zur Zeit der Verwendung basierte sie auf GPT-5.1

Kurzfassung

Xiangqi (lit. Elefantenschach) existiert bereits seit dem 12. Jahrhundert. Es ist das bekannteste Spiel in China und auch in anderen ostasiatischen Ländern verbreitet. Entsprechend gibt es, ähnlich wie beim Schach, bereits viele Ansätze, das Spiel mithilfe von Computern zu lösen: von klassischen Suchalgorithmen mit und ohne domänenspezifischem Wissen bis hin zu Deep Reinforcement Learning.

Viele dieser Ansätze basieren auf der Alpha-Beta Suche. Bisher gibt es keinen direkten Vergleich von klassischer Alpha-Beta Suche zu statistischen Algorithmen mit Policy Adaptation. In dieser Arbeit wird der Datensatz ergänzt, indem die Alpha-Beta Suche und Playout Policy Adaptation Algorithmen durch direktes Spielen gegeneinander verglichen werden. Es werden außerdem die Playout Policy Adaptation Algorithmen durch Parametervariationen für das Spiel Xiangqi optimiert.

Es stellt sich heraus, dass im direkten Vergleich unter den Playout Policy Adaptation Algorithmen MAST am besten für das Spiel geeignet ist, allerdings keiner dieser Algorithmen eine Chance gegen ABS hat. Es bestätigt den Fakt, dass die meisten vorhandenen Maschinen für das Spiel auf Alpha-Beta Suche basieren.

Inhaltsverzeichnis

Abbildungsverzeichnis i

Tabellenverzeichnis ii

Abkürzungsverzeichnis iv

1. Einleitung 1
1.1. Grundproblem und Relevanz . . . 1
1.2. Xiangqi . . . 1
1.2.1. Unentschieden . . . 3
1.2.2. Strategien . . . 4
1.3. Literatur . . . 4
1.4. Ziel und Struktur . . . 5

2. Methoden 7
2.1. Alpha-Beta Suche (ABS) . . . 7
2.2. Monte Carlo Tree Search (MCTS) . . . 11
2.3. Upper Confidence bounds applied to Trees (UCT) . . . 13
2.4. Playout Policy Adaptation Algorithmen . . . 14
2.4.1. Nested Rollout Policy Adaptation (NRPA) . . . 15
2.4.2. Playout Policy Adaptation (PPA) . . . 17
2.4.3. Playout Policy Adaptation with Penalty (PPAP) . . . 18
2.4.4. Playout Policy Adaptation with move Features (PPAF) . . . 19
2.4.5. Move Average Sampling Technique (MAST) . . . 19
2.5. Vergleich . . . 20
2.5.1. Gewinnprozentsatz . . . 20
2.5.2. Elo-Zahl . . . 20
2.5.3. Likelihood of Superiority (LOS) . . . 21
2.6. Programmiersprache . . . 22

3. Implementierung 23
3.1. Spieldarstellung . . . 23
3.2. Bitboards . . . 23
3.2.1. Speicher . . . 24
3.2.2. Leistung . . . 24
3.3. Zuggenerator . . . 24
3.4. PPA-Algorithmen . . . 26

4. Ergebnisse **28**
4.1. PPA vs. PPAP . . . 29
4.2. PPAF vs. PPAFP . . . 30
4.3. α-Variation . . . 31
4.4. k-Variation . . . 33
4.5. c-Variation . . . 36
4.6. Zeit pro Zug Variation . . . 37
4.7. Feste Playoutzahl . . . 41
4.8. PPA-Algorithmen vs. ABS . . . 43

5. Diskussion **46**
5.1. PPA vs. PPAP . . . 46
5.2. PPAF vs. PPAFP . . . 47
5.3. Playoutzahl . . . 48
5.4. α-Variation . . . 49
5.5. k-Variation . . . 50
5.6. Zuglimit . . . 50
5.7. c-Variation . . . 51
5.8. Zeit pro Zug Variation . . . 52
5.9. Feste Playoutzahl . . . 53
5.10. PPA-Algorithmen vs. ABS . . . 55
5.11. Limitationen . . . 56

6. Fazit **58**

Literaturverzeichnis **59**

Anhang **61**

A. PPAF vs. PPAFP **62**

B. α-Variation **63**

C. k-Variation **64**

D. c-Variation **65**

E. Zeit pro Zug Variation **69**

F. Feste Playoutzahl **71**

G. PPA-Algorithmen vs. ABS **72**

Abbildungsverzeichnis

1.1. Startboard Xiangqi . . . 3

2.1. Allgemeines Beispiel Alpha-Beta Intervalle . . . 8
2.2. Die vier Phasen der Monte Carlo Tree Search . . . 11

4.1. PPAP vs. PPA mit variierendem p . . . 29
4.2. PPAFP vs. PPAF mit variierendem p . . . 31
4.3. PPAP u. PPAFP vs. UCT mit variierendem α . . . 32
4.4. MAST vs. UCT mit variierendem k . . . 34
4.5. MAST vs. UCT mit variierendem k (zusätzliche Messungen) . . . 35
4.6. PPAP, PPAFP u. MAST vs. UCT mit variierendem c (zusätzliche Messungen) . 36
4.7. PPAP, PPAFP u. MAST vs. UCT mit variierender Zeit pro Zug . . . 37
4.8. PPAP, PPAFP u. MAST vs. UCT mit variierender Zeit pro Zug (zusätzliche Messungen) . . . 39
4.9. UCT vs. PPAP, PPAFP u. MAST mit 20000 Playouts . . . 41
4.10. MAST vs. PPAP u. PPAFP mit 20000 Playouts . . . 42
4.11. PPAP vs. PPAFP mit 20000 Playouts . . . 43
4.12. ABS vs. PPAP, PPAFP, MAST u. UCT mit variierender Zeit pro Zug . . . 44

D.1. PPAP, PPAFP u. MAST vs. UCT mit variierendem c . . . 65

Tabellenverzeichnis

4.1. PPA vs. PPAP mit variierendem p 30
4.2. PPAF vs. PPAFP mit variierendem p (gekürzt) 30
4.3. UCT vs. PPAP mit variierendem α (gekürzt) 33
4.4. UCT vs. PPAFP mit variierendem α (gekürzt) 33
4.5. UCT vs. MAST mit variierendem k (gekürzt) 36
4.6. UCT vs. MAST mit variierendem k (zusätzliche Messungen, gekürzt) 36
4.7. UCT vs. PPAP mit variierender Zeit pro Zug (gekürzt) 38
4.8. UCT vs. PPAFP mit variierender Zeit pro Zug (gekürzt) 38
4.9. UCT vs. MAST mit variierender Zeit pro Zug (gekürzt) 38
4.10. UCT vs. PPAP mit variierender Zeit pro Zug (zusätzliche Messungen, gekürzt) . 40
4.11. UCT vs. PPAFP mit variierender Zeit pro Zug (zusätzliche Messungen, gekürzt) 40
4.12. UCT vs. MAST mit variierender Zeit pro Zug (zusätzliche Messungen, gekürzt) 40
4.13. ABS vs. PPAP, PPAFP, MAST u. UCT mit variierender Zeit pro Zug (gekürzt) 45

A.1. PPAF vs. PPAFP mit variierendem p 62

B.1. UCT vs. PPAP mit variierendem α 63
B.2. UCT vs. PPAFP mit variierendem α 63

C.1. UCT vs. MAST mit variierendem k 64
C.2. UCT vs. MAST mit variierendem k (zusätzliche Messungen) 64

D.1. UCT vs. PPAP mir variierendem c 66
D.2. UCT vs. PPAFP mir variierendem c 66
D.3. UCT vs. MAST mir variierendem c 67
D.4. UCT vs. PPAP mit variierendem c (zusätzliche Messungen) 67
D.5. UCT vs. PPAFP mir variierendem c (zusätzliche Messungen) 68
D.6. UCT vs. MAST mir variierendem c (zusätzliche Messungen) 68

E.1. UCT vs. PPAP mit variierender Zeit pro Zug 69
E.2. UCT vs. PPAFP mit variierender Zeit pro Zug 69
E.3. UCT vs. MAST mit variierender Zeit pro Zug 69
E.4. UCT vs. PPAP mit variierender Zeit pro Zug (zusätzliche Messungen) 70
E.5. UCT vs. PPAFP mit variierender Zeit pro Zug (zusätzliche Messungen) 70
E.6. UCT vs. MAST mit variierender Zeit pro Zug (zusätzliche Messungen) 70

F.1. UCT vs. PPAP, PPAFP u. MAST mit 20000 Playouts 71
F.2. MAST vs. PPAP u. PPAFP mit 20000 Playouts 71
F.3. PPAP vs. PPAFP mit rund 20000 Playouts 71

G.1. ABS vs. PPAP, PPAFP, MAST u. UCT mit variierender Zeit pro Zug 72
G.2. ABS vs. PPAP, PPAFP, MAST u. UCT mit variierender Zeit pro Zug (biased) . 72

Abkürzungsverzeichnis

ABS Alpha-Beta Suche

LOS Likelihood of Superiority

MAST Most Average Sampling Technique

MASTF Most Average Sampling Technique with move Features

MCTS Monte Carlo Tree Search

NRPA Nested Rollout Policy Adaptation

PPA Playout Policy Adaptation

PPAF Playout Policy Adaptation with move Features

PPAFP Playout Policy Adaptation with move Features and Penalty

PPAP Playout Policy Adaptation with Penalty

UCT Upper Confidence bounds for Trees

1. Einleitung

1.1. Grundproblem und Relevanz

Seit einiger Zeit sind Zugfindungsalgorithmen für Spiele ein Thema in der Wissenschaft, zu dem ganzen Konferenzen abgehalten und unzählige Artikel verfasst werden. Seit Neuerem sind sie außerdem Spielplatz für KI-Implementationen. Häufig betrachtet werden dabei komplexere Spiele wie Schach, aber auch dessen Verwandte. In dieser Arbeit sollen Zugfindungsalgorithmen für das Chinesische Schach – Xiangqi – untersucht und verglichen werden.

Für das Spielen von Xiangqi existieren bereits einige algorithmische Modelle, wobei viele von ihnen auf der Alpha-Beta Suche (ABS) basieren[15]. Im Folgenden wird untersucht, ob Playout Policy Adaptation Algorithmen, also Erweiterungen der Monte Carlo Tree Search (MCTS) mit Upper Confidence bounds for Trees (UCT), mit einer Alpha-Beta Implementierung mithalten oder diese sogar in Spielstärke übertreffen können.

Die PPA-Algorithmen haben gegenüber der ABS den Vorteil, dass sie selbstständig während der Zugsuche lernen und keine Notwendigkeit für eine Bewertungsfunktion besteht. Bewertungsfunktionen benötigen Domänenwissen oder komplexe Trainingsalgorithmen, sind aber für die ABS unverzichtbar.

Ein direkter Vergleich von ABS und PPA-Algorithmen anhand des Spiels Xiangqi wurde noch nie durchgeführt. Während PPA-Algorithmen in manchen Versuchen eine deutliche Verbesserung der Spielstärke zu UCT zeigen[4], werden sie nun direkt der ABS gegenübergestellt.

1.2. Xiangqi

Xiangqi, auch bekannt unter den Namen Chinesisches Schach, Generalschach oder Elefantenschach[1], wird bereits seit dem 12. Jahrhundert vor allem in Ostasien gespielt. Mit etwa einer Milliarde SpielerInnen allein in China ist es weltweit eines der bekanntesten Spiele[16].

Bei Xiangqi handelt es sich um ein Zwei-Personen-Spiel mit perfekter Information. Ähnlich wie beim Westlichen Schach stehen sich die jeweils sechzehn Figuren der zwei Teams (Armeen) anfangs gegenüber und das Ziel ist, den gegnerischen König (bzw. General) zu schlagen[16]. Bei dem Spielfeld und vielen Figuren gibt es jedoch bedeutende Unterschiede zum Westlichen Schach.

[1] https://en.wikipedia.org/wiki/Xiangqi. Letzter Zugriff: 03.12.2025

Wie in Go werden die Figuren in Xiangqi auf den kreuzenden Linien des 9x10 Spielfelds (Abb. 1.1) platziert, dementsprechend gibt es keine schwarzen und weißen Felder. Horizontal durch die Mitte des Felds verläuft ein Fluss (河 *hé*), welcher beide SpielerInnen zunächst trennt. Dieser ist relevant für die Zugmöglichkeiten zweier Figurenarten. Zudem gibt es am oberen und unteren Ende des Spielfelds je einen 3x3 Bereich an Feldern, welcher als Palast (宮 *gōng*) bezeichnet wird. Diesen können sowohl der König, als auch eine weitere Figurenart nicht verlassen.[2]

Die Abbildung 1.1 zeigt die Startstellung von Xiangqi mit allen Figuren. Der Fluss wird hier durch fehlende vertikale Linien gekennzeichnet, der Palast durch diagonale Verbindungen der zugehörigen Felder. Die Zuordnung der chinesischen Zeichen zu ihren deutschen Namen findet sich in der Aufzählung der Spielfiguren.

Die klassischen Spielfiguren sind runde Steine, mit je einem chinesischen Zeichen – der Name der Figur. Dabei unterscheiden sich die Schriftzeichen für schwarz und rot leicht; sie sind entweder Homophone oder haben eine ähnliche Bedeutung. Das ist wahrscheinlich auf eine Zeit zurückzuführen, in welcher die Spielsteine keine verschiedenen Farben hatten, aber die zwei Teams trotzdem auseinander gehalten werden mussten.[3]

In modernen Turnieren beginnt Rot mit dem ersten Zug[4].

Die verschiedenen Spielfiguren[5] sind:

- (schwarz: 將 *jiàng*/rot: 帥 *shuài*) **König:** Der König darf horizontal oder vertikal genau ein Feld laufen, solange sich dieses im Palast befindet. Eine Spezialregel erlaubt ihm, wenn er dem anderen König direkt und ohne Hindernisse gegenübersteht, über das gesamte Feld zu springen und diesen zu schlagen („flying King“ Zug).
- (士 *shì*/仕 *shì*) **Berater:** Der Berater darf diagonal genau ein Feld laufen, solange sich dieses im Palast befindet.
- (象 *xiàng*/相 *xiàng*) **Elefant:** Der Elefant darf genau zwei Felder diagonal laufen, solange sich das Zielfeld in der eigenen Spielfeldhälfte (getrennt durch den Fluss) befindet. Er darf dabei keine anderen Figuren überspringen.
- (馬 *mǎ*/傌 *mǎ*) **Pferd:** Das Pferd bewegt sich wie im Westlichen Schach erst zwei Felder geradeaus und dann eines zur Seite. Allerdings kann es nicht über andere Figuren springen, die direkt neben ihm stehen.
- (車 *jū*/俥 *jū*) **Streitwagen:** Der Streitwagen darf sich wie der Turm im Westlichen Schach beliebig viele Felder horizontal oder vertikal in eine Richtung bewegen, bis er einen Gegner schlagen kann oder von einer freundlichen Figur geblockt wird.
- (砲 *pào*/炮 *pào*) **Kanone:** Die Kanone darf sich beliebig viele horizontale oder vertikale Felder in eine Richtung auf ein freies Feld bewegen, bis sie von einer beliebigen Figur geblockt wird. Schlagen kann sie nur, wenn sie vorher über genau eine andere beliebige Figur gesprungen ist. Dabei ist der Abstand zwischen dem geschlagenen Gegner und der

[2] https://en.wikipedia.org/wiki/Xiangqi. Letzter Zugriff: 03.12.2025
[3] https://en.wikipedia.org/wiki/Xiangqi. Letzter Zugriff: 03.12.2025
[4] https://en.wikipedia.org/wiki/Xiangqi. Letzter Zugriff: 03.12.2025
[5] https://en.wikipedia.org/wiki/Xiangqi. Letzter Zugriff: 03.12.2025

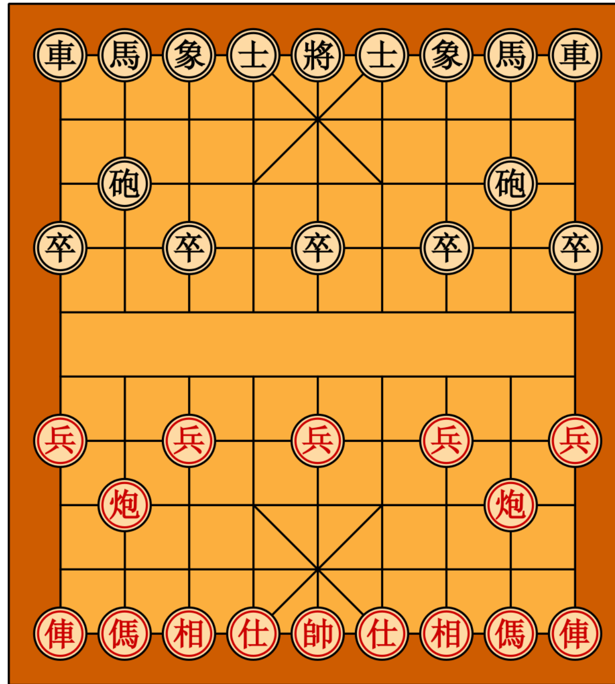

Abbildung 1.1.: Startboard Xiangqi[6]

Figur, die als Schanze fungiert, egal. Sie kann nicht über eine andere Figur auf ein freies Feld springen.

- (卒 *zú*/兵 *bīng*) **Soldat:** Der Soldat kann sich genau ein Feld vertikal vorwärts bewegen. Anders als im Westlichen Schach darf er auch nur auf Feldern schlagen, welche er betreten kann. Sobald ein Soldat den Fluss überquert und die gegnerische Spielfeldseite betritt, darf er alternativ genau ein Feld horizontal laufen. Wenn er den Spielfeldrand erreicht, kann er nicht in eine andere Figur eingetauscht werden.

Mit seinen verschiedenen Figuren und dem großen Spielfeld wurde Xiangqi in der Komplexität zwischen dem Westlichen Schach und dem komplexeren Go eingeordnet[16]. Der Suchbaum spannt sich dabei breit auf, was es rechnerisch unzumutbar macht, alle möglichen Spielfolgen zu durchsuchen.

1.2.1. Unentschieden

Unentschieden in Xiangqi können durch Zugwiederholungen auftreten. Die Regeln zu Wiederholungen unterscheiden sich in China, Taiwan und Hong Kong teils stark und sind sehr vielfältig. Hier werden die Regeln aufgeführt, wie sie in [16] (Appendix A) beschrieben worden sind.

Simplifiziert handelt es sich dann um eine Wiederholung, wenn ein Bordzustand zum dritten Mal auftritt.

[6]Quelle: https://commons.wikimedia.org/wiki/File:Xiangqi_Board.svg. Letzter Zugriff: 03.12.2025

In zwei Fällen wird das Spiel daraufhin mit einem Unentschieden beendet: Entweder wenn beide SpielerInnen keine Schlagzüge ausführen (können) oder wenn beide SpielerInnen Züge ausführen, die die jeweils andere SpielerIn zu einem bestimmten Zug zwingen (bspw. mit einem Check auf den König).

Verstößt nur eine der SpielerInnen gegen die Regeln zur Wiederholung, so hat diese verloren.

Eine SpielerIn kann außerdem die SchiedsrichterIn fragen, das Spiel mit einem Unentschieden zu beenden, wenn keine Figur geschlagen wurde und die GegnerIn in den letzten vierzig Zügen nicht mehr als zehn Checks ausgeführt hat. Diese Regel betrifft allerdings nur Turniersituationen, in welchen eine SchiedsrichterIn anwesend ist.

1.2.2. Strategien

Generell wird der Wert der Figuren (in absteigender Reihenfolge) wie folgt gewertet: König, Streitwagen, Kanone, Pferd, Elefant/Berater, Soldat[16]. Es empfiehlt sich in den meisten Fällen, keine Figur niedrigen Wertes gegen eine höherwertige zu tauschen. Dennoch kann auch ein Soldat in der richtigen Position sehr wertvoll sein[16].

Es ist außerdem zu beachten, dass mit Figuren wie Elefanten oder Beratern das Spiel nicht gewonnen werden kann, da sie nicht die gegnerische Spielfeldhälfte betreten dürfen und der König den eigenen Palast nicht verlassen kann. Daher muss darauf geachtet werden, dass auch Figuren anderer Art im Spiel bleiben.

Mögliche Strategien wie Fork, Pin oder Skewer[7] gleichen dem Westlichen Schach. Solche Spielspezifischen Taktiken können in Bewertungsfunktionen z.B. für ABS mit einbezogen werden. Für diese Arbeit lag der Fokus jedoch auf den Spiel-unabhängigen Aspekten der UCT-basierten Algorithmen.

1.3. Literatur

Eine gute Übersicht über das Spiel Xiangqi gibt es nicht nur auf Wikipedia und der Xiangqi Website, sondern auch in *Computer chinese chess*[16] von Yen et. al. Dort wird kurz die Spielhistorie aufgegriffen, dann die Regeln erklärt und die damalige Computerspielstärke im Vergleich mit menschlichen SpielerInnen dargestellt. Zusätzlich werden auch Techniken aus Xiangqiprogrammen erklärt und unter anderem Werte für die verschiedenen Spielfiguren aufgeführt, sowie eine Übersicht für einen Teil der Figurenarten geliefert, welche Felder des Spielfeldes welchen strategischen Wert besitzen. Auf diesem Domänenwissen basiert die Bewertungsfunktion der ABS für diese Arbeit.

Der Ausgangsalgorithmus aller in dieser Arbeit verwendeten Playout Policy Adaptation (PPA) Algorithmen ist die Monte Carlo Tree Search (MCTS). Sie wird in dem Artikel *Monte Carlo Tree Search: a review of recent modifications and applications*[19] von Świechowski et. al.

[7] https://www.xiangqi.com/. Letzter Zugriff: 03.12.2025

beschrieben. Die MCTS benötigt keine Heuristik, sondern spielt pro Zugsuche das Spiel zehntausende bis Millionen Male von der aktuellen Position bis zum Ende, mit zufälligen Zügen. Aus Spielausgängen wird eine Statistik erstellt, mit welcher der beste Zug ermittelt wird. Der Artikel beschreibt die Phasen von MCTS und ihren Einfluss auf Spielealgorithmen und KI im Allgemeinen, sowie die Spezifizierung UCT für MCTS.

Der Algorithmus Nested Rollout Policy Adaptation (NRPA) ist für Ein-Personen-Spiele konzipiert und wird in *Nested Rollout Policy Adaptation for Monte Carlo Tree Search*[13] von Christopher Rosin erstmals vorgestellt. In dem Artikel wird unter anderem ein Vergleich von MCTS und NRPA anhand verschiedener Spiele vorgenommen. In NRPA werden in Zugsuchen Spiele nicht mehr komplett zufällig gespielt, sondern von einer selbstständig über die Suche gelernten Policy gelenkt. Dieses Prinzip ist grundlegend für PPA-Algorithmen.

Für die PPA-Algorithmen selbst dient Cazenaves Artikel *Playout policy adaptation with move features*[4] als Grundlage. Die hier vorgestellten Algorithmen kombinieren das Konzept der MCTS mit Spieldurchläufen (Playouts) zur Bildung von Statistiken, mit NRPA mit Policy geleiteten Playouts. Der Artikel beschreibt die Algorithmen Playout Policy Adaptation (PPA), Playout Policy Adaptation with move Features (PPAF), Most Average Sampling Technique (MAST) und Most Average Sampling Technique with move Features (MASTF) und stellt sie in den Vergleich mit MCTS, bzw. UCT. Für den Vergleich werden verschiedene Zwei-Personen-Spiele herangezogen, unter anderem Akari und Knightthrough. Anstatt die Algorithmen mit einem gleichen Zeitbudget zu vergleichen, wird in diesem Artikel eine bestimmte Anzahl Playouts pro Zugsuche festgelegt.

Weiterführende Techniken zu PPA-Algorithmen finden sich in *Memorizing the Playout Policy*[5] von Cazenave und Diemert. Der Artikel schlägt vor, PPAF „warm“ zu starten, sprich jede Zugsuche mit in vorherigen Suchen gesammeltem Vorwissen zu beginnen.

Ein alternativer Ansatz zur algorithmischen Lösung von Xiangqi findet sich in *Deep Reinforcement Learning Xiangqi Player with Monte Carlo Tree Search*[17] von Yilmaz et. al. Dort wird ein KI-Model vorgestellt, welches mit Deep Reinforcement Learning und Monte Carlo Tree Search für bessere Entscheidungsfindung versucht, eine Alpha-Zero ähnliche Spiele-KI nachzubilden. Diese soll besser mit Xiangqis hohem Branchingfaktor umgehen können, als z.B. Standard-MCTS das kann, und durch Spielen gegen sich selbst Strategien eigenständig verbessern.

Ganz ohne eine Suche kommt die Spiele-KI in *Mastering Chinese Chess AI(Xiangqi) Without Search*[10] von Lin et. al. aus. Die Autoren eliminieren in ihrer Implementierung die Zeit- und Ressourcen-intensive Monte Carlo Tree Search und erstellen einen Deep Reinforcement Learning Algorithmus, welcher erst anhand von Expertenspielen und dann durch Spielen gegen sich selbst Xiangqi erlernt. Der resultierende Algorithmus kann mit den Top 0,1% der menschlichen Xiangqi-SpielerInnen mithalten.

1.4. Ziel und Struktur

In dieser Arbeit soll eine klassische Alpha-Beta Implementierung mit Transposition Table und Iterative Deepening mit verschiedenen Playout Policy Adaptation Algorithmen verglichen wer-

den. Die PPA-Algorithmen werden vorher gegeneinander, bzw. gegen UCT getestet und die einstellbaren Parameter für Xiangqi möglichst optimiert. Nur die besten Parameterkombinationen spielen gegen ABS. Letztendlich soll sich zeigen, ob ein UCT-basierter Algorithmus in der Lage ist, ABS in Xiangqi zu schlagen und damit Domänenwissen optional zu machen. Gleichzeitig ergänzt der Vergleich von ABS und PPA-Algorithmen den bestehenden Datensatz.

In Kapitel 2 findet sich eine Erklärung der verwendeten Methoden. Hier werden ABS, UCT und die PPA-Algorithmen im Fließtext und anhand von Pseudocode beschrieben. Zudem werden die zu vergleichenden Metriken vorgestellt und die Wahl der Programmiersprache begründet.

Details zur Implementierung werden in Kapitel 3 besprochen. Es werden zudem einige Programmierentscheidungen begründet.

Kapitel 4 stellt die Ergebnisse der Versuche vor, also speziell das Abschneiden der Algorithmen im direkten Vergleich zueinander. Vollständige Daten zu allen Versuchen finden sich außerdem im Anhang.

In Kapitel 5 werden die gefundenen Ergebnisse interpretiert und soweit möglich ein Vergleich mit bestehender Literatur geführt. Es wird außerdem auf Limitationen der Arbeit eingegangen.

Eine Zusammenfassung der Ergebnisse und ein Ausblick auf mögliche weitere Forschung findet sich in Kapitel 6.

2. Methoden

In diesem Kapitel werden die für diese Arbeit relevanten Algorithmen vorgestellt, sowie die Metriken, anhand welcher sie zu vergleichen sind. Details zur Implementierung finden sich im folgenden Kapitel.

2.1. Alpha-Beta Suche (ABS)

Bei der ABS handelt es sich um eine Optimierung des Minimax-Algorithmus. Während Minimax alle Züge bis zu der bestimmten Tiefe durchsuchen muss, führt ABS sogenannte „Cutoffs“ durch[8][14].

Der Suchbaum für Minimax wächst exponentiell mit der Suchtiefe. ABS schwächt diesen Effekt, aber eliminiert ihn nicht[14].

Die namensgebenden Variablen Alpha und Beta werden rekursiv weitergegeben[14] und stellen jeweils den minimalen Score der maximierenden SpielerIn und den maximalen Score der minimierenden SpielerIn dar. Simpler gesagt handelt es sich um Werte, die jeweils den SpielerInnen mindestens als Ausgang garantiert sind. Eine SpielerIn versucht, den zurückgegebenen Wert zu minimieren, die andere zu maximieren. Dabei zeigt ein sehr hoher Score den Gewinn der maximierenden SpielerIn an und ein sehr niedriger Score den Gewinn der minimierenden SpielerIn.[8][1]

Wurde schon ein guter Zug gefunden, kann davon ausgegangen werden, dass die SpielerIn mit Zugrecht keinen schlechteren nehmen wird – eine weitere Erkundung des Suchbaums muss hier also nicht mehr stattfinden. Es wird ein Cutoff durchgeführt. So kann ABS eine bedeutende Reduzierung zu durchsuchender Zweige ermöglichen, ohne die Gefahr, bessere Züge zu übersehen.[8]

Jeder Knoten im Suchbaum entspricht einem Spielstand. Eine Bewertungsfunktion wird benötigt, die jedem beliebigen Spielstand einen Wert zuweisen kann. Für solch eine Funktion ist entweder Domänenwissen nötig oder sie muss mit maschinellem Lernen erstellt werden[15]. Je besser die Bewertungsfunktion, desto besser funktioniert ABS auch schon bei niedrigeren Suchtiefen[1].

Es werde die Abbildung 2.1 betrachtet. In Knoten auf Höhe von A hat A Zugrecht, in Knoten auf Höhe von B ist B die aktuelle SpielerIn. A maximiert, B minimiert. α stellt die untere Schranke dar, β die obere. In der Standardvariante von ABS werden beide Parameter mit jeweils minus Unendlich und Unendlich initialisiert. In einer praktischen Implementierung genügen Werte, die

[1] https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning. Letzter Zugriff: 09.12.2025

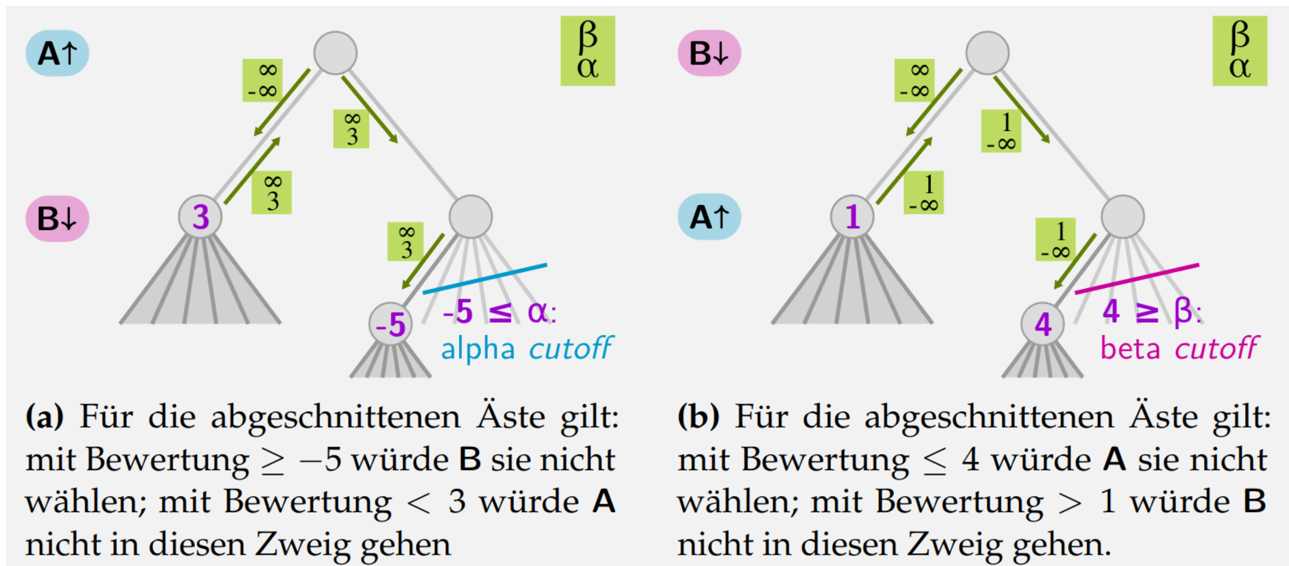

Abbildung 2.1.: Allgemeines Beispiel Alpha-Beta Intervalle (Quelle: [1])

garantiert unter der niedrigst möglichen Bewertung oder über der höchst möglichen Bewertung liegen.

In Abbildung 2.1 (a) wurde nun schon der erste Unterbaum komplett bis zur gegebenen Tiefe durchsucht und für B der beste Wert 3 gefunden. Dies wird als neue untere Schranke gesetzt. Im nächsten Unterbaum wird ein Wert von -5 als mögliche Option für B gefunden. Da B minimieren will, würde aus dem Elternknoten ein Zug mit einer Bewertung von kleiner gleich -5 gewählt werden. Das liegt aber bereits unter der unteren Schranke, dh. A kann garantiert einen besseren Wert erreichen, wenn es den ersten Zug wählt. Der Rest des rechten Unterbaums muss also nicht durchsucht werden; das Ergebnis für A ist garantiert schlechter als die bereits gefundene Option. Hierbei handelt es sich um ein α-Cutoff.

Genauso funktioniert es in die andere Richtung. In Abbildung 2.1 (b) ist B bereits der Wert 1 aus dem linken Unterbaum garantiert. A würde aber in dem rechten Unterbaum einen Zug mit einer Bewertung von 4 oder höher wählen. Wieder muss der Rest des Zweigs nicht durchsucht werden; es liegt ein β-Cutoff vor.

Dieser Algorithmus geht immer davon aus, dass beide SpielerInnen optimal spielen. Andernfalls kann das Ergebnis günstiger als die gefundene Schranke ausfallen[1].

ABS übersieht im durchsuchten Teil des Baums keine guten oder gefährlichen Züge, es kann allerdings der Horizonteffekt auftreten: ABS stoppt die Suche bei einer bestimmten Tiefe. Ist bei dieser das Spiel noch nicht in allen Unterbäumen beendet, gibt es nicht untersuchte Spielstände. In Spielen, wo sich die Bewertung eines Spielstands schnell stark ändern kann (z.B. durch Schlagzüge), kann es vorkommen, dass die Suche direkt vor solch einer Schwankung beendet wird, bzw. solche Züge ausführt, die die Situation über den Suchhorizont hinausschieben. Aus Sicht dieser Suche ist das Problem damit gelöst. Es kann nicht erkannt werden, ob die Situation unausweichlich ist. Im Zweifelsfall wird ein Zug dadurch besser gewertet, als gerecht-

fertigt gewesen wäre, oder es werden schwache Züge ausgeführt, um die ungewollte Situation hinauszuzögern.[2]

Erweiterungen

Es gibt einige Methoden, die ABS weiter zu optimieren. Die in dieser Arbeit verwendeten werden hier aufgeführt. Es handelt sich allerdings nur um einen Bruchteil der Möglichkeiten.

Iterative Deepening[9]

Der Rekursionsanker der ABS ist die in die Funktion gegebene Tiefe. Ist diese Null, wird die Bordposition ausgewertet und die Bewertung zurückgegeben. Beim Iterative Deepening wird ABS mehrmals aufgerufen. Zuerst mit der Tiefe 1, dann mit der Tiefe 2, etc., bis entweder eine Zieltiefe erreicht wurde oder eine gegebene Zeit abgelaufen ist.

Die benötigte Zeit für eine Tiefe hängt unter anderem von dem Branchingfaktor des Suchbaums und der Zahl der Cutoffs ab, kann also nicht im Vorhinein genau bekannt sein. Soll der Algorithmus also in einer bestimmten Zeit, anstatt mit einer bestimmten Tiefe arbeiten (z.B. um einen möglichst fairen Vergleich zu anderen Algorithmen zu ermöglichen), kann Iterative Deepening verwendet werden.

Zurückgegeben wird entweder der Zug, den die tiefste vollendete Suche gefunden hat, oder ein Zwischenergebnis aus der angefangenen Suche, sollte dieses einen besseren Wert haben als der Zug aus der vorherigen Suche. In dieser Arbeit werden nur Ergebnisse von vollendeten Suchen zurückgegeben, aber beide Implementierungen sind möglich.

Iterative Deepening kann außerdem effektiv für die Zugsortierung eingesetzt werden. Der in der vorherigen Suche gefundene beste Zug hat bereits einen guten Wert. Dieser Zug kann in der Suche mit der nächst höheren Tiefe als erstes untersucht werden und potentiell zu vielen Cutoffs führen.

Zugsortierung

Die Zugsortierung ist relevant für ABS, da von ihr die Anzahl der möglichen Cutoffs abhängt, und damit die Zeitersparnis von ABS gegenüber dem Minimax-Algorithmus[14].

Minimax muss alle Knoten des Suchbaums bis zur Tiefe t absuchen. Wird von einem festen Branchingfaktor b ausgegangen, so müssen b^t Knoten untersucht werden[8]. Bei maximal ungünstiger Zugsortierung verfällt ABS zu einer Minimax-Suche.

Eine solche schlechte Zugsortierung liegt dann vor, wenn alle neu gefundenen Werte innerhalb der Schranken α und β liegen und so nie ein Cutoff durchgeführt werden kann. Bei einer guten Sortierung dagegen wird der Zug mit dem besten Wert zuerst untersucht und die Schranken können schnell nah aneinander gelegt werden. Je näher die Schranken beieinander liegen, desto

[2] https://www.chessprogramming.org/Horizon_Effect. Letzter Zugriff: 09.12.2025

mehr Cutoffs können durchgeführt werden und desto mehr Zeit wird eingespart. Dabei enthält der minimale Suchbaum $b^{\lceil \frac{t}{2} \rceil} + b^{\lfloor \frac{t}{2} \rfloor} - 1$ Knoten.[8]

Mögliche Strategien für die Zugsortierung sind, abgesehen von der bereits erwähnten Methode mit Iterative Deepening, bspw. nach dem Wert der Spielsteine, Schlagzüge zuerst oder nach den zuvor gefundenen Werten in der Transposition Table.

Da ABS auch ohne elaboriertere Verfahren den PPA-Algorithmen überlegen ist, wurde in dieser Arbeit nur eine sehr einfache Zugsortierung umgesetzt: Die Züge sind nach dem Wert einer Spielfigur nach [16] geordnet, mit den Zügen des Königs zuerst und denen der Soldaten als letztes. Es wird angenommen, dass die Züge von höherwertigen Figuren tendenziell die Bewertung der Spielposition am stärksten beeinflussen.

Transposition Table

Die Transposition Table ist eine Tabelle, in welcher Informationen über Spielpositionen indexiert von Hashwerten gespeichert werden. Sie dient dazu, Permutationen in der Zugreihenfolge auszunutzen. Wurde eine Spielposition über eine andere Zugfolge bereits erreicht und untersucht (oder in Iterative Deepening in einer vorherigen Suche erreicht), kann sie über den Hashwert des Spielbretts in der Tabelle gefunden werden. Das kann eine weitere Durchsuchung aus dieser Position ersparen, wenn der gefundene Tabelleneintrag als gut genug gewertet wird. Das ist in dieser Arbeit der Fall, wenn die gespeicherte Tiefe größer gleich der ist, mit welcher die aktuelle Suche von dieser Position ausgeführt werden sollte. Da eine tiefere Suche mindestens ein gleich gutes und möglicherweise ein besseres Ergebnis liefert, muss die Position neu untersucht werden, wenn die Tiefe des gespeicherten Eintrags zu gering ist.[2]

Die Informationen, die ein Tabelleneintrag enthält, sind theoretisch frei wählbar. In dieser Arbeit wurden folgende Werte gespeichert:

is_set: Ein Boolwert der anzeigt, ob ein Eintrag aktuell gültig ist. Zum „Löschen“ eines Eintrags muss diese Variable lediglich auf *false* gesetzt werden.

Hashwert: Der Hashwert wird beim Zobrist Hashing durch verschiedene logische Operationen erzeugt und sollte für jede Bordposition einzigartig sein[18]. Es handelt sich um einen unsigned 64-bit Integerwert. Dieser kann in einem Array nicht direkt als Tabellenindex verwendet werden, da 2^{64} Einträge extrem viel Speicher verbrauchen würden. Der Index wird demnach mithilfe des Hashwerts und einer Modulooperation ermittelt. Dadurch kann es sein, dass verschiedene Spielpositionen den selben Index haben. Um falsche Informationsrückgaben zu vermeiden, wird der Hashwert im Eintrag gespeichert und vor dem Auslesen abgeglichen.[2]

Zug u. Wert: Der beste für diese Position gefundene Zug und der erreichte Wert. Für das sofortige Zurückgeben für den Fall, dass es sich bei dem Eintrag nicht nur um eine Schranke handelt.[2]

Schranken: Zwei Boolwerte, einer für die obere, und einer für die untere Schranke. Beim Lesen eines Eintrags wird überprüft, ob es sich bei dem gespeicherten Wert um eine Schranke handelt. Ist der Boolwert für die obere Schranke *true* wird das β der aktuellen Suche mit dem gespeicherten Wert angepasst. Liegt eine untere Schranke vor, wird α angepasst. Handelt es

sich weder um eine obere noch eine untere Schranke, enthält der Tabelleneintrag einen exakten Wert. Dieser und der gespeicherte Zug können sofort zurückgegeben werden; es findet keine weitere Suche in diesem Zweig statt.[2]

Tiefe: Dient der Bewertung der Güte des gespeicherten Tabelleneintrags[2].

2.2. Monte Carlo Tree Search (MCTS)[19]

Für das Verständnis der PPA-Algorithmen ist es wichtig, erst deren Grundlagen zu begreifen. Alle umgesetzten PPA-Algorithmen basieren auf MCTS mit UCT sowie NRPA. Sie unterscheiden sich lediglich in der Simulationsphase von MCTS mit UCT.

MCTS baut den Suchbaum prinzipiell anders auf, als ABS das tut. Bei ABS handelt es sich um eine Depth-First Suche, das bedeutet, dass in der Regel von links nach rechts der Suchbaum durchgegangen wird. Ein Zweig wird bis zu einer vorgegebenen Tiefe oder einem Endzustand durchlaufen, dann wird zu dem nächst höheren Knoten mit unerkundeten Kindern zurückgegangen und der links äußerste Zweig des nächsten Kindes bis zum Ende untersucht[9]. MCTS dagegen wird von einer Heuristik geleitet. Es wird immer der als am besten eingeschätzte Knoten als nächstes untersucht (Best-First Suche). Dadurch entsteht ein asymmetrischer Suchbaum, wie auch in Abbildung 2.2 zu sehen.[7] Die Heuristik wird in der folgenden Sektion 2.3 vorgestellt.

Die MCTS startet bei dem aktuellen Spielstand, dem Wurzelknoten im Suchbaum. Sie besteht aus vier Phasen, die in der Abbildung 2.2 zu sehen sind. Die Zahlen in den Knoten geben die Gewinne versus die Anzahl der Besuche des Knotens wieder. In weiß hinterlegten Knoten ist die aktuelle SpielerIn am Zug, in grau hinterlegten die GegnerIn. Die Zahl der gewonnen Spiele gilt dementsprechend für die SpielerIn, um deren Knoten es sich handelt. Im Playout besuchte Spielstände werden nicht im Baum gespeichert. Das Playout wird in der Abbildung im Simulationsschritt mit einem gewellten Pfeil dargestellt.

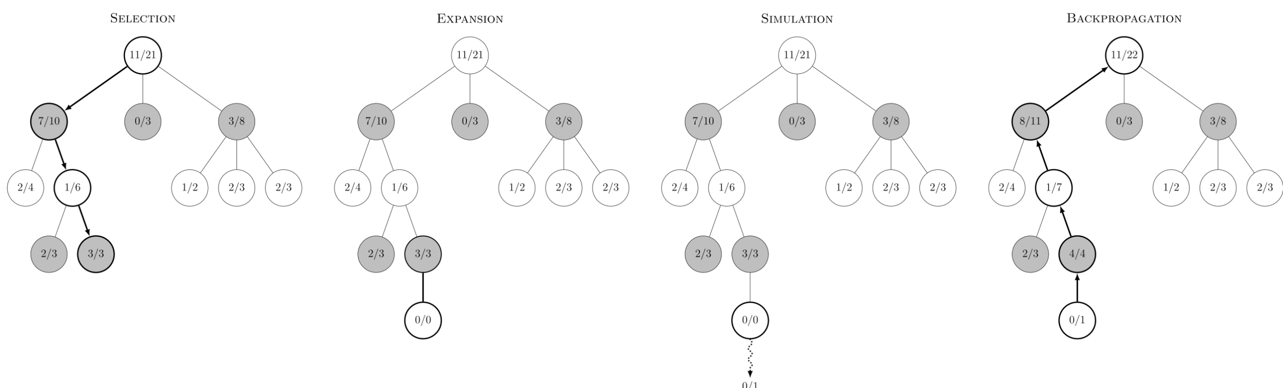

Abbildung 2.2.: Die vier Phasen der Monte Carlo Tree Search.[3]

[3]Quelle: Robert Moss. Letzter Zugriff: 28.01.2026

1. **Auswahl** (Selection): Ausgehend vom Wurzelknoten wird so lange jeweils ein Kindknoten nach einer festgelegten Auswahlstrategie ausgewählt, bis ein Blattknoten erreicht wird. In diesem Fall ist ein Knoten dann ein Blattknoten, wenn es sich entweder um einen finalen Spielstand handelt oder noch keines der Kinder erkundet wurde. Die Auswahlstrategie sollte hierbei eine gute Balance zwischen Erkundung neuer Züge und Ausnutzung bekannter guter Züge finden.
2. **Ausbau** (Expansion): Wenn es sich bei dem Knoten nicht um einen finalen Spielzustand handelt, wird in dieser Phase ein Kindknoten hinzugefügt. Sollte dabei ein finaler Zustand erreicht werden, wird der nächste Schritt übersprungen.
3. **Simulation** (Simulation): Von dem im vorherigen Schritt neu hinzugefügten Knoten aus wird ein Playout gestartet, bei welchem so lange zufällige Züge gespielt werden, bis ein Endzustand erreicht wird.
4. **Rückpropagierung** (Backpropagation): Läuft die Knoten von dem Endzustand bis zur Wurzel hinauf und passt in jedem die Gewinne/Besuche Rate an. Hierbei ist zu beachten, dass nach jedem Schritt der Wert der Gewinnvariable (Eins = Gewinn, Null = Verlust) invertiert werden muss.

Die vier Schritte werden so lange wiederholt, bis eine gegebene Zeit abgelaufen oder eine bestimmte Anzahl an Playouts erreicht ist. Dann kann der Zug mit den meisten Playouts für die aktuelle SpielerIn gewählt werden. Es wird nicht nach der besten Gewinnrate entschieden, da diese keine Information darüber gibt, wie oft der Knoten tatsächlich besucht wurde. Mit einer gut konfigurierten Auswahlstrategie dagegen werden gute Züge häufiger ausgeführt.

Algorithm 1 Grundlegende Struktur der MCTS[4]

1: **function** MCTS($board, player$)
2: **for** $i \leftarrow 0$ **to** maximum index of move **do**
3: $policy[i] \leftarrow 0.0$
4: $games[i] \leftarrow 0.0$
5: $wins[i] \leftarrow 0.0$
6: **end for**
7: **for** $i \leftarrow 0$ **to** number of playouts **do**
8: $b \leftarrow board$
9: $winner \leftarrow$ UCT($b, player, policy$)
10: $b1 \leftarrow board$
11: adapt($winner, b1, player, b.playout, policy$)
12: **end for**
13: **return** move with the most playouts
14: **end function**

In Algorithmus 1 ist die grundlegende Struktur von MCTS gezeigt, wobei die magentafarbenen Zeilen nur für PPA-Algorithmen relevant sind und die blauen Zeilen ausschließlich für MAST (Subsek. 2.4.5).

Der Suchbaum wird über die Länge eines Spiels gespeichert und mit jeder Zugsuche weiter ausgebaut.

2.3. Upper Confidence bounds applied to Trees (UCT)[19]

In der vorherigen Sektion wird die Anwendung einer Auswahlstrategie für die erste Phase der MCTS erwähnt. Die Standard-Auswahlstrategie für die meisten MCTS Implementationen heutzutage ist UCT. UCT ist eine Heuristik, die anhand einer Formel einen Wert für bereits erkundete Knoten errechnet. In der Auswahlphase von MCTS wird dann immer der Knoten mit dem höchsten errechneten Wert ausgewählt, bis ein Blattknoten erreicht wird. In manchen Artikeln wird der MCTS Algorithmus mit UCT kurz als UCT bezeichnet, wie z.B. in [4]. Das gilt auch für diese Arbeit.

Die Formel ist:

$$\text{Wert(Zug)} = \frac{w}{p} + c \cdot \sqrt{\frac{ln(T)}{p}}$$

mit w der verzeichneten Anzahl der gewonnen Playouts für den Zug, p wie oft der Zug gespielt wurde und T die Anzahl der Playouts insgesamt von diesem Zustand aus. c ist die sogenannte Explorationskonstante und wird oft bei $\sqrt{2}$ angesetzt, kann aber mit Experimenten im jeweiligen Anwendungszweck weiter angepasst werden.

In Algorithmus 2 entspricht diese Formel den Zeilen 10–14. Der grüne Code in Zeilen 13 und 14 zeigt, wie in dieser Arbeit Unentschieden behandelt werden. Unentschieden können hier nur durch das Erreichen der maximalen Playoutlänge oder durch häufige Zugwiederholungen entstehen. Sie geben also wenig Aufschluss über die relative Spielstärke der Algorithmen und sollen möglichst neutral in die Bewertung mit einfließen[4].

Der erste Summand der Gleichung ist die Gewinnrate des Zuges. Erfolgt die Bewertung (überwiegend) nach diesem Wert, handelt es sich um die Exploitation bekannter, guter Züge. Der zweite Summand stellt den Explorationswert dar. Je größer dieser ist, desto öfter werden neue Züge untersucht. Die Balance zwischen Exploitation und Exploration ist dementsprechend abhängig von c.

Damit bei bisher ununtersuchten Zügen ($p = 0$) kein Fehler auftritt, kann der Wert des Zuges für diesen Fall beispielsweise auf einen sehr großen Wert gesetzt werden. Das führt dazu, dass jeder Zug einmal ausprobiert wird, bevor solche mit gutem Wert bevorzugt werden.

UCT wird auch für die PPA-Algorithmen als Auswahlstrategie verwendet. Alle Vorkommnisse von `policy` in Algorithmus 2 sind nur für PPA-Algorithmen relevant. Die in Zeilen 6 und 25 erwähnte Transposition Table enthält den bisher erkundeten Suchgraphen. Sie wird in Sektion 3.4 näher erläutert.

[4]Eine ähnliche Implementierung findet sich hier. Letzter Zugriff: 24.02.2026

Algorithm 2 UCT für PPA[4]

```
1: function UCT(board, player, policy)            ▷ für Standard MCTS policy weglassen
2:     moves ← possible moves on board
3:     if board is terminal then
4:         return winner(board)
5:     end if
6:     t ← entry of board in the transposition table
7:     if t exists then
8:         bestValue ← −∞
9:         for each m in moves do
10:            T ← t.totalPlayouts
11:            w ← t.wins[m]
12:            p ← t.playouts[m]
13:            d ← t.draws[m]
14:            value ← (w+0,5·d)/p + c · sqrt(log(T)/p)
15:            if value > bestValue then
16:                bestValue ← value
17:                bestMove ← m
18:            end if
19:        end for
20:        play(board, bestMove)
21:        player ← opponent(player)
22:        res ← UCT(board, player, policy)
23:        update t with res                        ▷ Rückpropagierung
24:    else
25:        t ← new entry of board in the transposition table
26:        res ← playout(board, player, policy)
27:        update t with res                        ▷ Rückpropagierung
28:    end if
29:    return res
30: end function
```

2.4. Playout Policy Adaptation Algorithmen

Standard-UCT basiert auf der Sammlung statistischer Daten durch viele zufällige Playouts[19]. Doch mit zufälligen Zügen können sich Playouts ergeben, die sich in einem Spiel normalerweise nicht finden würden, wie z.B., dass eine SpielerIn den eigenen König in einen Check setzt. Die Idee von Playout Policy Adaptation ist es, durch Anpassung der Wahrscheinlichkeiten in der Zugauswahl schon in Playouts tendenziell günstigere Züge zu wählen und interessante Sequenzen näher zu erkunden, in der Hoffnung, dass dies die Qualität der gesammelten Daten und damit der Statistik hebt[4]. Dadurch soll eine bessere Zugauswahl ermöglicht und die Spielstärke gehoben werden.

Die für diese Arbeit relevanten PPA-Algorithmen werden im folgenden Text vorgestellt.

2.4.1. Nested Rollout Policy Adaptation (NRPA)[13]

NRPA führt die Rollout (auch Playout) Policy Adaptation ein. Im Gegensatz zu UCT arbeitet NRPA mit Rekursion. Jedes Level speichert hierbei die bisher beste gefundene Zugsequenz und den dazugehörigen Score.

Es wird eine Policy Tabelle erstellt, die für jeden validen Zug ein Gewicht enthält, welches mit Null initialisiert wird. Die Tabelle wird ausschließlich von höheren zu niedrigeren Rekursionslevels weitergegeben (Alg. 3, Zeile 15).

Nach Aufruf des Algorithmus erfolgt die Rekursion, bis Level 0 erreicht wird. Hier (Alg. 3, Zeile 2) wird ein Rollout/Playout vom Wurzelknoten aus gestartet, doch anders als in UCT ist dieses nicht komplett zufällig. Die Wahrscheinlichkeiten der Züge sind proportional zu den Exponenten der Gewichte der Züge (Alg. 3, Zeile 7). Dabei gibt `code(move)` lediglich die Kodierung des Zuges zurück; in dieser Arbeit ein Integer, der eindeutig das Start- und Zielfeld kodiert. Der Rückgabewert wird zur Indexierung der Policy Tabelle genutzt. `child(move)` gibt den Knoten zurück, der von dem aktuellen Knoten über den als Argument gegebenen Zug erreicht wird.

Die Wahrscheinlichkeit eines Zuges berechnet sich wie folgt:

$$P(m) = \frac{\mathrm{e}^{w_m}}{\sum_{m_i \in M} \mathrm{e}^{w_{m_i}}}$$

Aus dem Gewicht eines Zuges w_m wird jeweils die Wahrscheinlichkeit P berechnet, dass ein Zug m als nächstes aus der Menge aller legalen Züge M gewählt wird.

Das Playout endet, wenn ein Endspielzustand gefunden wurde und gibt anschließend die gespielte Sequenz und die Bewertung zurück in das nächst höhere Rekursionslevel. In jedem Level ist der bisher beste Score (mit Sequenz) gespeichert und ist der neue Rückgabewert besser, werden der beste Score und die beste Sequenz aktualisiert (Alg. 3, Zeilen 16–18).

Nach jedem rekursiven Aufruf wird die Policy Tabelle adaptiert, egal, ob die neue Sequenz besser war als die alte (Alg. 3, Zeile 20). In der `adapt()` Funktion wird jeder Zug in der aktuell besten Sequenz um α (eine globale Konstante) erhöht (Alg. 4, Zeile 5) und anschließend von allen legalen Zügen α mal der Exponent des Gewichts des jeweiligen Zuges subtrahiert (Alg. 4, Zeilen 7–8).

Im folgenden rekursiven Aufruf wird dann die angepasste Policy Tabelle als Argument gegeben.

Die zweite globale Konstante, N, legt fest, wie oft das jeweils niedrigere Level aufgerufen wird. Am Ende kann schließlich der erste Zug aus der besten Sequenz des höchsten Levels zurückgegeben werden.

Für Zwei-Personen-Spiele eignet sich dieser Algorithmus nicht, da dort nicht wirklich eine „beste Sequenz“ existieren kann: Eine Sequenz ist immer nur für eine der beiden SpielerInnen gut. NRPAs fundamentale Funktionsweise basiert aber auf dem Speichern und Nutzen dieser besten Sequenz. Während der Algorithmus sehr gut für einige Puzzles (z.B. Morpion Solitaire) funktioniert, ist er für Xiangqi nicht geeignet.

Algorithm 3 Nested Rollout Policy Adaptation[13]

```
1:  function NRPA(level, pol)
2:      if level = 0 then                                                    ▷ Rollout/Playout
3:          node ← root()
4:          ply ← 0
5:          seq ← {}
6:          while num_child(node) > 0 do
7:              choose seq[ply] ← move with probability proportional to exp(pol[code(move)])
8:              node ←child(seq[ply])
9:              ply ← ply + 1
10:         end while
11:         return (score(node), seq)
12:     else
13:         best_score ← −∞
14:         for N iterations do
15:             (result, new) ← NRPA(level − 1, pol)
16:             if result ≥ best_score then
17:                 best_score ← result
18:                 seq ← new
19:             end if
20:             pol ← ADAPT(pol, seq)
21:         end for
22:         return (best_score, seq)
23:     end if
24: end function
```

Algorithm 4 NRPA Adapt[13]

```
1:  function ADAPT(pol, seq)
2:      node ← root()
3:      pol′ ← pol
4:      for ply = 0 to length(seq) − 1 do
5:          pol′[code(seq[ply])] ← pol′[code(seq[ply])] + α
6:          z ← ∑ exp(pol[code(m)])
7:          for legal moves m from node do
8:              pol′[code(m)] ← pol′[code(m)] − α · exp(pol[code(m)]) / z
9:          end for
10:         node ←child(seq[ply])
11:     end for
12:     return pol′
13: end function
```

Die folgenden PPA-Algorithmen übernehmen das Konzept der Policy Tabelle mit `adapt()` Funktion und machen es für Zwei-Personen-Spiele nutzbar.

2.4.2. Playout Policy Adaptation (PPA)[4]

PPA bezeichnet sowohl eine Art von Algorithmus, in welchem Playouts von einer gelernten, dynamischen Policy geleitet werden, als auch einen fest definierten, von Tristan Cazenave entwickelten Algorithmus. Dieser Algorithmus basiert auf UCT (Alg. 1/2) und wurde von NRPA (Alg. 3) inspiriert.

Die `adapt()` Funktion für PPA ist in Algorithmus 5 dargestellt. Sie ist dem `adapt()` von NRPA sehr ähnlich. Auch `code(move)` ist äquivalent zu NRPA. Der bedeutende Unterschied ist, dass hier nur die Policy Tabelle angepasst wird, wenn es sich bei dem Gewinner des Playouts um die aktuelle SpielerIn in dem Zustand handelt (Alg. 5, Zeile 4). So werden nur Züge verstärkt, die zu einem Gewinn geführt haben, und PPA lernt keine schlechten Züge.

Algorithm 5 PPA Adapt[4]

1: **procedure** ADAPT($winner, board, player, playout, policy$)
2: $polp \leftarrow policy$
3: **for** $move$ **in** $playout$ **do**
4: **if** $winner = player$ **then**
5: $polp[\text{code}(move)] \leftarrow polp[\text{code}(move)] + \alpha$
6: $z \leftarrow \sum \exp(policy[\text{code}(m)])$
7: **for** m **in** possible moves on board **do**
8: $polp[\text{code}(m)] \leftarrow polp[\text{code}(m)] - \alpha \cdot \frac{\exp(policy[\text{code}(m)])}{z}$
9: **end for**
10: **end if**
11: play($board, move$)
12: $player \leftarrow opponent(player)$
13: **end for**
14: $policy \leftarrow polp$
15: **end procedure**

Algorithm 6 Die Playout Funktion[4]

1: **function** PLAYOUT($board, player, policy$)
2: **while** true **do**
3: **if** $board$ is terminal **then**
4: **return** winner($board$)
5: **end if**
6: $z \leftarrow \sum \exp(k \cdot policy[\text{code}(m)])$
7: choose m in possible moves with probability proportional to $\frac{\exp(k \cdot policy[\text{code}(m)])}{z}$
8: play($board, move$)
9: $player \leftarrow$ opponent($player$)
10: **end while**
11: **end function**

Es soll an dieser Stelle außerdem die Playout Funktion in Algorithmus 6 vorgestellt werden. Diese ist anwendbar für PPA und auch die folgenden PPA-Algorithmen. Die Wahrscheinlichkeit, dass ein Zug ausgewählt wird, ist proportional zu dem Exponent des Zuggewichts, geteilt durch die Summe aller exponierten Zuggewichte (also gemäß der Gibbs-Boltzmann-Verteilung). Für UCT wird die Zeile 7 durch eine Zugwahl mit gleich verteilter Wahrscheinlichkeit ersetzt.

Der Parameter k ist relevant für MAST, in anderen Algorithmen wird er, wie in Cazenaves Artikel[4], auf 1 gesetzt.

2.4.3. Playout Policy Adaptation with Penalty (PPAP)

Playout Policy Adaptation with Penalty (PPAP) ist eine Variante von PPA und wurde für diese Arbeit erdacht. Sie wird nicht in Cazenaves Artikel[4] aufgeführt, unterscheidet sich allerdings auch nur in wenigen Zeilen von PPA. Die `adapt()` Funktion ist leicht anders aufgebaut, wie in Algorithmus 7 zu sehen. Die maßgebliche Veränderung liegt in Zeile 7 und wurde magentafarben markiert.

Algorithm 7 PPAP Adapt[4]

1: **procedure** ADAPT($winner, board, player, playout, policy$)
2: $polp \leftarrow policy$
3: **for** $move$ **in** $playout$ **do**
4: **if** $winner = player$ **then**
5: $polp[\text{code}(move)] \leftarrow polp[\text{code}(move)] + \alpha$
6: **else**
7: $polp[\text{code}(move)] \leftarrow polp[\text{code}(move)] - \frac{\alpha}{p}$
8: **end if**
9: $z \leftarrow \sum \exp(policy[\text{code}(m)])$
10: **for** m **in** possible moves on board **do**
11: $polp[\text{code}(m)] \leftarrow polp[\text{code}(m)] - \alpha \cdot \dfrac{\exp(policy[\text{code}(m)])}{z}$
12: **end for**
13: play($board, move$)
14: $player \leftarrow$opponent($player$)
15: **end for**
16: $policy \leftarrow polp$
17: **end procedure**

Anders als bei Standard-PPA wird in PPAP für schlechte Züge, also solche, die zu verlorenen Endspielständen geführt haben, ein Wert relativ zu α abgezogen. Dabei ist p eine weitere globale Variable. Läuft p gegen Unendlich, konvergiert PPAP zu Standard-PPA.

Die Idee hinter der Änderung ist, dass vermeintlich schlechte Züge eine Strafe erhalten und dadurch im nächsten Playout weniger wahrscheinlich gewählt werden. Das führt dazu, dass sich schneller eine Tendenz bildet, als wenn nur die Gewichte der guten Züge erhöht werden. Parameter p wurde eingeführt, um steuern zu können, wie groß diese Tendenz ist. Legt sich der

Algorithmus zu schnell fest, können gute Züge übersehen werden, die nur zufällig in verlorenen Playouts vorgekommen sind.

Eine weitere Änderung zu PPA ist, dass die Berechnung der Summe z und die eigentliche Anpassung der Policy nun für jeden Zug in der Sequenz stattfinden. Dies kostet mehr Zeit und kann dazu führen, dass weniger Playouts durchgeführt werden können. Das lohnt sich dann, wenn dadurch eine hochwertigere Policy Tabelle entsteht.

Da die Wahrscheinlichkeiten für Züge mit den Exponenten der Zuggewichte berechnet werden, stellt es kein Problem dar, wenn die Gewichte der Züge negativ werden.

2.4.4. Playout Policy Adaptation with move Features (PPAF)[4]

PPAF ist eine Erweiterung von PPA. Am Großteil der Implementierung ändert sich nichts, z.B. bleiben `adapt()`, UCT und das Playout gleich. Die bedeutende Änderung liegt in `code(move)`. Ist ein Zug in PPA beispielsweise durch das Start- und Endfeld definiert, kann in PPAF ein Zug mit dem selben Start- und Endfeld mehr als eine Kodierung haben, denn es werden Zugfeatures mitberücksichtigt. Ein Feature kann prinzipiell alles sein: ob es sich um einen Schlagzug handelt, ob der Zug die Figur auf ein bedrohtes Feld setzt, den Wert der Figur (hierfür ist Domänenwissen notwendig), etc. Es muss allerdings darauf geachtet werden, dass mehr einbezogene Features tendenziell mehr Playouts brauchen, um aussagekräftige Statistiken zu erstellen. Gibt es zu viele Codes für einen Zug, bzw. hat der Algorithmus nicht genügend Zeit, wird er keine belastbare Strategie entwickeln können. In dieser Arbeit werden zwei binäre Features verwendet: ob es sich um einen Schlagzug handelt und ob das Endfeld vom Gegner bedroht wird.

In Cazenaves Experimenten schnitt PPAF in den meisten Spielen sehr gut im Vergleich mit den anderen PPA-Algorithmen ab. Hierbei muss allerdings beachtet werden, dass die Vergleiche mit einer festen Anzahl an Playouts durchgeführt wurden, nicht mit einer festen Zeit. Wird allen Algorithmen die gleiche Zeit für eine Suche zugeteilt, kann sich das Ergebnis ändern, da z.B. MAST eine schnellere `adapt()` Funktion hat und mehr Playouts in gleicher Zeit ausführen kann, was die Statistik stärkt.

Wie auch PPA erhält PPAF außerdem eine Version mit Penalty für schlechte Züge/Zugfeatures: Playout Policy Adaptation with move Features and Penalty (PPAFP). Diese stammt nicht aus Cazenaves Artikel[4] und kombiniert lediglich das `adapt()` von PPAP mit dem `code(move)` aus PPAF, weswegen kein eigener Pseudocode abgebildet wird.

2.4.5. Move Average Sampling Technique (MAST)[4]

Während PPA und PPAF lernen, indem sie einen Zug gegen andere, in dem Zustand verfügbare Züge vergleichen (ähnlich zu Reinforcement Learning), nutzt MAST über die gesamte Zugsuche gesammelte Daten. Dafür sind die blauen Zeilen in Algorithmus 1 relevant: es wird für jeden Zugcode gespeichert, wie oft er gespielt wurde, und wie oft er in einer Sequenz enthalten war, die zu einem Sieg führte. Basierend auf diesem Verhältnis wird die Policy Tabelle angepasst (Alg. 8, Zeile 7).

Der Vorteil ist, dass in `adapt()` nicht jedes Mal über alle legalen Züge des Zustands iteriert werden muss, was eine Verringerung der Laufzeit mit sich führt. Allerdings dauert es für MAST so ebenfalls länger, eine Policy zu lernen, denn es wird immer nur das Gewicht des im Playout enthaltenen Zuges angepasst, nicht wie in PPA und PPAF das Gewicht aller legalen Züge.

Algorithm 8 MAST Adapt[4]

1: **procedure** ADAPT($winner, board, player, playout, policy$)
2:     **for** $move$ **in** $playout$ **do**
3:         $games[\text{code}(move)] \leftarrow games[\text{code}(move)] + 1$
4:         **if** $winner = player$ **then**
5:             $wins[\text{code}(move)] \leftarrow wins[\text{code}(move)] + 1$
6:         **end if**
7:         $policy[\text{code}(move)] \leftarrow \dfrac{wins[\text{code}(move)]}{games[\text{code}(move)]}$
8:         $\text{play}(board, move)$
9:         $player \leftarrow \text{opponent}(player)$
10:     **end for**
11:     $policy \leftarrow polp$
12: **end procedure**

2.5. Vergleich

2.5.1. Gewinnprozentsatz

Die offensichtlichste Weise, Zugfindungsalgorithmen zu vergleichen, ist anhand des Gewinnprozentsatzes: Der Algorithmus, der mehr Spiele gewinnt, hat eine größere Spielstärke. Dies ist auch der Vergleichsansatz in Cazenaves Artikel[4]. Je mehr Spiele durchgeführt werden, desto aussagekräftiger wird dieser Wert. Der Gewinnprozentsatz selbst gibt aber keine Aussage über die Anzahl der ausgeführten Spiele.

Die Anzahl der Spiele in dieser Arbeit wird durch den entstehenden Zeitaufwand begrenzt, denn vor allem bei größerem Zeitbudget pro Zug kann sich eine Vergleichsreihe über dutzende Stunden ziehen.

2.5.2. Elo-Zahl

Es ist schwierig, die Spielstärke eines Menschen oder auch einer Spiele-KI komplett fair zu bewerten. Weit verbreitet ist heutzutage die Elo-Zahl, benannt nach ihrem Erfinder Arpad Elo. Sie stellt ein relatives Bewertungssystem dar. Die Elo-Zahl gibt dabei Aufschluss über die Wahrscheinlichkeit, welche SpielerIn ein Spiel gewinnen wird.[5]

[5] https://en.wikipedia.org/wiki/Elo_rating_system. Letzter Zugriff: 11.12.2025

Hat eine SpielerIn beispielsweise ein um 100 Punkte höheres Elo-Rating, ist die Wahrscheinlichkeit, dass sie gewinnen wird, 64%. Bei 200 Punkten Unterschied liegt die Wahrscheinlichkeit bei 76%.[6]

In dieser Arbeit wird zum Vergleich der verschiedenen Algorithmen, neben dem Gewinnprozentsatz, auch die Elo-Differenz errechnet. Diese stellt nicht nur die Ergebnisse der simulierten Spiele dar, sondern gibt auch einen Tipp ab, wie wahrscheinlich ein Algorithmus gegen den anderen in zukünftigen Spielen gewinnen wird. Die Anzahl der Unentschieden und die Anzahl der Spiele insgesamt werden mit einbezogen, was zu einer differenzierteren Bewertung führen sollte.[7]

Die verwendete Formel[8] ist:

$$\text{Elo-Differenz} = -\frac{\log\left(\frac{1}{\text{w}} - 1\right) \cdot 400}{\log(10)}$$

mit Gewinnanteil w:

$$\text{w} = \frac{\text{Gewinne} + 0.5 \cdot \text{Unentschieden}}{\text{Spiele}}$$

Es ist allerdings zu beachten, dass es sich bei den berechneten Werten um die Elo-Differenz handelt, nicht die Elo-Bewertung. Die Elo-Differenz gilt nur in Bezug auf die beiden verglichenen Algorithmen. Sie kann nicht dazu verwendet werden, Algorithmen zu vergleichen, die nicht direkt gegeneinander gespielt haben, wie es mit der Elo-Bewertung möglich wäre. Um eine Elo-Bewertung zu erhalten, müssten die Algorithmen gegen eine KI oder einen Menschen mit bereits bekannter Elo-Bewertung spielen, um eine Einstufung zu ermöglichen.

2.5.3. Likelihood of Superiority (LOS)

Anders als bei der Gewinnrate fließt in der Likelihood of Superiority (LOS)[9] die Anzahl der Spiele insgesamt mit ein. Es wird ein Wert zurückgegeben, der anzeigt, wie wahrscheinlich Algorithmus A wirklich stärker ist als Algorithmus B.

So ist bei insgesamt zehn Spielen, von denen A sieben gewinnt und B drei, die Gewinnrate die gleiche, wie bei hundert Spielen mit je siebzig und dreißig Gewinnen. Die LOS (aus As Sicht) wird dagegen im zweiten Fall deutlich höher sein, da eine Basis von hundert Spielen statistisch aussagekräftiger ist als eine mit nur zehn.

[6] https://en.wikipedia.org/wiki/Elo_rating_system. Letzter Zugriff: 11.12.2025
[7] https://www.chessprogramming.org/Match_Statistics. Letzter Zugriff: 11.12.2025
[8] https://www.chessprogramming.org/Match_Statistics. Letzter Zugriff: 11.12.2025
[9] https://www.chessprogramming.org/Match_Statistics. Letzter Zugriff: 11.12.2025

Die LOS wird häufig in Schach-KIs benutzt, um die Spielstärke besser bewerten zu können. Auch in dieser Arbeit wird sie für den Vergleich der Algorithmen verwendet, mit folgender Formel[10]:

$$\text{LOS} = 0,5 + 0,5 \cdot \text{erf}(\frac{\text{Gewinne} - \text{Verluste}}{\sqrt{2 \cdot (\text{Gewinne} + \text{Verluste})}})$$

Dabei steht erf für die (Gauß'sche) Fehlerfunktion[11].

2.6. Programmiersprache

Als Programmiersprache wurde für diese Arbeit C++ gewählt. Der Hauptgrund für diese Wahl ist die hohe Laufgeschwindigkeit, die durch Compileroptimierungen möglich ist. Denn je schneller die einzelnen Funktionen ausgeführt werden, desto tiefer kommt bspw. ABS in einer vorgegebenen Zeit oder desto mehr Playouts können ausgespielt werden. Das verbessert die Funktion der Algorithmen.

In vielen Tests[11] steht C++ im Punkt Laufgeschwindigkeit nur hinter C, aber deutlich vor Java oder Python. Zwischen C und C++ war es dann eine Wahl nach persönlicher Präferenz.

[10] https://www.chessprogramming.org/Match_Statistics. Letzter Zugriff: 11.12.2025
[11] https://en.wikipedia.org/wiki/Error_function. Letzter Zugriff: 11.12.2025

3. Implementierung

Die komplette Implementierung des Spiels und der Algorithmen für diese Arbeit erfolgte in C++ Version 201703 (C++17).

Die Implementierung umfasst: eine Spielklasse für Xiangqi, Xiangqi Spielregeln, ein Bitboard-Struct, ABS mit Transposition Table und Zobrist Hashing, UCT, alle in Sektion 2.4 erklärten PPA-Algorithmen außer NRPA und eine Funktion zum empirischen Vergleich der Algorithmen.

Es wurden außerdem Skripte für die Erstellung der in Kapitel 4 gezeigten Ergebnisse geschrieben, sowie zusätzliche Skripte für die Ausführung der Versuche auf dem Computing Cluster Hydra der TU Berlin.

Besonders relevante Punkte/Designentscheidungen werden im Folgenden genauer erläutert.

3.1. Spieldarstellung

Das Spiel selbst wurde als Klasse definiert, die vor allem die aktuelle SpielerIn, ein `std::array` der Größe neunzig mit Integern für die Figuren (positiv schwarz, negativ rot) und Null für freie Felder, und ein Array von Bitboards enthält.

3.2. Bitboards

Bitboards wurden erstmals in den 1950er Jahren für Computer Schach verwendet. Gegenüber Integer Arrays haben sie zwei bedeutende Vorteile: Sie benötigen weniger Speicherplatz und ermöglichen außerdem sehr schnelle Berechnungen[3]. Anstelle eines Integers pro Feld auf dem Spielbrett entspricht hier jedes einzelne Bit einem Feld.

Das Bitboard-Struct für diese Arbeit wurde mit der Onlineversion von Microsoft Copilot (GPT-5.1) generiert. Die einzelnen Funktionen für das Struct wurden einzeln über mehrere Sessions angefordert, aber ebenfalls generiert. Darauf wird auch in einem Kommentar im entsprechenden Code hingewiesen.

3.2.1. Speicher

Im Fall von Xiangqi ist ein Spielbrett neunzig Felder groß. Würde ein Integerarray gespeichert werden, würden (auf einem 64-bit System mit einer Integergröße von 4 Byte) für das Board also 360 Byte anfallen.

Anders als im Westlichen Schach, welches mit 64 Feldern perfekt in einen 64-bit Integer passt, benötigt Xiangqi neunzig Bit. In dieser Arbeit wird das durch ein Struct mit einem 64-bit und einem 32-bit Integer umgesetzt. Aber selbst wenn das automatische Padding die Größe des Bitboards auf zwei 64-bit Integer erhöht, ergibt das nur einen Speicherverbrauch von sechzehn Byte für ein gesamtes Board – der gleiche Platz wie vier Felder in einem Integerarray.

Dafür müssen, um alle Informationen zu erhalten und einen Spielstand vollständig darzustellen, mehrere Bitboards gespeichert werden. In dieser Implementierung gibt es jeweils ein Bitboard, welches alle Figuren einer Farbe enthält, ein Bitboard je Figurentyp (egal welcher Farbe) und ein Bitboard mit allen Figuren. Bei sieben verschiedenen Figuren ergibt sich ein Speicherverbrauch von nur 160 Bytes, was problemlos mehrmals in den L1 Cache passt[1]. Dadurch können Zugriffe sehr schnell erfolgen.

3.2.2. Leistung

Änderungen am Bitboard erfolgen mit binären Operationen, welche sehr schnell ausgeführt werden können[3].

Sollen z.B. nur die schwarzen Soldaten gefunden werden, wird das Bitboard für schwarze Figuren mit dem Bitboard für Soldaten verundet – das passiert für das gesamte Board parallel und muss nicht für jedes Feld einzeln durchgeführt werden[3]. Es bleiben die Bits 1, die Felder mit schwarzen Soldaten kennzeichnen. Über diese kann dann beispielsweise mit `__builtin_ctzll`[2] iteriert werden, eine bitweise Funktion, die die Anzahl der „trailing Zeros“ zurückgibt. Das entspricht dem Index der Figur auf dem Board. C++ stellt auch die Klasse `Bitset` zur Verfügung, diese wurde allerdings im finalen Code nicht verwendet, unter anderem um die Compiler-eigene `__builtin_ctzll` Funktion nutzen zu können.

3.3. Zuggenerator

Der Zuggenerator setzt die Spielregeln für Xiangqi um und muss in der Lage sein, möglichst schnell alle legalen Züge für einen Zustand zurückzugeben. Hier können auch auf den ersten Blick unbedeutende Optimierungen relevant sein, denn die Funktion zur Zuggeneration wird in ABS und auch den PPA-Algorithmen einige hunderttausend bis Millionen Mal (je nach zugeteilter Zeit pro Zug) ausgeführt.

In dieser Arbeit wurde der Zuggenerator als eine Klasse umgesetzt.

[1] https://www.elektronik-kompendium.de/sites/com/0309291.htm. Letzter Zugriff: 06.02.2026
[2] https://www.geeksforgeeks.org/c/builtin-functions-gcc-compiler/. Letzter Zugriff: 25.02.2026

In der ersten Codeversion wurden Vektoren mit einer Zugklasse gefüllt und jeder Figurentyp hatte eine eigene Funktion, deren Ergebnis an die übergreifende Zuggenerationsfunktion weitergegeben und an einen einzelnen Zugvektor angehängt wurde. Bitboards basierten auf der C++ eigenen Klasse `Bitset` und alle Berechnungen für Züge fanden zur Spiellaufzeit statt. Die Zeit für die Zuggeneration aus der Startposition von Xiangqi lag in für diese Arbeit durchgeführten Benchmarks zwischen fünfzehn und zwanzig Mikrosekunden. Für ABS ist das sehr langsam, denn die Zuggenerationsfunktion wird in jedem untersuchten Knoten aufgerufen und wird mit einer langsamen Laufzeit zum Leistungsengpass.

In der zweiten Codeversion wurden die Vektoren durch ein Zuglisten-Struct ersetzt, das ein Zugarray (mit einer Zugklasse) und einen Integer als Zähler enthält. Außerdem wurde die Zuggeneration für alle Figuren in eine einzige Funktion geschrieben, um das Kopieren von Zügen zu vermeiden. Bitsets wurden ersetzt durch das eigene Bitboard-Struct (Sek. 3.2), das sehr schnell über Bits iteriert. Außerdem wurde die Zuggenerationsklasse erweitert, sodass bei ihrer Erstellung Vorberechnungen durchgeführt werden, die die Zuggeneration zur Laufzeit des Algorithmus beschleunigen. Das verringerte die Laufzeit der Zuggenerationsfunktion für die Startposition auf durchschnittlich rund fünf Mikrosekunden.

Für die dritte Codeversion wurde das Zuglisten-Struct so verändert, dass es nicht mehr mit einer Zugklasse arbeitet sondern mit Integern, die je einen Zug kodieren. Die verwendete Formel ist:

$$\text{Zug} = \text{Startfeld} \cdot 90 + \text{Zielfeld}$$

Alle entsprechenden Funktionen wurden angepasst. Die Zuggenerationsfunktion läuft jetzt auf der Startposition bei einmaliger Ausführung im Schnitt zwischen 3,5 und 4 Mikrosekunden. Für die ABS bedeutet das, dass sie von der Startposition in unter einer Sekunde Suchtiefe fünf beenden kann (ohne Iterative Deepening, mit Transposition Table).

Die eingeführten Vorberechnungen finden bei der Erstellung eines Zuggenerator Objekts statt und beschleunigen die spätere Zuggeneration. Dabei wird ein höherer Speicherverbrauch in Kauf genommen, um die Performanz während des Spiels zu steigern.

Für den König, die Berater und die Soldaten werden in dieser Vorberechnung für jedes der neunzig Felder (bzw. nur die Palastfelder) alle möglichen Zielfelder ausgerechnet und anschließend als Bitboards in einem Array gespeichert. Indexiert werden diese Bitboards durch den Index des Startfeldes. In der eigentlichen Zuggeneration muss dann nur noch das entsprechende Bitboard aus dem Array abgerufen und mit der Inverse des Bitboards befreundeter Figuren verundet werden. Das Ergebnis ist ein Bitboard mit dem Wert 1 für alle legalen Zielfelder. In einer Schleife kann nun mit `__builtin_ctzll` über diese Einsen iteriert, je der Zugcode berechnet und im Rückgabearray gespeichert werden.

Für den König wird außerdem zur Spiellaufzeit einmal in einer Schleife abgesucht, ob er dem anderen König direkt und ohne Hindernisse gegenübersteht. Diese Information ist relevant für den „flying King“ Zug.

Für die Pferde und Elefanten braucht es einen Extraschritt, denn hier muss eine Prüfung stattfinden, dass keine anderen Figuren illegal übersprungen werden. Beim Elefanten wird ebenfalls

ein Bitboard mit Kandidaten vorberechnet. Über die Einsen im Kandidatenbitboard wird iteriert. Index idx_t bezeichnet das mögliche Zielfeld, Index idx_e das Startfeld (wo der Elefant steht). Mit diesen Werten lässt sich einfach der Index des Feldes idx_{frei} berechnen, welches frei bleiben muss:

$$idx_{frei} = \frac{idx_e + idx_t}{2}$$

Bei Pferden wird bei Erstellung der Zuggeneratorklasse zusätzlich ein 2D Array angelegt, welches den Feldindex zurückgibt, der für die Gültigkeit des Zuges frei sein muss. Dies kann in dem Bitboard, das alle Figuren enthält, schnell überprüft werden.

Am schwierigsten gestaltet sich die Berechnung für die sogenannten „sliding pieces“, den Streitwagen und die Kanone. Hier wurden verschiedene Implementierungen mit vorberechneten Strahlen pro Position und Richtung ausprobiert, jedoch verbesserte keine von ihnen die Schnelligkeit über eine simple Schleife, die zur Spiellaufzeit das Feld in alle Richtungen nach Hindernissen absucht. Vermutlich wird diese vom Compiler stark optimiert und ist deswegen schneller als schlechter zu optimierende Implementationen.

Der Code für die Benchmarks wurde mit GCC 13.3.0 kompiliert und lief auf einer Maschine mit Ubuntu 24.04 (CPU: AMD Ryzen 7 3750H with Radeon Vega Mobile Gfx).

3.4. PPA-Algorithmen

Bei der Umsetzung der Playout Policy Adaptation Algorithmen wurde sich so eng wie möglich an den Pseudocode von Cazenave[4] gehalten.

Die Transposition Table (TT) dient hier, anders als bei ABS, weniger der Beschleunigung der Suche und mehr dem Aufbau eines Suchbaums – bzw. handelt es sich mit der Nutzung einer TT nicht mehr um einen Baum, sondern um einen gerichteten Graphen, denn Knoten können mehr als ein Elternteil haben und auch Zyklen sind möglich. Ebenfalls anders als in ABS wird hier deswegen kein Array für die TT verwendet, sondern eine Hashmap mit dem Zobrist Hash der Bordposition als Schlüssel. Die Hashmap (`std::unordered_map`) hat eine Kollisionsstrategie[3], sodass Knoten nie verwechselt werden – das würde die gelernten Policies korrumpieren und potentiell die Funktionsweise der Algorithmen stark beeinträchtigen.

Ein Tabelleneintrag ist ein Struct, welches einen Boolean **is_set** enthält, den Hashwert, einen Integer für die absolute Zahl an Playouts, in welchen die Position besucht wurde, und drei `std::unordered_maps`, die jeweils einen Zug und die dazugehörige Anzahl der Playouts, der Gewinne und der Unentschieden enthalten. Es gibt außerdem einen Vektor mit allen legalen Zügen für den Zustand, damit diese nicht jedes Mal neu berechnet werden müssen.

Auch die Policy wurde als Struct umgesetzt. Sie beinhaltet ein `std::array weights`, welches pro SpielerIn eine `std::unordered_map` enthält. Die Hashmaps ordnen einem Zugcode je ein

[3] https://en.cppreference.com/w/cpp/container/unordered_map.html. Letzter Zugriff: 06.02.2026

Gewicht zu. Das Struct umfasst außerdem eine Funktion, die mit einer Wahrscheinlichkeit proportional zu den Gewichten einen nächsten Zug auswählen und zurückgeben kann.

Das α für PPA, PPAP, PPAF und PPAFP, der Penalty-Wert p für PPAP und PPAFP, der Wert k für MAST, die Explorationskonstante c für alle UCT-basierten Algorithmen, und die in einem Playout abgelaufene Zugsequenz werden global gespeichert. Die Zugsequenz wird anschließend in die `adapt()` Funktion gegeben.

Für jeden UCT-basierten Algorithmus existiert eine globale Transposition Table des Typs `std::unordered_map` mit dem eigenen Suchgraphen.

Die Algorithmen werden durch einen Integerwert kodiert, der es ermöglicht, je nur eine einzige Funktion für den Aufruf, UCT, das Playout und die Zugauswahl des zurückzugebenden Zuges zu implementieren. Für die Auswahl der korrekten Selektionsfunktion im Playout und der richtigen `adapt()` Funktion können `if`-Abfragen und der Integerwert für den Algorithmus verwendet werden. Dadurch gibt es weniger duplizierten Code.

4. Ergebnisse

Da die durchgeführten Vergleiche jeweils einige Stunden dauern und das Speichern des Suchgraphen mehrere Gigabyte verbraucht, wurden sie auf dem Computing Cluster Hydra der TU Berlin durchgeführt. Dadurch unterscheiden sich die CPU Spezifikationen teilweise von Versuch zu Versuch. Ein direkter Vergleich von beispielsweise der durchschnittlichen Anzahl an Playouts pro Zug ist also nicht immer direkt möglich; es müssen die technischen Daten der CPU mit einbezogen werden, da diese Einfluss auf die Laufgeschwindigkeit des Codes haben.

Alle Server laufen mit Ubuntu 22.04.2 und kompiliert wurde der Code mit GCC 11.4.0, mit Hilfe von CMake 3.22.1.

In allen Tabellen mit Versuchsergebnissen (also auch solchen im Anhang) wird die Elo-Differenz nur für einen der Algorithmen aufgeführt, da es sich nicht um einen festen Elo-Wert, sondern lediglich die Differenz zwischen den Algorithmen handelt.

Farblich in Tabellen markierte Werte für einen variierenden Parameter kennzeichnen jeweils den Wert, welcher für folgende Versuche ausgewählt wurde.

Playouts pro Zug sind nur relevant für UCT-basierte Algorithmen, c ebenfalls. α ist nur für PPA, PPAP, PPAF und PPAFP von Bedeutung, k nur für MAST. Der Penalty-Wert p ausschließlich für PPAP und PPAFP. Alle nicht relevanten Werte werden in Tabellen und Beschreibungen weggelassen.

Je nach der Hälfte der Spiele einer Versuchsreihe wurde der startende Algorithmus gewechselt.

Reguläre Regeln für Unentschieden in Xiangqi wurden der Vereinfachung halber nicht implementiert. Wird im Folgenden von unentschiedenen Spielen gesprochen, so ist gemeint, dass das festgelegte maximale Zuglimit für ein Spiel erreicht wurde.

Es werden folgende Metriken in den Tabellen in diesem Kapitel und im Anhang verwendet:

Gewinne: Absolute Anzahl von Gewinnen
Rate: Gewinnrate
Playouts: Durchschnittliche Anzahl von Playouts pro Zug
Schn. Sp.: Schnellstes gewonnenes Spiel (Anzahl Züge)
Elo-Diff.: Elo-Differenz
Unent.: Absolute Anzahl von Unentschieden
Züge/Sp.: Durchschnittliche Anzahl an Zügen pro Spiel

4.1. PPA vs. PPAP

Es wird im folgenden Vergleich nur der Parameter p variiert, der den Penalty-Wert in PPAP zur Abschwächung wahrscheinlich schlechter Züge reguliert.

CPU: Intel(R) Xeon(R) Platinum 8480CL

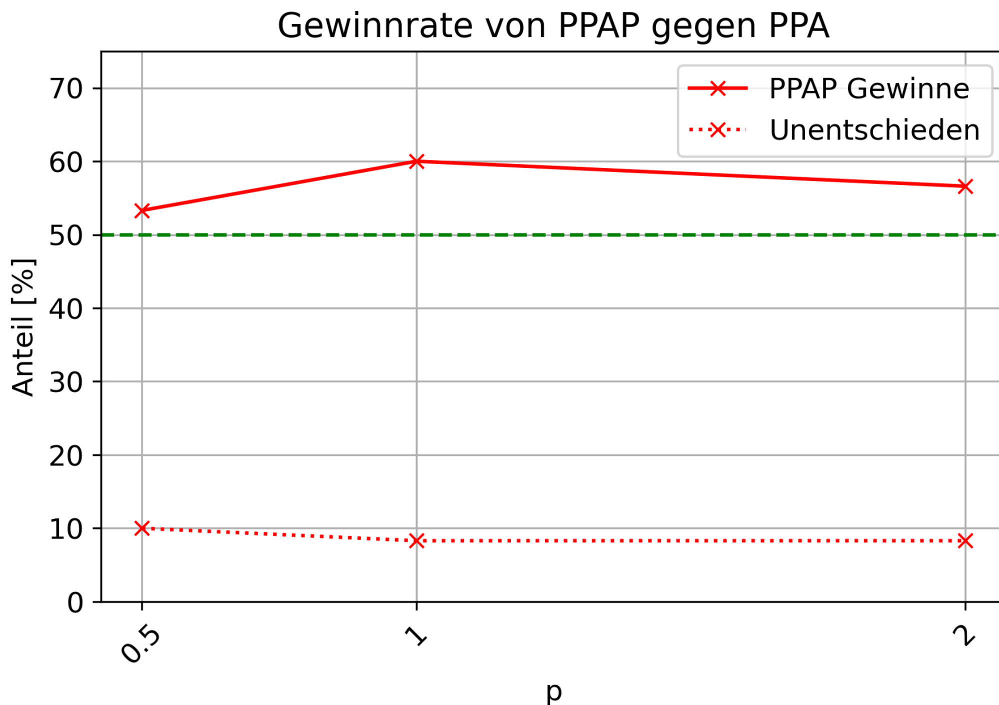

Abbildung 4.1.: Vergleich der Gewinnrate von PPAP gegen PPA mit variierendem p nach 60 Spielen.
$\alpha = 1$, $c = 1{,}414$, Zeit pro Zug = 2s, max. Zugzahl pro Spiel = 120.

Die Abbildung 4.1 zeigt, dass PPAP für alle getesteten Werte für p etwas besser abschneidet als reguläres PPA: Es gibt wenige Unentschieden und PPAP hat eine Gewinnrate von über 50% für alle getesteten Werte.

In 4.1 wird exemplarisch eine Tabelle mit allen gemessenen Werten dargestellt. Für folgende Versuche finden sich die vollständigen Tabellen jeweils im Anhang.

Tabelle 4.1 zeigt ebenfalls die höhere Gewinnrate von PPAP. Es wird auch ersichtlich, dass der absolute Unterschied von gewonnen Spielen nicht sehr hoch ist. Die LOS liegt trotzdem jeweils über 90% und die Elo-Differenz ist positiv für PPAP für alle p-Werte. Daraus folgend und weil sich die Algorithmen sehr ähnlich sind, wird in den folgenden Vergleichen mit PPAP weitergearbeitet und PPA außer Acht gelassen. Als Wert für p wird 1 festgelegt, da dieser laut Tabelle 4.1 die beste Gewinnrate, LOS und Elo-Differenz liefert.

Ebenfalls sichtbar ist eine leichte Tendenz in PPAP, weniger Playouts pro Zug zu schaffen, als PPA es tut.

Die Anzahl der Züge pro Spiel ist relativ niedrig, wenn sie mit der durchschnittlichen Länge eines Xiangqi-Spiels verglichen wird: diese liegt bei 95 Zügen pro Spiel[17].

Metrik	PPA	PPAP	PPA	PPAP	PPA	PPAP
p	0,5		1		2	
Gewinne	22	32	19	36	21	34
Rate [%]	36,67	53,33	31,66	**60**	35	56,67
Playouts	8246	8130	7799	7449	8023	7607
Schn. Sp.		5	7		7	
Elo-Diff.		58		101		76
LOS [%]	8,67	91,32	1,09	98,91	3,98	96,02
Unent.	6		5		5	
Züge/Sp.	45		37		41	

Tabelle 4.1.: PPA vs. PPAP mit variierendem p nach 60 Spielen.
$\alpha = 1$, $c = 1{,}414$, Zeit pro Zug = 2s, Max Zugzahl pro Spiel = 120.

4.2. PPAF vs. PPAFP

Es wird im folgenden Vergleich nur der Parameter p variiert, der den Penalty-Wert in PPAFP reguliert.

CPU: Intel(R) Xeon(R) Platinum 8480CL

Wie PPA erreicht PPAF mit einer Strafe für schlechte Züge/Zugfeatures für jeden getesteten Wert für p eine Gewinnrate von über 50% (Abb. 4.2). Allerdings ist der Unterschied in der Anzahl der gewonnenen Spiele geringer (Anhang, Tab. A.1), sodass auch die LOS weniger aussagekräftig ist. Ein durchschnittliches Spiel ist mit um die 15 Züge extrem kurz.

Auffällig ist die etwas höhere Anzahl von Playouts pro Zug für PPAFP gegenüber PPAF (Tab. 4.2). Dies ist anders als in den Versuchen mit PPA (Tab. 4.1). Insgesamt zeigt PPAF (Tab. 4.2) deutlich weniger Playouts pro Zugsuche als PPA (Tab. 4.1).

Obwohl der Unterschied zwischen PPAF und PPAFP gering ist, wird in folgenden Versuchen PPAFP verwendet. Für p wird der Wert 2 gewählt.

Metrik	PPAF	PPAFP	PPAF	PPAFP	PPAF	PPAFP
p	0,5		1		2	
Rate [%]	45	**55**	46,67	53,33	45	**55**
Playouts	1552	1804	1576	1776	1653	1817
Züge/Sp.	15		14		16	

Tabelle 4.2.: PPAF vs. PPAFP mit variierendem p nach 60 Spielen (gekürzt).
$\alpha = 1$, $c = 1{,}414$, Zeit pro Zug = 2s, max. Zugzahl pro Spiel = 120.

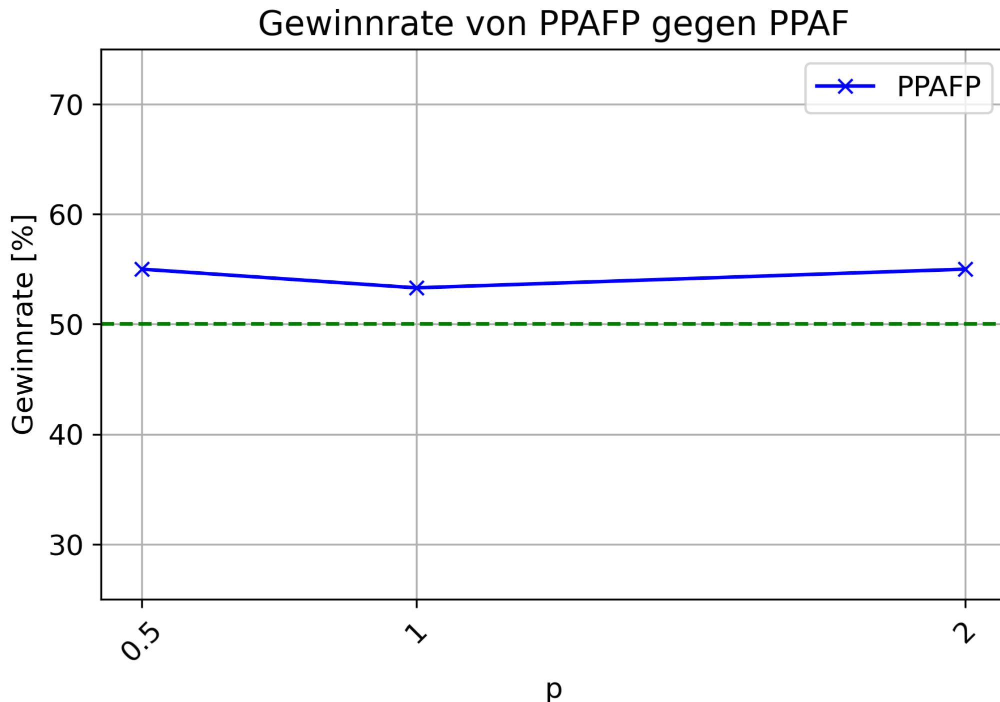

Abbildung 4.2.: Vergleich der Gewinnrate von PPAFP gegen PPAF mit variierendem p nach 60 Spielen.
$\alpha = 1$, $c = 1{,}414$, Zeit pro Zug = 2s, max. Zugzahl pro Spiel = 120.

4.3. α-Variation

Es wird im folgenden Vergleich nur der Parameter α variiert, der bestimmt, wie schnell wahrscheinlich gute Züge in der Policy Tabelle von PPAP und PPAFP gestärkt werden.

Die Werte für α wurden dabei aus dem Artikel von Cazenave[4] übernommen, bzw. wurden solche Werte ausgewählt, die in seinen Experimenten für verschiedene Spiele am besten abschnitten.

CPU: Intel(R) Xeon(R) Gold 6326 CPU @ 2.90GHz

In Abbildung 4.3 ist zu erkennen, dass sowohl PPAP als auch PPAFP keine hohen Gewinnraten gegen UCT erreichen können. Der Peak liegt bei 15% für PPAFP für einen α-Wert von 0,32. In allen anderen Konstellationen ist die Gewinnrate für die PPA-Algorithmen niedriger.

Auch wird sichtbar, dass vor allem für PPAP die Zahl der mit einem Unentschieden beendeten Spiele über dem Anteil der gewonnenen Spiele liegt. Für PPAFP ist dies weniger der Fall. Ob es sich bei den Unentschieden um ein ausgeglicheneres Spiel handelt oder sie auf wiederholende Zugfolgen zurückzuführen sind, wird aus den Ergebnissen nicht deutlich.

Im Spiel gegen PPAP kann UCT trotz des hohen Anteils an Unentschieden immer eine Gewinnrate von über 75% erreichen (Tab. 4.3). Der Peak für PPAP liegt hier bei einer Gewinnrate von 10% für einen α-Wert von 0,04. Die gleiche Elo-Differenz kann allerdings auch in anderen

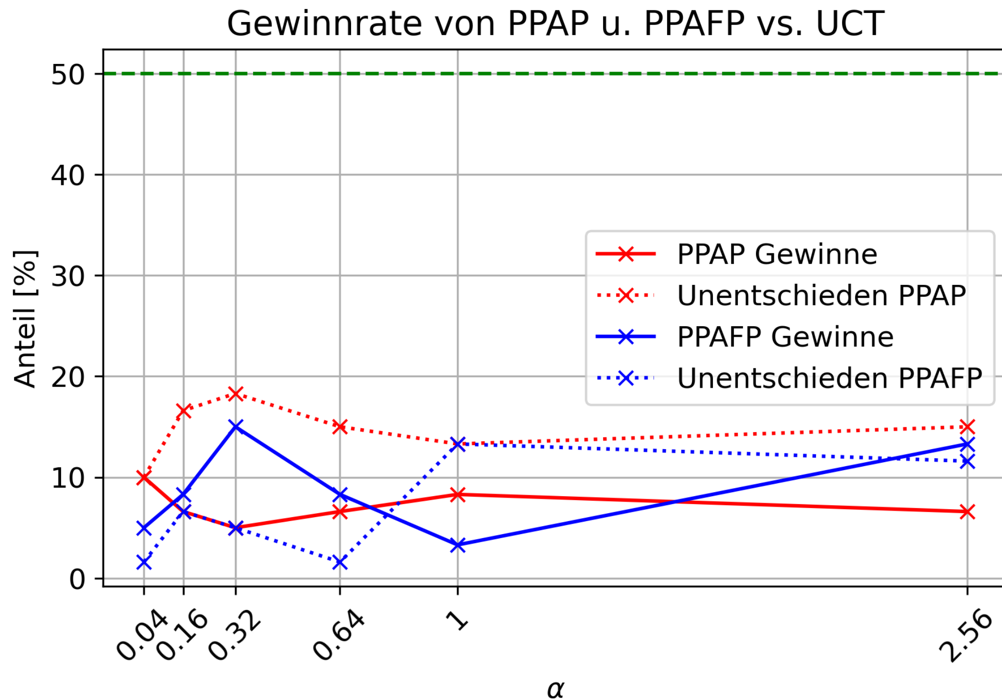

Abbildung 4.3.: Vergleich der Gewinnraten von PPAP u. PPAFP gegen UCT mit variierendem α nach 60 Spielen.
p (PPAP) = 1, p (PPAFP) = 2, c = 1,414, Zeit pro Zug = 2s, max. Zugzahl pro Spiel = 120.

Messungen, nämlich mit α = 0,16 und 1, betrachtet werden (Anhang, Tab. B.1). Eine unterschiedliche Gewinnrate mit gleicher Elo-Differenz ist bei einer unterschiedlichen Zahl von Unentschieden möglich.

Die LOS liegt für jeden gemessenen Wert bei rund 100% zugunsten von UCT (Anhang, Tab. B.1). Das und die hohen Elo-Differenzen sagen aus, dass UCT auch zukünftige Spiele mit hoher Wahrscheinlichkeit gewinnen wird.

Ein durchschnittliches Spiel ist im Kontext von Xiangqi mit um die 40 Züge (Anhang, Tab. B.1) kurz. Das schnellste Spiel für jede Versuchsreihe wird von UCT gewonnen, immer in unter zehn Zügen. Gibt es mehrere kürzeste Spiele, geht diese Information verloren, da die Vergleichsfunktion nur ein kürzestes Spiel aufgezeichnet hat.

Wird die Anzahl der Playouts betrachtet, so findet sich dort eine Differenz von mehreren Größenordnungen zwischen UCT und PPAP (Tab. 4.3). UCT erreicht deutlich über einhunderttausend Playouts, PPAP nur zwischen sieben- und zehntausend.

Da mehrere α-Werte die selbe Elo-Differenz vorweisen, wird das α für PPAP über die Gewinnrate bestimmt. Diese ist für ein α von 0,04 am größten.

PPAFP kann ebenfalls keine hohen Gewinnraten gegen UCT erreichen. UCT gewinnt für jeden getesteten Wert 75% der Spiele oder mehr (Tab. 4.4). Am größten ist die Differenz bei dem α-Wert 0,04, wo UCT eine Gewinnrate von 93,33% erreicht.

Metrik	UCT	PPAP	UCT	PPAP	UCT	PPAP	UCT	PPAP	UCT	PPAP	UCT	PPAP
α	0,04		0,16		0,32		0,64		1		2,56	
Rate [%]	80	**10**	76,67	6,57	76,67	5	78,33	6,67	78,33	8,33	78,33	6,67
Playouts	140587	7037	136798	7525	136000	9079	136865	8204	141818	8502	141374	9948

Tabelle 4.3.: UCT vs. PPAP mit variierendem α nach 60 Spielen (gekürzt).
$p = 1$, $c = 1{,}414$, Zeit pro Zug = 2s, max. Zugzahl pro Spiel = 120.

Auch gegen PPAFP kann UCT in jedem getesteten Fall eine LOS von rund 100% erreichen (Anhang, Tab. B.2). Die Elo-Differenzen liegen bei Werten zwischen 250 und 490. Beides zeigt eine hohe Wahrscheinlichkeit an, dass UCT zukünftige Spiele gewinnen wird.

Die Länge eines durchschnittlichen Spiels ist mit 20 bis 30 Zügen (Anhang, Tab. B.2) sehr kurz. Gegen PPAFP gewinnt UCT ebenfalls jeweils das schnellste Spiel, in nur drei bis sieben Zügen. Um ein Spiel mit einer Länge von drei Zügen zu erreichen, muss eine Seite einen Zug ausführen, welcher aktiv dem Gegner hilft. Noch mehr als bei PPAP unterscheidet sich die Anzahl der durchgeführten Playouts für PPAFP und UCT stark: UCT erreicht etwa hundertfünfzig- bis zweihunderttausend Playouts, PPAFP nie mehr als viertausend (Tab. 4.4).

Für folgende Versuche wird für PPAFP ein α von 0,32 gewählt. Dieser Wert zeigt die beste Gewinnrate. Die Elo-Differenz ist für 2,56 besser, aber da der Ursprung der höheren Anzahl an Unentschieden nicht klar ist, wird der Wert nach der Gewinnrate ausgewählt.

Metrik	UCT	PPAFP	UCT	PPAFP	UCT	PPAFP	UCT	PPAFP	UCT	PPAFP	UCT	PPAFP
α	0,04		0,16		0,32		0,64		1		2,56	
Rate [%]	93,33	5	85	8,33	80	**15**	90	8,33	83,33	3,33	75	13,33
Playouts	188912	1941	191568	3593	184490	3264	211290	2907	159560	3527	165500	3960

Tabelle 4.4.: UCT vs. PPAFP mit variierendem α nach 60 Spielen (gekürzt).
$p = 2$, $c = 1{,}414$, Zeit pro Zug = 2s, max. Zugzahl pro Spiel = 120.

4.4. k-Variation

Es wird im folgenden Vergleich nur der Parameter k variiert, der die Zuggewichte (und damit Wahrscheinlichkeiten) in der Zugauswahl im Playout von MAST reguliert.

Die Werte für k wurden aus dem Artikel von Cazenave[4] übernommen, bzw. wurden solche Werte ausgewählt, die in seinen Versuchen die besten Ergebnisse für verschiedene Spiele lieferten.

CPU (1. Messung): Intel(R) Xeon(R) Gold 6326 CPU @ 2.90GHz

CPU (2. Messung): Intel(R) Xeon(R) Gold 6240L CPU @ 2.60GHz

[1] Die logarithmische Skalierung der x-Achse dient lediglich zur besseren Darstellung, da sich sonst die Beschriftungen überschneiden. Die an der x-Achse markierten Werte sind exakt die für den Versuch verwendeten Werte für k.

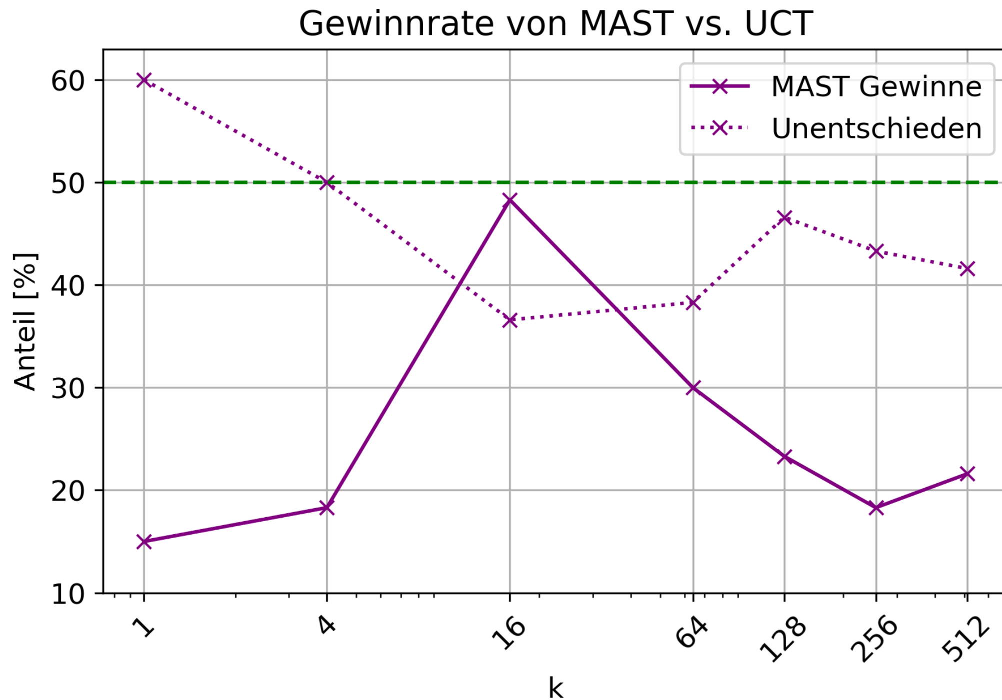

Abbildung 4.4.: Vergleich der Gewinnrate von MAST gegen UCT mit variierendem k nach 60 Spielen.
c = 1,414, Zeit pro Zug = 2s, max. Zugzahl pro Spiel = 120.[1]

Die Abbildung 4.4 zeigt hohe Anteile von Unentschieden für alle getesteten Werte für k, vor allem für den Wert 1. Das lässt nur noch einen kleinen Teil tatsächlich entschiedener Spiele übrig. Für eine gute Statistik reicht das nicht mehr aus, weswegen für k-Werte um den bisher besten gefundenen Wert eine zweite Versuchsreihe durchgeführt wurde. Die maximale Zugzahl pro Spiel wurde hierfür auf 150 erhöht, in der Hoffnung, dass sich der prozentuale Anteil an Unentschieden verringert. Das Ändern dieser Grenze wirkt sich potentiell auf die Gewinnrate und alle davon abhängigen Metriken aus. Es wurden außerdem im zweiten Versuch mehr Spiele insgesamt gespielt, falls die Zahl der Unentschieden nicht mit höherem Zuglimit zurückgeht. So gibt es in jedem Fall eine größere Zahl an entschiedenen Spielen.

In der Abbildung 4.5 wird sichtbar, dass die Anzahl der Unentschieden trotz Verschiebung des maximalen Zuglimits sehr hoch bleibt. Doch für zwei der Werte liegt nun die Gewinnrate von MAST über dem Anteil von Unentschieden. Die Basis von entschiedenen Spielen wird als ausreichend für die statistische Untersuchung gewertet.

In keinem der Tests kann MAST eine Gewinnrate von über 50% erreichen, allerdings zeigt sich bei beiden Messungen ein klarer Peak für den k-Wert 16 mit rund 48% Gewinnrate (Abb. 4.4 / Abb. 4.5). UCT kann erstmals nicht über die Hälfte der Spiele für sich entscheiden. Zwar kann es in der ersten Versuchsreihe für alle Werte für k außer 16 die höhere Gewinnrate und damit eine positive Elo-Differenz erreichen, aber MAST liegt jeweils nur wenige Prozent zurück (Tab. 4.5).

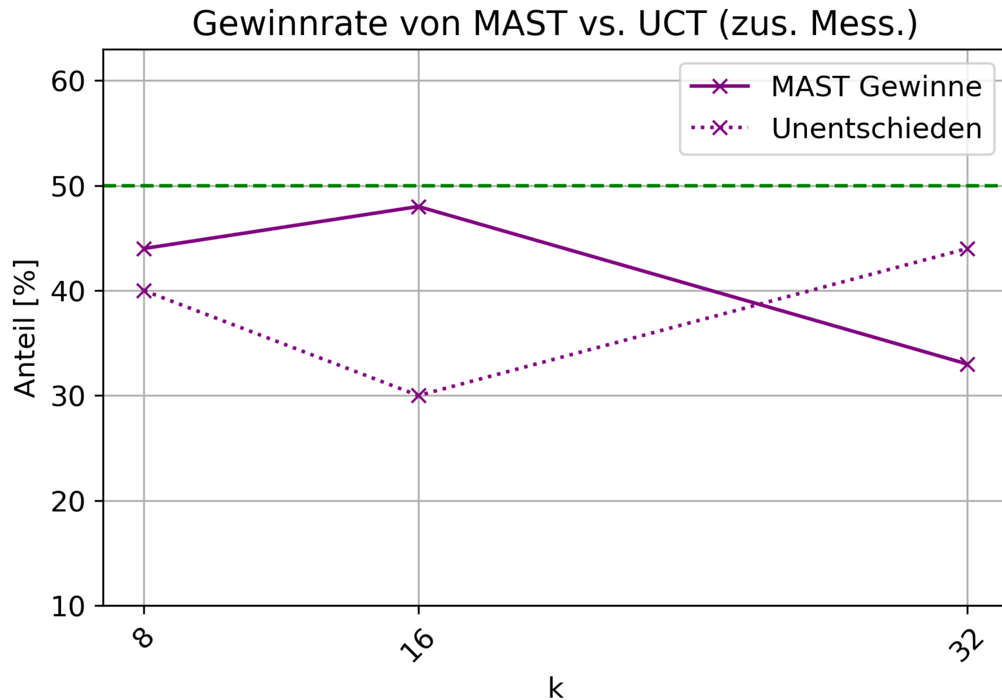

Abbildung 4.5.: Vergleich der Gewinnrate von MAST gegen UCT mit variierendem k nach 100 Spielen (zusätzliche Messungen).
$c = 1{,}414$, Zeit pro Zug = 2s, max. Zugzahl pro Spiel = 150.

Insgesamt sind die Elo-Differenzen deutlich niedriger als in vorherigen Versuchen, vor allem in der ersten Versuchsreihe (Anhang, Tab. C.1). Die LOS liegt erstmals für einen PPA-Algorithmus gegen UCT nicht bei 100% für UCT. In der ersten Versuchsreihe kann MAST eine LOS von 99,94% für $k = 16$ erhalten, im zweiten Versuch sogar eine LOS über 90% für alle getesteten Werte (Anhang, Tab. C.2). Für diese ist MAST also wahrscheinlich UCT in Spielstärke überlegen.

Das schnellste Spiel wird ebenfalls, wie bei PPAP und PPAFP zuvor, für jeden getesteten Wert von UCT gewonnen. Diese Spiele sind wieder sehr kurz, mit zwischen 3 und 14 Zügen jedoch im Schnitt etwas länger als bei PPAP und PPAFP (Anhang, Tab. C.1 / Tab. C.2). Die durchschnittliche Spiellänge liegt mit 71 bis 104 Zügen deutlich näher am Durchschnitt für Xiangqi als in vorherigen Versuchen.

In der Anzahl der Playouts besteht auch in diesen Versuchen eine Differenz zwischen den Algorithmen, diese ist aber deutlich kleiner als im Vergleich von UCT mit PPAP und PPAFP. MAST konnte im Schnitt etwas mehr als die Hälfte der Playouts von UCT erreichen (Tab. 4.5 / Tab. 4.6).

Auch wenn für $k = 8$ eine etwas höhere Elo-Differenz und LOS errechnet wurden (Tab. C.2), wird für folgende Versuche der Wert 16 für k gewählt. Dieser Wert erreicht nicht nur in beiden Versuchsreihen die beste Gewinnrate, sondern ist auch der einzige Wert bis hierhin, mit welchem ein PPA-Algorithmus UCT im schnellsten Spiel schlagen konnte.

Metrik	UCT	MAST	UCT	MAST	UCT	MAST	UCT	MAST	UCT	MAST	UCT	MAST	UCT	MAST
k	1		4		16		64		128		256		512	
Rate [%]	25	15	31,67	18,33	15	**48,33**	31,67	30	30	23,33	38,33	18,33	36,67	21,67
Playouts	97264	34628	107517	48381	98796	71056	93685	63334	90309	59511	99486	60336	87228	55946
LOS [%]	89	11,03	92,79	7,21	0,06	99,94	56,53	43,47	76,03	23,98	98,02	1,98	93,59	6,41

Tabelle 4.5.: UCT vs. MAST mit variierendem k nach 60 Spielen (gekürzt).
c = 1,414, Zeit pro Zug = 2s, max. Zugzahl pro Spiel = 120.

Metrik	UCT	MAST	UCT	MAST	UCT	MAST
k	8		16		32	
Rate [%]	16	44	22	48	23	33
Playouts	71189	43019	61986	39682	60855	37924

Tabelle 4.6.: UCT vs. MAST mit variierendem k nach 100 Spielen (zusätzliche Messungen, gekürzt).
c = 1,414, Zeit pro Zug = 2s, max. Zugzahl pro Spiel = 150.

4.5. c-Variation

Die Anpassung der Explorationskonstante c ist nicht spezifisch für PPA-Algorithmen, sondern wird auch für Standard-UCT vorgenommen. Eine detaillierte Beschreibung der Ergebnisse befindet sich in Anhang D. An dieser Stelle wird nur eine kurze Übersicht der Ergebnisse geliefert.

Für die c-Variation wurden zwei Versuchsreihen durchgeführt, da sich in der ersten eine hohe Anzahl von Unentschieden zeigte. Im zweiten Versuch wurden Werte für c getestet, die entweder sehr hohe Anteile von Unentschieden aufwiesen und/oder vielversprechend im ersten Versuch schienen. Die je höchste Gewinnrate für PPAP, PPAFP und MAST wurde in der zweiten Versuchsreihe erreicht. Die entsprechenden Gewinnraten und Unentschieden gegen UCT aus der zweiten Versuchsreihe finden sich in Abbildung 4.6.

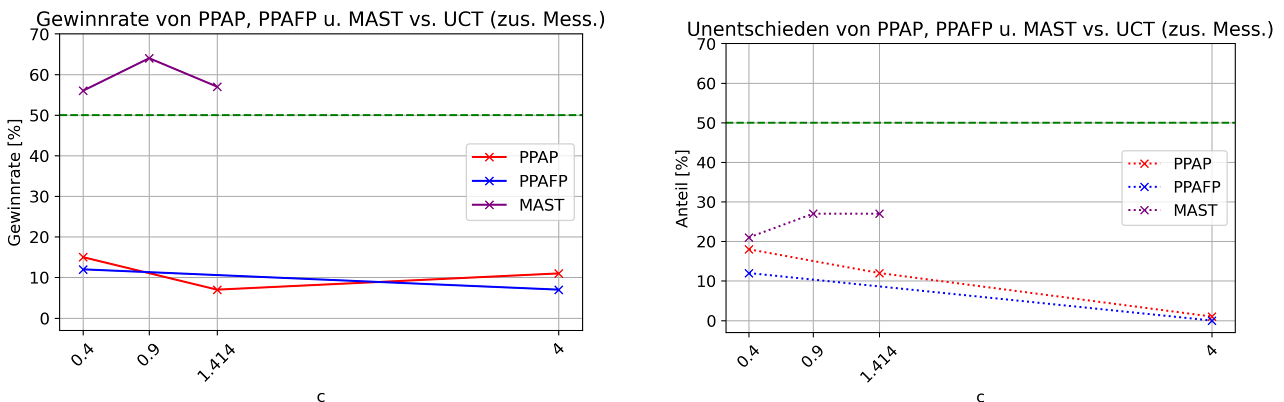

Abbildung 4.6.: Vergleich der Gewinnraten und Unentschieden von PPAP, PPAFP u. MAST gegen UCT mit variierendem c nach 100 Spielen (zusätzliche Messungen).
p (PPAP) = 1, p (PPAFP) = 2, α (PPAP) = 0,04, α (PPAFP) = 0,32, k = 16, Zeit pro Zug = 2s, max. Zugzahl pro Spiel = 170.

Es ist zu sehen, dass nur MAST über 50% der Spiele gegen UCT gewinnen kann. Die beste Gewinnrate findet sich für PPAP und PPAFP bei einem c von 0,4 und für MAST bei 0,9. Das sind jeweils die Werte, die in folgenden Versuchen für die PPA-Algorithmen für c verwendet werden.

4.6. Zeit pro Zug Variation

Es wurde im folgenden Vergleich nur die gegebene Zeit pro Zugsuche variiert.

CPU: AMD EPYC 7453 28-Core Processor

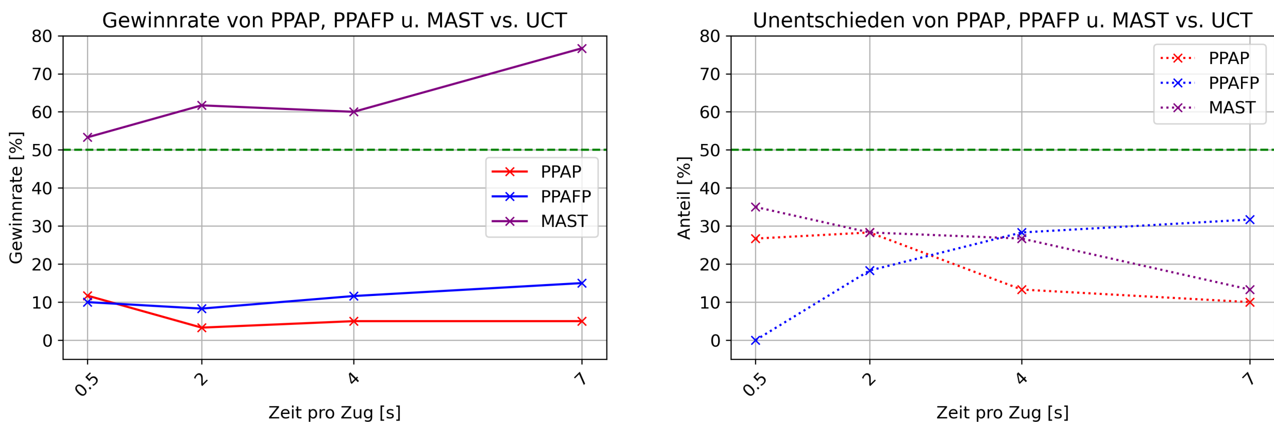

Abbildung 4.7.: Vergleich der Gewinnraten und Unentschieden von PPAP, PPAFP u. MAST gegen UCT mit variierender Zeit pro Zug nach 60 Spielen.
p (PPAP) = 1, p (PPAFP) = 2, α (PPAP) = 0,04, α (PPAFP) = 0,32, k = 16, c (PPAP/PPAFP) = 0,4, c (MAST) = 0,9, max. Zugzahl pro Spiel = 150.

In der Abbildung 4.7 ist zu sehen, dass, wie in vorherigen Versuchen, PPAP und PPAFP keine hohe Gewinnrate gegen UCT erzielen können, während MAST mit zunehmender Bedenkzeit noch besser abschneidet. Aber auch PPAFP gewinnt mit mehr Zeit tendenziell einen größeren Anteil an Spielen. PPAP dagegen schneidet bei nur 0,5s Bedenkzeit am besten ab.

Der Trend bei den Unentschieden ist für PPAFP proportional zur Zeit, für MAST antiproportional. Der Verlauf bei PPAP ist etwas stufiger, mit den wenigsten Unentschieden bei einer Bedenkzeit von 7s.

In der Tabelle E.1 (Anhang) ist zu erkennen, dass die Elo-Differenz im Vergleich von PPAP und UCT mit mehr Zeit wächst. So gewinnt PPAP nie mehr als ein paar Spiele. UCT gewinnt außerdem jeweils das schnellste Spiel und hat in jedem getesteten Fall eine LOS von rund 100%. Auch die durchschnittliche Spiellänge fällt etwas mit längerer Bedenkzeit.

Die Anzahl der Unentschieden und Züge pro Spiel geht mit zunehmender Bedenkzeit zurück (Tab. 4.7). Bei 7s gewinnt UCT 85% aller Spiele. Gleichzeitig wird der relative Unterschied der Anzahl an Playouts pro Zug tendenziell größer.

Metrik	UCT	PPAP	UCT	PPAP	UCT	PPAP	UCT	PPAP
Zeit/Zug [s]	0,5		2		4		7	
Rate [%]	61,67	11,67	68,33	3,33	81,67	5	85	5
Playouts	34840	2226	132425	9944	303018	17435	551805	26134
Unent.	16		17		8		6	
Züge/Sp.	89		83		66		67	

Tabelle 4.7.: UCT vs. PPAP mit variierender Zeit pro Zug nach 60 Spielen (gekürzt). $p = 1$, $\alpha = 0{,}04$, $c = 0{,}4$, max. Zugzahl pro Spiel = 150.

Für PPAFP ist in Abbildung 4.7 ein leichter Aufwärtstrend der Gewinnrate mit steigender Bedenkzeit zu erkennen. Deutlich schneller als die Gewinnrate steigt für diesen Algorithmus aber der Anteil der Unentschieden.

In der Tabelle 4.8 ist außerdem eine sehr deutliche Zunahme an durchschnittlichen Zügen pro Spiel proportional zur Bedenkzeit zu sehen. Während ein Spiel für 0,5s mit nur 21 Zügen sehr kurz ist, entspricht die Länge bei 7s fast dem Durchschnitt für Xiangqi.

Bei 7s ist die Elo-Differenz zwischen den beiden Algorithmen am kleinsten und die LOS liegt nicht mehr ganz bei 100% für UCT (Anhang, Tab. E.2). UCT gewinnt hier nur noch knapp über die Hälfte der Spiele. Trotzdem kann PPAFP nicht mehr als 15% der Spiele für sich entscheiden.

Metrik	UCT	PPAFP	UCT	PPAFP	UCT	PPAFP	UCT	PPAFP
Zeit/Zug [s]	0,5		2		4		7	
Rate [%]	90	10	73,33	8,33	60	11,67	53,33	**15**
Playouts	60698	592	170821	3384	303134	10539	526749	22920
Unent.	0		11		17		19	
Züge/Sp.	21		61		80		90	

Tabelle 4.8.: UCT vs. PPAFP mit variierender Zeit pro Zug nach 60 Spielen (gekürzt). $p = 2$, $\alpha = 0{,}32$, $c = 0{,}4$, max. Zugzahl pro Spiel = 150.

Für MAST ist eine deutliche Steigerung der Gewinnrate mit der Zeit zu erkennen (Abb. 4.7). Bei 7s pro Zug liegt diese fast bei 80%. Die Rate der Unentschieden sinkt dagegen mit steigender Zeit.

Metrik	UCT	MAST	UCT	MAST	UCT	MAST	UCT	MAST
Zeit/Zug [s]	0,5		2		4		7	
Rate [%]	11,67	53,33	10	61,67	13,33	60	10	**76,67**
Playouts	29366	19979	113451	81631	224032	174314	420586	372427
Unent.	21		17		16		8	
Züge/Sp.	101		90		82		72	

Tabelle 4.9.: UCT vs. MAST mit variierender Zeit pro Zug nach 60 Spielen (gekürzt). $k = 16$, $c = 0{,}9$, max. Zugzahl pro Spiel = 150.

Tabelle 4.9 zeigt diesen Trend genauer. Für jeden getesteten Wert erreicht MAST eine Gewinnrate von über 50% und eine LOS von rund 100% (Anhang, Tab. E.3). Der Anteil von Unentschieden und die durchschnittliche Länge eines Spiels sinken dabei mit mehr Zeit (Tab. 4.9). Mit jeder Messung ist der Unterschied der Playoutzahl zwischen UCT und MAST relativ gesehen kleiner. Dies ist außerdem die erste Messreihe, in welcher MAST in drei von vier Versuchen das schnellste Spiel gegen UCT gewinnt. Der Trend der Unentschieden und Spiellängen geht hier im Vergleich zu PPAFP in die entgegengesetzte Richtung.

In einer zweiten Versuchsreihe wurden Werte mit großem Anteil von Unentschieden wiederholt und eine Messung mit 10s pro Zug hinzugefügt. Es wurden die selben Parameter verwendet, aber 100 Spiele pro Messung ausgeführt, um mit dem gleichen Anteil von Unentschieden trotzdem mehr entschiedene Spiele zu haben. Die maximale Zugzahl pro Spiel wurde nicht hochgesetzt, da dies auch die Dauer der Versuche und den benötigten Speicherplatz deutlich erhöht hätte.

CPU (PPAP): Intel(R) Xeon(R) Gold 6326 CPU @ 2.90GHz

CPU (PPAFP): Intel(R) Xeon(R) Gold 6240L CPU @ 2.60GHz

CPU (MAST): Intel(R) Xeon(R) Platinum 8480CL

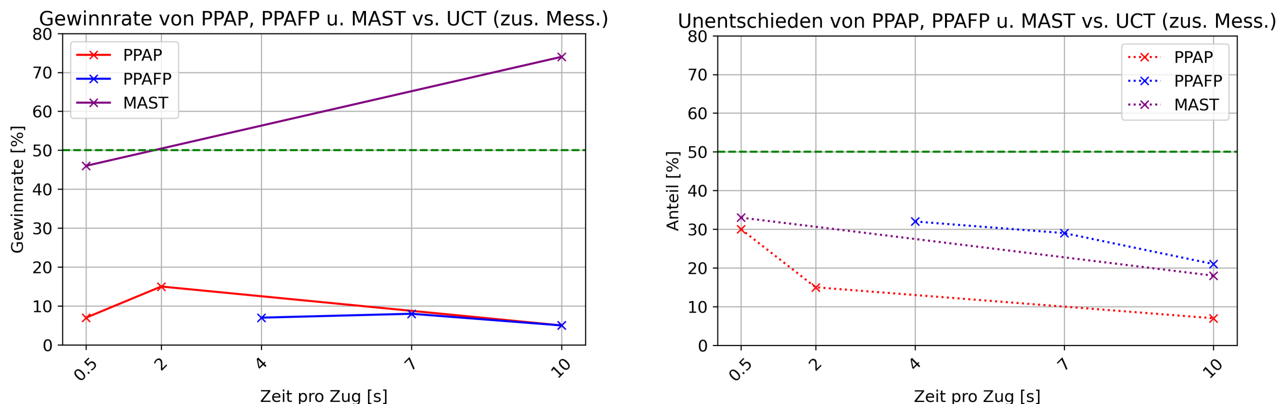

Abbildung 4.8.: Vergleich der Gewinnraten und Unentschieden von PPAP, PPAFP u. MAST gegen UCT mit variierender Zeit pro Zug (zusätzliche Messungen). p (PPAP) = 1, p (PPAFP) = 2, α (PPAP) = 0,04, α (PPAFP) = 0,32, k = 16, c (PPAP/PPAFP) = 0,4, c (MAST) = 0,9, max. Zugzahl pro Spiel = 150.

Die Abbildung 4.8 zeigt nur noch für MAST einen Anstieg der Gewinnrate mit der Zeit. Für PPAP liegt der Peak jetzt bei 2s Bedenkzeit. Die Unentschieden für alle drei Algorithmen nehmen mit steigender Bedenkzeit ab.

Für PPAP ist zu erkennen, dass bei 0,5s mehr Unentschieden vorliegen als zuvor, zugunsten von UCT (Anhang, Tab. E.4). Bei 2s kann PPAP eine größere Gewinnrate erreichen als im letzten Versuch (Tab. 4.10). Für 10s dagegen gewinnt UCT 88% der Spiele, und erreicht eine Elo-Differenz von über 400 (Anhang, Tab. E.4). Es setzt sich der Trend aus der vorherigen Messung fort, dass mit zunehmender Zeit tendenziell die Spiele kürzer und die Gewinnrate von PPAP schlechter wird, während es weniger Unentschieden gibt. Der Unterschied der Playoutzahl ist bei 10s relativ gesehen größer als bei anderen Zeiten (Tab. 4.10).

Metrik	UCT	PPAP	UCT	PPAP	UCT	PPAP
Zeit/Zug [s]	0,5		2		10	
Rate [%]	63	7	70	**15**	88	5
Playouts	31033	1990	140762	12676	785108	41433

Tabelle 4.10.: UCT vs. PPAP mit variierender Zeit pro Zug nach 100 Spielen (zusätzliche Messungen, gekürzt).
$p = 1$, $\alpha = 0{,}04$, $c = 0{,}4$, max. Zugzahl pro Spiel = 150.

In der zusätzlichen Messung schneidet PPAFP leicht schlechter ab als zuvor. Während die absolute Anzahl an gewonnenen Spielen ungefähr gleich ist, liegt der prozentuale Gewinnanteil unter dem zuvor gemessenen (Tab. 4.8 / Tab. 4.11). Auch wenn in der vorherigen Versuchsreihe eine steigende Gewinnrate mit steigender Bedenkzeit beobachtet werden konnte, zeigt der Versuch mit 10s pro Zug ein schlechteres Ergebnis für PPAFP als für die anderen beiden Werte (Tab. 4.11).

Metrik	UCT	PPAFP	UCT	PPAFP	UCT	PPAFP
Zeit/Zug [s]	4		7		10	
Rate [%]	61	7	63	8	74	5
Playouts	210624	5984	294245	10670	405730	9131

Tabelle 4.11.: UCT vs. PPAFP mit variierender Zeit pro Zug nach 100 Spielen (zusätzliche Messungen, gekürzt).
$p = 2$, $\alpha = 0{,}32$, $c = 0{,}4$, max. Zugzahl pro Spiel = 150.

Für jeden hier gemessenen Wert liegt hier die LOS für UCT bei 100% und die Elo-Differenz über 200 (Anhang, Tab. E.5). UCT gewinnt wieder jedes schnellste Spiel. Die durchschnittliche Länge eines Spiels geht mit steigender Zeit leicht zurück.

Es ist anzumerken, dass in dieser zweiten Messung der relative Unterschied zwischen der Anzahl der Playouts für UCT und PPAFP wächst, anstatt sich zu verringern. Vor allem für eine Bedenkzeit von 10s wird dies deutlich, wo die durchschnittliche Anzahl der Playouts für PPAFP sogar unter der mit 7s liegt (Tab. 4.11).

Metrik	UCT	MAST	UCT	MAST
Zeit/Zug [s]	0,5		10	
Rate [%]	21	46	8	74
Playouts	30148	20892	683026	647092

Tabelle 4.12.: UCT vs. MAST mit variierender Zeit pro Zug nach 100 Spielen (zusätzliche Messungen, gekürzt).
$k = 16$, $c = 0{,}9$, max. Zugzahl pro Spiel = 150.

Für MAST liefert die erneute Messung für 0,5s ein insgesamt sehr ähnliches Ergebnis (Tab. 4.12). Für 10s ist, ähnlich wie bei 7s, eine sehr hohe Gewinnrate mit guter Elo-Differenz und

ähnlicher Spiellänge zu beobachten (Anhang, Tab. E.6). Die Anzahl der Playouts ist relativ gesehen ebenfalls ähnlich. Auch hier gewinnt MAST jeweils das schnellste Spiel.

4.7. Feste Playoutzahl

Als Abschluss der Versuchsreihe der UCT-basierten Algorithmen wurde ein weiterer Vergleich durchgeführt, welcher als einziger nicht beiden Algorithmen die gleiche Zeit für die Zugfindung zur Verfügung stellt. Stattdessen wurde eine Grenze von 20000 auszuspielenden Playouts pro Zug gesetzt. Da die Algorithmen unterschiedlich viele Playouts in gegebener Zeit schaffen, entsteht dadurch potentiell ein Vorteil für langsamere Algorithmen, da diese mehr der Ressource Zeit verbrauchen können.

Dieser Vergleich wird durchgeführt, sowohl um die Ergebnisse besser mit Cazenaves Tests[4] vergleichen zu können, als auch um die Qualität eines Playouts zu bewerten.

Es wurde für jede Kombination der UCT-basierten Algorithmen eine Versuchsreihe durchgeführt, mit den bisher besten gefundenen Parameterwerten. Die maximale Zugzahl wurde auf 170 gesetzt, um den Anteil der Unentschieden möglichste gering zu halten und längere Spiele nicht abzuschneiden. Es wurden 100 Spiele pro Vergleich durchgeführt, um selbst mit vielen Unentschieden noch genügend entschiedene Spiele zu haben.

CPU: Intel(R) Xeon(R) Gold 6326 CPU @ 2.90GHz

CPU (PPAP vs. PPAFP): Intel(R) Xeon(R) Gold 6240L CPU @ 2.60GHz

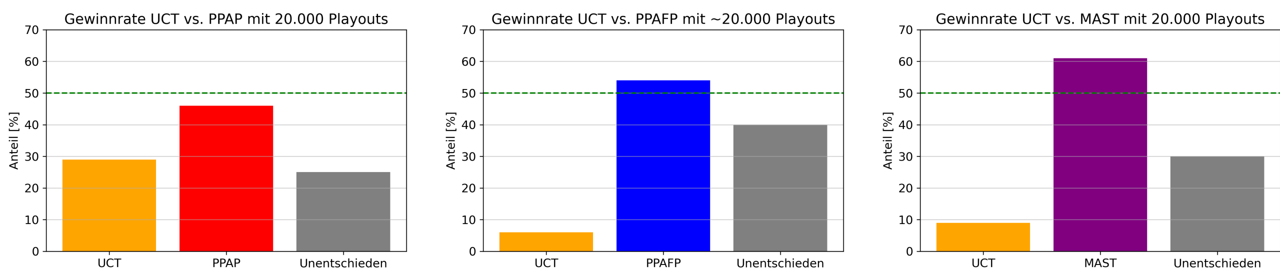

Abbildung 4.9.: Vergleich der Gewinnraten und Unentschieden von UCT gegen PPAP, PPAFP u. MAST mit 20000 Playouts pro Zugsuche nach 100 Spielen[2].
p (PPAP) = 1, p (PPAFP) = 2, α (PPAP) = 0,04, α (PPAFP) = 0,32, k = 16, c (PPAP/PPAFP) = 0,4, c (MAST) = 0,9, max. Zugzahl pro Spiel = 170.

In der Abbildung 4.9 wird sichtbar, dass alle PPA-Algorithmen gut gegen UCT abschneiden, wenn dieses nicht mehr den Vorteil vieler Playouts hat. Dabei schneidet PPAP insgesamt am schlechtesten ab; anders als PPAFP und MAST schafft es nicht, über die Hälfte der Spiele für sich zu entscheiden. Es liegen weniger Unentschieden vor als bei PPAFP und MAST, dafür gewinnt UCT insgesamt mehr Spiele.

Am besten schneidet wieder MAST ab, mit einer Gewinnrate von knapp über 60%. Am schlechtesten ist das Ergebnis für UCT im Vergleich zu PPAFP, wo es nur 6 Spiele gewinnen kann.

[2]PPAFP erreichte trotz einer Bedenkzeit von 30s pro Zug nur eine durchschnittliche Playoutzahl von 19407.

In der Tabelle F.1 (Anhang) zeigt sich weiterhin, dass die Länge eines Spiels je um den Durchschnitt für Xiangqi liegt. UCT kann jeweils keine LOS über 2,5% erhalten und gewinnt nie das schnellste Spiel. Die Elo-Differenzen sind nicht so hoch wie in einigen der vorherigen Versuche, aber immer positiv für den jeweiligen PPA-Algorithmus.

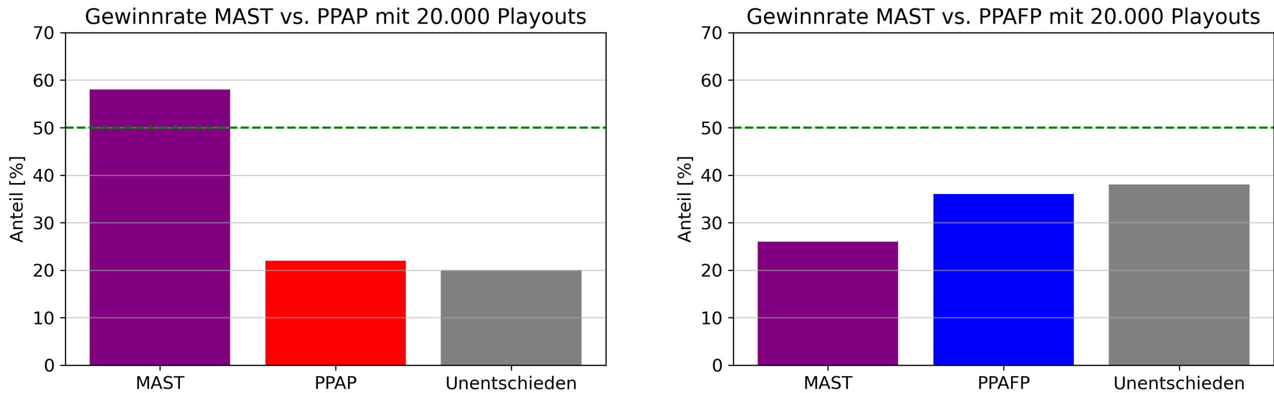

Abbildung 4.10.: Vergleich der Gewinnraten und Unentschieden von MAST gegen PPAP u. PPAFP mit 20000 Playouts nach 100 Spielen.
p (PPAP) = 1, p (PPAFP) = 2, α (PPAP) = 0,04, α (PPAFP) = 0,32, k = 16, c = 0,65, max. Zugzahl pro Spiel = 170.

Wird MAST mit PPAP und PPAFP mit der gleichen Anzahl an Playouts verglichen, entstehen die Ergebnisse in Abbildung 4.10. Für den Parameter c, welchen sich beide Algorithmen teilen, wurde der Wert 0,65 gewählt, welcher mittig zwischen den gefundenen Optima liegt. Es ist zu sehen, dass MAST deutlich gegen PPAP gewinnt, während es gegen PPAFP keinen Vorteil erspielen kann. In dem Vergleich von MAST und PPAFP ist der Anteil der Unentschieden größer als die Gewinnrate eines einzelnen Algorithmus. Anders als in dem Testlauf gegen UCT konnte PPAFP hier die vollen 20000 Playouts erreichen.

In der Tabelle F.2 (Anhang) ist außerdem zu sehen, dass die LOS für MAST gegen PPAP bei rund 100% liegt, während die Elo-Differenz fast 200 erreicht. Trotzdem gewinnt PPAP das schnellste Spiel. Da es sich aber um eine einmalige Beobachtung handelt, kann daraus keine statistische Relevanz abgeleitet werden.

Bei dem Vergleich von MAST und PPAFP ist das Ergebnis ausgeglichener, mit einer geringen Elo-Differenz und einer weniger eindeutigen LOS (Anhang, Tab. F.2). Für beide Versuche war die Länge eines durchschnittlichen Spiels nicht allzu weit von dem Durchschnitt für Xiangqi entfernt, wobei die Spiele von MAST gegen PPAP tendenziell kürzer waren.

Im direkten Vergleich von PPAP und PPAFP wurde der Parameter p auf 1,5 gesetzt und α auf 0,18, jeweils die Mitte zwischen den gefundenen Optima. Es zeigt sich ein klarer Vorteil für PPAFP (Abb. 4.11). PPAFP gewinnt über die Hälfte der Spiele, obwohl es in der gegebenen Zeit im Durchschnitt deutlich weniger als das eigentliche Limit an Playouts erreicht. Das Ergebnis reicht für eine LOS für PPAFP von rund 100% und eine Elo-Differenz von knapp 150

[3] PPAFP erreichte selbst mit 35s Bedenkzeit nur rund 15000 Playouts.

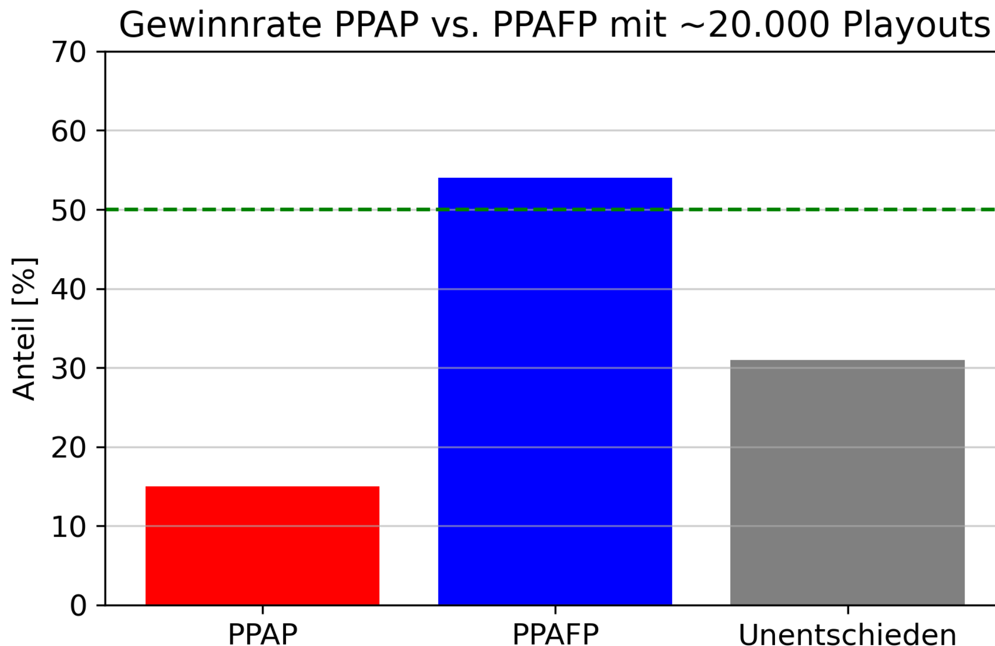

Abbildung 4.11.: Vergleich der Gewinnrate und Unentschieden von PPAP gegen PPAFP mit rund 20000 Playouts[3].
p = 1,5, α = 0,18, c = 0,4, max. Zugzahl pro Spiel = 170.

(Anhang, Tab. F.3). Trotzdem gewinnt PPAP das schnellste Spiel, welches 17 Züge lang war. Die durchschnittliche Spiellänge liegt sehr nah am Durchschnitt für Xiangqi.

4.8. PPA-Algorithmen vs. ABS

In den bisherigen Versuchen hat sich ergeben, dass nur ein gut konfiguriertes MAST gegen UCT gewinnen kann. Dennoch wurden alle PPA-Algorithmen, soweit im Rahmen der durchgeführten Versuche möglich, für Xiangqi optimiert.

Nun da die Konfiguration der PPA-Algorithmen für diese Arbeit abgeschlossen ist, wird der Vergleich gegen ABS durchgeführt. Dazu wird ein festes Zeitbudget pro Zugsuche gesetzt. ABS bekommt keine feste Tiefe, sondern wendet Iterative Deepening an. Für die PPA-Algorithmen gelten die zuvor gefundenen Werte.

CPU: Intel(R) Xeon(R) Gold 6326 CPU @ 2.90GHz

CPU (UCT): Intel(R) Xeon(R) Gold 6240L CPU @ 2.60GHz

In Abbildung 4.12 wird die Gewinnrate von ABS gegen den jeweiligen PPA-Algorithmus geplottet, da die Gewinnraten für die PPA-Algorithmen (fast) durchgehend bei 0% liegen und damit keine aussagekräftige Grafik bilden würden. Ebenfalls in Abbildung 4.12 zu sehen ist der Anteil der Unentschieden.

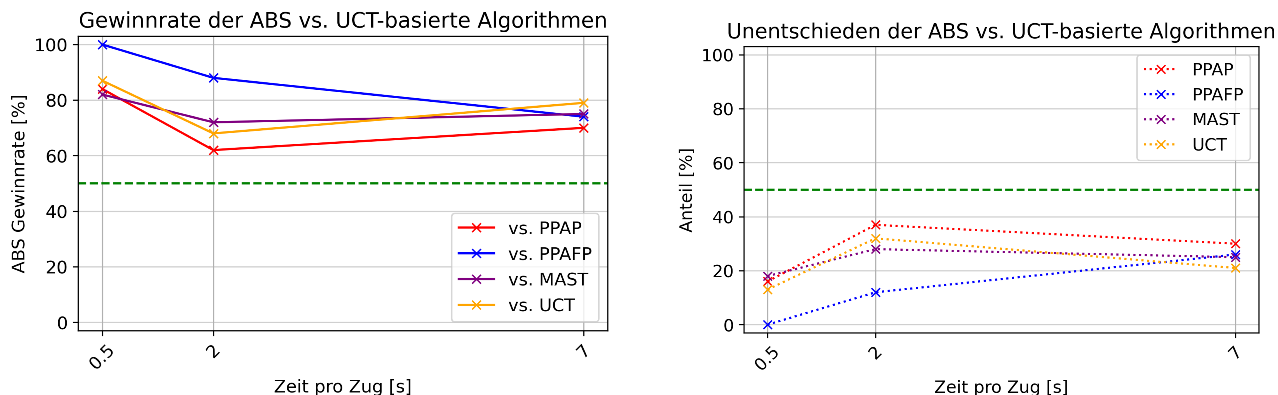

Abbildung 4.12.: Vergleich der Gewinnraten und Unentschieden von ABS gegen PPAP, PPAFP, MAST u. UCT mit variierender Zeit pro Zug nach 100 Spielen.
p (PPAP) = 1, p (PPAFP) = 2, α (PPAP) = 0,04, α (PPAFP) = 0,32, k = 16, c (PPAP/PPAFP) = 0,4, c (MAST) = 0,9, c (UCT) = 1,414, max. Zugzahl pro Spiel = 200.

Sehr deutlich wird, dass keiner der UCT-basierten Algorithmen gut gegen ABS abschneiden kann. Die Gewinnrate liegt für ABS für alle Versuche über 60%, wobei es sich bei dem Rest der Spiele (fast) ausschließlich um Unentschieden handelt.

Das beste Ergebnis, welches PPAP erreicht, ist bei 2s pro Zug, wo der Algorithmus als einziger ein Spiel gegen ABS gewinnen kann und außerdem die höchste Anzahl an Unentschieden erreicht (Tab. 4.13). Damit ist die Elo-Differenz hier am geringsten, mit über 240 aber immer noch sehr hoch (Anhang, Tab. G.1).

PPAFP schneidet für 0,5s pro Zug mit Abstand am schlechtesten ab und ABS gewinnt jedes einzelne Spiel (Tab. 4.13). Die durchschnittliche Länge eines Spiels liegt bei nur 25 Zügen. PPAFP erreicht mit 0,5s Bedenkzeit deutlich weniger als tausend Playouts pro Zugsuche. Die Elo-Differenz ist für einen Fall mit 100-prozentiger Gewinnrate nicht definiert, da eine Division durch Null erfolgen würde (Anhang, Tab. G.1). Mit zunehmender Bedenkzeit pro Zug kann PPAFP zwar immer noch kein Spiel für sich entscheiden, es erreicht aber mehr Playouts, längere Spiele und eine höhere Rate an Unentschieden.

MAST kann in diesem Vergleich nicht so gut abschneiden, wie in den vorherigen Versuchen (Tab. 4.13). Auch für MAST wurde die höchste Zahl an Unentschieden und damit die niedrigste Elo-Differenz für eine Bedenkzeit von 2s erreicht. Die Länge eines Spiels liegt vor allem für 2s und 7s sehr nahe am Durchschnitt für Xiangqi.

Für UCT lief die Messung auf einem anderen Server des Clusters, weshalb die Playouts in der Tabelle 4.13 nicht direkt mit denen der anderen Algorithmen zu vergleichen sind. c wurde für den Versuch auf den Defaultwert $\sqrt{2}$ gesetzt. Auch hier sind die längste Spiellänge und höchste Zahl der Unentschieden bei 2s Bedenkzeit zu sehen.

Für ausnahmslos alle getesteten Algorithmen liegt eine LOS von 100% für ABS vor, daher findet sich dieser Wert nicht in den Tabellen. Da nur ABS Spiele eindeutig für sich entscheiden konnte – mit einer Ausnahme – wurde das schnellste Spiel ebenfalls immer von ABS gewonnen,

Metrik	PPAP			PPAFP			MAST			UCT		
Zeit/Zug [s]	0,5	2	7	0,5	2	7	0,5	2	7	0,5	2	7
Rate [%]	0	**1**	0	0	0	0	0	0	0	0	0	0
Unent. [%]	16	37	30	0	12	26	18	28	25	13	32	21
Playouts	14923	45734	112786	612	3648	27431	26999	167428	855265	21752	95964	601950
Züge/Sp.	64	108	97	25	63	91	74	93	94	60	96	82

Tabelle 4.13.: ABS vs. PPAP, PPAFP, MAST u. UCT mit variierender Zeit pro Zug nach 100 Spielen (gekürzt).
p (PPAP) = 1, p (PPAFP) = 2, α (PPAP) = 0,04, α (PPAFP) = 0,32, k = 16, c (PPAP/PPAFP) = 0,4, c (MAST) = 0,9, c (UCT) = 1,414, max. Zugzahl pro Spiel = 200.

mit Längen von sechs bis 22 Zügen (nicht in Tabellen abgebildet). In den meisten Fällen lag die Länge des schnellsten gewonnenen Spiels im unteren Zehnerbereich.

Die letzte Tabelle G.2 (Anhang) zeigt den gleichen Versuch von zuvor wiederholt, nur dass dieses Mal in Iterative Deepening das gegebene Zeitlimit durch zwei geteilt wurde.

CPU: Intel(R) Xeon(R) Platinum 8480CL

CPU (PPAFP): Intel(R) Xeon(R) Gold 6240L CPU @ 2.60GHz

Es ist zu sehen, dass ABS eine etwas niedrigere Durchschnittstiefe erreicht, aber immer noch alle bis auf zwei Spiele gewinnen kann (Anhang, Tab. G.2). Interessant ist, dass PPAP in dem Versuch, in dem es gewinnt, das schnellste Spiel mit 16 Zügen für sich entscheiden kann.

Davon abgesehen scheinen die Ergebnisse sehr ähnlich wie zuvor. Die LOS liegt nach wie vor bei 100% für ABS für alle getesteten Konstellationen und Werte.

Es ist außerdem zu sehen, dass die Anzahl der Unentschieden nun auch für UCT linear ansteigt (Anhang, Tab. G.2). Obwohl UCT über zwei Millionen Playouts erreicht, kann es aber kein Spiel für sich entscheiden.

5. Diskussion

Der Simplifizierung halber wurde in dieser Arbeit kein Mechanismus implementiert, welcher Zugwiederholungen außerhalb von Playouts erkennt und verhindert. Es ist daher in den Ergebnissen nicht mehr möglich, die Ursache für ein Unentschieden zu erkennen. Ein Unentschieden kann entweder dadurch entstehen, dass beide verglichenen Algorithmen ungefähr gleich gut spielen und das vorgegebene Zuglimit erreichen, weil keiner die Überhand gewinnen kann oder durch wiederholende Zyklen von Zügen. Die Anzahl der Unentschieden beeinflusst auch die Elo-Differenz, nicht aber die Gewinnrate. Wegen der unklaren Ursache für Unentschieden wird demnach die Gewinnrate als Indiz für Spielstärke der Elo-Differenz vorgezogen. In den meisten Fällen ist jedoch sowieso einer der verglichenen Algorithmen klar überlegen, sodass nicht in Frage steht, welcher Algorithmus besser ist, sondern nur, mit welchem Parameter der unterlegene Algorithmus am wenigsten schlecht abschneidet.

5.1. PPA vs. PPAP

Insgesamt erreicht PPAP eine leicht höhere Gewinnrate als PPA (Tab. 4.1). PPAP zeigt außerdem die Tendenz, weniger Playouts in der gegebenen Zeit zu schaffen. Zusammen deutet das darauf hin, dass PPAP eine qualitativ hochwertigere Policy Tabelle erstellt, da es mit weniger Erkundung eine öfter zum Sieg führende Strategie entwickeln kann. Die niedrigere Playout Zahl ist dabei vermutlich darauf zurückzuführen, dass PPAP in `adapt()` für jeden Zug Zeilen 9–11 in Algorithmus 7 durchführen muss, anstatt nur für die Züge, die zu einem Gewinn geführt haben.

Die Variation des Parameters p im Rahmen der gewählten Werte scheint keinen sehr hohen Einfluss auf den Vergleich der Algorithmen zu haben. Die Gewinnrate von PPAP bewegt sich nur in einem Intervall von 7%, die Länge eines durchschnittlichen Spiels ist sehr ähnlich und die LOS liegt bei über 90% für PPAP für alle getesteten Werte (Tab. 4.1). Es scheint, dass ein Penalty-Wert für schlechte Züge den Algorithmus verbessert, aber um aussagekräftigere Ergebnisse zu der Auswirkung einer Variation von p zu erhalten, hätten vermutlich Werte für p mit einem größeren Abstand voneinander gewählt werden müssen.

In einer praktischen Anwendung scheint es jedoch nicht sinnvoll, p zu groß oder zu klein zu wählen. Für sehr kleine p werden die Gewichte von Zügen sehr schnell gesenkt, sodass vielleicht gute Züge zu früh aussortiert werden, nur weil sie einmal in einem verlorenen Playout gespielt wurden. Für sehr große p dagegen unterscheidet sich PPAP kaum noch von PPA, denn wenn p gegen Unendlich läuft, sollte PPAP zu PPA konvergieren und die Gewinnrate sich 50% annähern.

Für folgende Versuche wurde PPAP mit dem Wert 1 für p gewählt, da dieser die besten Werte für PPAP im Versuch geliefert hat (Tab. 4.1). Die Elo-Differenz von 101 sagt aus, dass PPAP ein nächstes Spiel mit einer Wahrscheinlichkeit von ca. 64% gewinnt. Die LOS von knapp 100% bedeutet, dass PPAP sehr wahrscheinlich der überlegene Algorithmus ist – zumindest in Verbindung mit den restlichen gewählten Parametern. Für ein sichereres Ergebnis müssten PPA und PPAP in einer größeren Anzahl an Spielen verglichen und mehr Parameter variiert werden, das wurde allerdings aus Zeitgründen vernachlässigt. Stattdessen wird sich auf die sehr hohe LOS verlassen, die die Spielzahl bereits mit einbezieht, und im Weiteren nur noch mit PPAP experimentiert, nicht mit PPA.

5.2. PPAF vs. PPAFP

Auf den ersten Blick sieht das Ergebnis dieses Versuchs ähnlich aus wie bei PPA und PPAP. Doch auch wenn die Gewinnrate von PPAFP gegen PPAF etwas über 50% liegt (Abb. 4.2), sind die Ergebnisse mit hoher Wahrscheinlichkeit wenig aussagekräftig. Zu beachten ist hierfür die Länge eines durchschnittlichen Spiels: Tabelle 4.2 zeigt, dass im Schnitt nur 15 Züge gespielt werden. Im Vergleich zu Xiangqis Durchschnitt von 95 Zügen ist das sehr kurz. Ebenfalls schaffen weder PPAF noch PPAFP viele Playouts pro Zug in der gegebenen Zeit. Es werden also wenige Knoten erkundet, was sowohl in der Selektionsphase, als auch für die Policy Tabelle zu einer weniger aussagekräftigen Statistik führt. Bei dem Start einer jeden Suche wird mit einer gleichmäßigen Policy Tabelle begonnen, wodurch effektiv zufällige Playouts entstehen. Beide Algorithmen übersehen wahrscheinlich die meisten Gefahren für den eigenen König und so ist der gewinnende Algorithmus vermutlich der, der zufällig schnell die besseren Züge findet. Das Ergebnis hat wenig mit der tatsächlichen Spielstärke zu tun, was auch erklärt, warum die Gewinnraten so nah an 50% liegen. Das zeigt sich auch in der niedrigen Elo-Differenz, welche für alle Vergleiche unter 35 bleibt (Tab. A.1).

Eine Aussage über den Einfluss von p auf den Vergleich kann daher, wie bei PPA, nur schlecht getroffen werden.

Überraschend ist, dass PPAFP im Durchschnitt eine höhere Anzahl von Playouts pro Zug im Gegensatz zu PPAF abschließen kann, obwohl in PPAFP mehr Code ausgeführt wird. Dies ist ein Unterschied zu PPA/PPAP. Der Grund für diese Beobachtung ist nicht aus den aufgenommenen Ergebnissen erkenntlich. Möglicherweise profitieren PPAF und PPAFP unterschiedlich von Compileroptimierungen oder Cashing (z.B. von Featurecodes).

Obwohl keine klare Überlegenheit für einen der beiden Algorithmen besteht, wurden weitere Experimente mit PPAFP durchgeführt, mit dem Wert 2 für p. Diese Entscheidung basiert auf dem besseren Abschneiden von PPAP im Gegensatz zu PPA (Abb. 4.1) und der Ähnlichkeit von PPA und PPAF. Zudem scheint es sinnvoll, PPAF zu ermöglichen, schneller Gewichte in der Policy Tabelle anpassen zu können, da dieser Algorithmus von allen PPA-Algorithmen mit Abstand am wenigsten Playouts in einer gegebenen Zeit ausführen kann (Tab. G.1). Da PPAF außerdem mehr Zugcodes aufweist als PPA oder MAST, scheint eine schnellere Lenkung der Policy Tabelle besonders wichtig.

Für p wird 2 gewählt, damit PPAFP nicht in die entgegengesetzte Richtung übersteuert und zu schnell Züge als schlecht bewertet, die noch nicht ausreichend erkundet wurden.

5.3. Playoutzahl

In allen durchgeführten Versuchen zwischen den PPA-Algorithmen und UCT ist zu sehen, dass UCT mehr Playouts in gegebener Zeit durchführen kann (z.B. Tab. G.2). Besonders groß ist diese Differenz für UCT vs. PPAP oder PPAFP, wo die PPA-Algorithmen meistens deutlich weniger als 10% der Playoutzahl von UCT erreichen können. Die Diskrepanz ist vermutlich damit zu begründen, dass UCT lediglich eine Zufallszahl generieren und den entsprechenden Zug aus der Liste entnehmen muss, die der Zuggenerator zurückgibt. Für alle PPA-Algorithmen kommt eine zeitaufwändigere Zugauswahl, plus die `adapt()` Funktion nach jedem Playout hinzu. Vor allem PPAP und PPAFP müssen viel Zeit investieren, um nicht nur die Exponenten der Gewichte zu berechnen und proportional dazu eine Zufallszahl zu wählen, sondern auch für jede Playout-Sequenz zweimal alle legalen Züge für jeden besuchten Zustand zu berechnen: einmal in der Zugauswahl, um die Summe aller exponierten Gewichte zu berechnen (Alg. 6, Zeilen 6–7), einmal in `adapt()`, um die Gewichte der anderen Züge anzupassen (Alg. 7, Zeilen 9–11). Das steigert den Zeitaufwand für ein einzelnes Playout enorm.

Für PPAFP kommt zusätzlich die Zeit dazu, die für das Erkennen der Features und Errechnen des Featurecodes benötigt wird, sowohl in den Playouts, als auch in `adapt()`, damit die Policy Tabelle indexiert werden kann.

Durch die Vielzahl an Playouts ist UCT in der Lage, viel mehr Knoten des Suchgraphen zu erkunden, als die PPA-Algorithmen es können. Es findet dadurch wahrscheinlicher gute Züge oder entdeckt Gefahren für den eigenen König. Die Selektion wird gestärkt, da mit mehr Informationen eine bessere Statistik entsteht. Der Versuch in Sektion 4.7 zeigt, dass die PPA-Algorithmen Playouts besser lenken können als der Zufall in UCT, aber ab einem bestimmten Mehr an Playouts insgesamt scheinen sie nicht mehr in der Lage zu sein, den Nachteil weniger Playouts durch bessere Playouts auszugleichen.

Dabei ist zu beobachten, dass PPAFP etwas besser als oder gleich gut wie PPAP gegen UCT abschneidet (Tab. 4.3 / Tab. 4.4). PPAFP erzeugt eine bessere Policy Tabelle (Abb. 4.11) und im Vergleich zu den zehntausenden Playouts, die UCT durchführt, macht es scheinbar kaum noch einen Unterschied, ob der gegnerische Algorithmus nun acht- oder zweitausend Playouts ausführen kann. Dass sich diese Tendenz in mehreren Versuchsreihen zeigt, scheint zu bestätigen, dass das bessere Verhältnis von Playoutzahl und Gewinnrate gegen UCT von PPAFP gegenüber PPAP nicht nur auf Zufall basiert.

Wahrscheinlich ist diese Differenz in der Playoutzahl der Hauptgrund, warum PPAP und PPAFP in keinem der fairen Versuche mit UCT mithalten können.

Es sei auch auf das jeweils schnellste Spiel hingewiesen, welches UCT in fast jedem Fall gegen PPAP und PPAFP gewinnt, in jeweils sehr wenigen Zügen (Tab. B.1 / Tab. B.2). In diesen Spielen bleibt den Algorithmen keine Zeit, viele Knoten zu besuchen und eine belastbare Statistik aufzubauen. Züge werden mehr oder weniger zufällig gespielt. Die höhere Anzahl an Playouts

für UCT bedeutet, dass es eine höhere Chance hat, früher Siegerzüge zu entdecken, bzw. auch Gefahren zu erkennen. PPAP und PPAFP dagegen können sich nicht verteidigen, wenn sie die Bedrohung nicht sehen. Je größer die Differenz der Playouts, desto höher also auch die Chance, dass UCT das schnellste Spiel gewinnt.

Anders verhält sich die Situation bei MAST, welches oft um die Hälfte der Playoutzahl von UCT erreicht (z.B. Tab. 4.5). MAST hat eine schnellere `adapt()` Funktion; diese enthält z.B. keine Schleife(n) (Alg. 8). Dadurch muss MAST weniger Abstand in den Playoutzahlen mit seiner Policy Tabelle ausgleichen und kann mit bestimmten Parameterkonstellationen gegen UCT gewinnen.

5.4. α-Variation

Aus der Abbildung 4.3 wird schnell ersichtlich, dass weder PPAP, noch PPAFP mit UCT mit zufälligen Playouts mithalten können, unabhängig vom Wert für α. Das wird ebenfalls bestätigt von den durchschnittlich kurzen Spielen (Tab. B.1 / Tab. B.2), die andeuten, dass UCT klar überlegen ist und relativ schnell die meisten Spiele für sich entscheidet.

Der Hauptgrund für UCTs Überlegenheit wurde bereits in der Sektion 5.3 erläutert.

Die Variation von α scheint für PPAP, wie bei p zuvor, die Ergebnisse nicht extrem zu beeinflussen (Tab. 4.3). Vermutlich ist der Unterschied in der Spielstärke der gegnerischen Algorithmen so groß, dass Änderungen abhängig von α verloren gehen oder nur sehr schwach ausfallen. Da PPAP für jedes α nur zwischen drei und sechs Spiele gewinnt, lässt sich nur schwer eine Aussage treffen, welches α am besten geeignet ist. Letztendlich wurde der Wert 0,04 für PPAP gewählt, aufgrund der höchsten Gewinnrate, und weil der selbe Wert in einigen Spielen in Cazenaves Artikel[4] Erfolg hatte.

Im Spiel mit PPAFP scheint α einen leicht stärkeren Einfluss auf den Spielausgang zu haben. Die Zahl der von PPAFP gewonnenen Spiele liegt zwischen zwei und neun (Tab. B.2), was eine etwas größere Schwankung darstellt. Die niedrige Zahl an Unentschieden und Zügen pro Spiel deutet an, dass Spiele schnell entschieden wurden. Vermutlich wurde jeweils kein großer Suchgraph aufgespannt, aber es scheint, als wäre PPAFP teilweise in der Lage gewesen, schnell genug die Policy Tabelle anzupassen, um gute Züge zu finden, oder aber zufällig gute Züge auszusuchen. Für ein α von 0,32 erreicht PPAFP immerhin eine Gewinnrate von 15%, mehr als PPAP. Wie auch bei PPAP wurde der Wert für α für folgende Versuche nach der Gewinnrate ausgewählt.

Es ist ebenfalls möglich, dass das unterschiedliche Abschneiden von PPAP und PPAFP gegen UCT auch von ihren unterschiedlichen Werten für p abhängt, da schlechte Züge die Differenz ihres Gewichts und $\frac{\alpha}{p}$ zugewiesen bekommen. α beeinflusst also nicht nur, wie viel vermutlich gute Züge gestärkt, sondern auch, wie viel vermutlich schlechte Züge bestraft werden[1].

[1] Um den Zusammenhang besser zu erkennen, könnte z.B. in der Zukunft eine Versuchsreihe durchgeführt werden, in welcher das Verhältnis von α und p variiert wird.

5.5. k-Variation

Anders als bei den bisher getesteten Algorithmen gelingt es MAST, sehr geringe Elo-Differenzen zu erreichen (Tab. C.1). Obwohl die meisten Konfigurationen mit einem Vorteil für UCT enden, ist der Unterschied zwischen beiden Algorithmen deutlich kleiner. Gründe unabhängig von k wurden bereits in Sektion 5.3 erwähnt.

Ein k größer Eins verstärkt Gewichte proportional in der Zugauswahl. Dieser Parameter wurde eingeführt, da MAST immer nur die Gewichte von Zügen anpassen kann, welche in einem Playout vorkommen, nicht wie PPAP und PPAFP auch andere in einem erreichten Zustand legale Züge[4]. Durch das Verstärken der Gewichte werden die Wahrscheinlichkeiten anders verteilt. Es scheint, dass die Anpassung dieser Verteilung einen großen Einfluss auf die Spielstärke von MAST hat: Für $k = 16$ schafft MAST es als erster PPA-Algorithmus, mehr Spiele zu gewinnen als UCT (Tab. 4.5). Der selbe Wert in der zweiten Versuchsreihe zeigt das erste schnellste Spiel bisher, welches nicht von UCT gewonnen wurde (Tab. C.2).

Während die meisten Werte für k relativ ähnliche Ergebnisse liefern (Tab. 4.5), zeigt sich ein deutliches Optimum nahe $k = 16$. Das kristallisiert sich vor allem in der zweiten Versuchsreihe heraus, wo MAST für alle Werte eine hohe LOS erreichen kann (Tab. C.2).

Die durchschnittliche Spiellänge und Zahl der Unentschieden steht dagegen nicht erkennbar im Zusammenhang mit dem Wert für k. Die Spiele scheinen insgesamt ausgeglichener als in den Vergleichen mit PPAP und PPAFP, mit Spiellängen deutlich näher am Durchschnitt für Xiangqi und mehr Unentschieden (Tab. C.1).

Für zukünftige Versuche wurde der Wert 16 für k gewählt, da dieser in beiden Versuchsreihen die beste Gewinnrate erzielt und außerdem der einzige Wert ist, für welchen MAST das schnellste Spiel gewinnen konnte.

5.6. Zuglimit

Während in dem Versuch zu der k-Variation das Heben des maximalen Zuglimits nicht den gewünschten Effekt hatte und der Anteil der Unentschieden ungefähr gleich blieb (Tab. C.1 / Tab. C.2), zeigt es in den Versuchsreihen zur c-Variation eine Wirkung. Für ein c von 0,4 können sowohl PPAP als auch PPAFP mit einem höheren Zuglimit eine bessere Gewinnrate erreichen (Tab. D.1 / Tab. D.4 u. Tab. D.2 / Tab. D.5). PPAP verdreifacht seine Gewinnrate sogar. Ein Teil der Unentschieden im ersten Versuch scheint dementsprechend nicht auf Zyklen, sondern auf vorzeitig abgebrochene Spiele rückführbar.

Längere Spiele werden also tendenziell eher von den PPA-Algorithmen gewonnen als kürzere. Das weist darauf hin, dass PPAP und PPAFP später im Spiel besser gegen UCT ankommen. Ein möglicher Grund dafür ist, dass auch die PPA-Algorithmen nach vielen Zügen einen größeren Suchgraphen aufgespannt haben. Die Selektionsphase ist dadurch aussagekräftiger und basiert weniger auf Zufall. Vielleicht steigert ein weiter aufgespannter Suchgraph die Effektivität von Policy-geleiteten Playouts.

5.7. c-Variation

Für PPAP und PPAFP ist in diesem Versuch sehr gut zu betrachten, dass sich die durchschnittliche Spiellänge antiproportional zu der Größe von c verhält: Je höher der Wert der Explorationskonstante, desto schneller ist ein Spiel entschieden (Tab. D.1 / Tab. D.2). Es liegen auch je weniger Unentschieden vor. Das ist vermutlich darauf zurückzuführen, dass zu wenig die bereits bekannten, guten Züge ausgenutzt werden. Es entstehen Spiele, deren Ausgang darauf basiert, ob ein Algorithmus zufällig einen neuen guten Zug findet. Hier kommt wieder der Vorteil von UCT zur Geltung, eine Vielzahl an Playouts durchführen zu können.

Auch ist ein Abnehmen der Playoutzahl für PPAP und PPAFP mit steigendem c zu beobachten (Tab. D.1 / Tab. D.2). Das liegt vermutlich daran, dass kürzere Wege im Suchgraph, also potentiell solche, die schnell zu einem Gewinn führen, nicht so oft erkundet und stattdessen neue Pfade gesucht werden. Sind Zugsequenzen im Playout länger, können weniger Playouts insgesamt durchgeführt werden.

Für PPAP und PPAFP ist es mit großer Explorationskonstante schwieriger, eine gute Policy Tabelle zu konfigurieren. Wird die Suche weiter gestreut, werden einzelne Züge nicht oft genug gespielt, um relevante statistische Daten zu sammeln, und sie erhalten nur die Nachteile (ein größerer Zeitverbrauch) und nicht die Vorteile (eine gute Policy Tabelle zur Playout-Lenkung) ihrer Implementierung. Dadurch kann UCT in den meisten Fällen das Spiel für sich entscheiden.

In der zweiten Versuchsreihe ist für PPAP außerdem ein (vergleichsweise) guter Gewinnsatz für $c = 4$ zu erkennen. Aber hier sind die durchschnittlichen Spiele sehr kurz. Beides zusammen deutet weniger darauf hin, dass $c = 4$ ein günstiger Wert für PPAP ist, und mehr darauf, dass es sich um eine ungünstige Zahl für UCT handelt, da beide die gleiche Konstante c nutzen. Der Fokus liegt zu sehr bei der Exploration und bereits gefundene gute Züge werden nicht ausreichend ausgenutzt.

Für MAST ist kein direkter Zusammenhang zwischen c und der Spiellänge, noch der Zahl der Unentschieden zu erkennen (Tab. D.3).

Die Tabelle D.6 zeigt, dass MAST für jeden der Werte für c in der verfeinerten Messung eine Gewinnrate von über 50% hat, und UCT jeweils nur wenige Spiele für sich entscheiden kann. Die durchschnittlichen Spiellängen liegen sehr nah am Durchschnitt für Xiangqi.

Dass UCT gegen MAST bei höheren c vergleichsweise besser abschneidet, könnte darin begründet sein, dass mit steigendem c die statistische Datensammlung (für beide Algorithmen) ab einem bestimmten Punkt nicht mehr (gut) funktioniert. Es wird immer zufälliger, welche Seite das Spiel gewinnt. Die Hypothese ist, dass mit genügend großem c die Elo-Differenz gegen Null konvergiert. So kann auch das bessere Abschneiden von PPAP für $c = 4$ (Tab. D.4) begründet werden.

Wieder ist zu beobachten, dass, mit nur zwei Ausnahmen, UCT immer das schnellste Spiel gewinnt, egal wie hoch sonst die Gewinnrate des Algorithmus ist (z.B. Tab. D.6). Das wird wieder daran liegen, dass vor allem kurze Spiele relativ zufällig sind, weil noch keine (belastbare) Statistik vorliegt.

Insgesamt scheinen im Kontext von Xiangqi alle Algorithmen von kleineren Werten für c zu profitieren.

5.8. Zeit pro Zug Variation

Im Vergleich mit verschiedenen Zeiten wäre zu erwarten gewesen, dass der relative Unterschied zwischen der Anzahl der Playouts von UCT und dem jeweils anderen Algorithmus schrumpft, und damit alle PPA-Algorithmen vergleichsweise besser abschneiden.

Für PPAP ist dies nicht der Fall: Der relative Abstand der Playoutzahl verändert sich wenig und auch nicht linear. Er scheint außerdem nicht proportional mit der Gewinnrate zusammenzuhängen. In den Tabellen 4.7 und 4.10 ist zu sehen, dass mit mehr Zeit der Vorteil von UCT tendenziell weiter wächst. PPAP kann in beiden Versuchsreihen nur zweimal eine Gewinnrate von über 10% erhalten. Da der relative Unterschied zwischen der Anzahl der Playouts sich kaum verändert, hat UCT noch mehr Zeit, durch schiere Quantität eine gute Statistik zu erstellen, während PPAPs Policy Tabelle scheinbar keine ausreichend qualitativ hochwertigen Playouts leitet, um eine höhere Gewinnchance zu erhalten. Das zeigt sich auch in der durchschnittlichen Länge eines Spiels, die mit zunehmender Zeit geringer wird (Tab. E.4): die Spiele sind also schneller entschieden, meist zugunsten von UCT. Die Ergebnisse lassen nicht vermuten, dass mit genügend Bedenkzeit PPAP mit UCT gleichziehen kann.

PPAFP kann für jede Zeit nur einen Bruchteil der Playouts erreichen, die UCT ausführt, aber der relative Unterschied nimmt in der ersten Versuchsreihe um ein paar Prozent ab (Tab. 4.8). Diese kleine Steigerung, bzw. die absolut höhere Zahl an Playouts für PPAFP zeigt eine deutliche Änderung: die Elo-Differenz sinkt mit steigender Bedenkzeit und die Spiele werden durchschnittlich deutlich länger (Tab. E.2), was auf ein ausgeglicheneres Spiel hinweist. Je länger die Bedenkzeit, desto besser schneidet PPAFP tendenziell gegen UCT ab. Es scheint als ob, anders als bei PPAP, PPAFP die Policy Tabelle mit mehr Playouts genügend verbessern kann, um etwas zu UCT aufzuholen. Das liegt vermutlich daran, dass ab einem bestimmten Punkt der Nutzen der erstellten Policy Tabelle über den Nutzen von einem Mehr an Playouts hinausgeht. Es kann vermutet werden, dass mit einem sehr großen Zeitbudget PPAFP irgendwann in der Lage ist, UCT zu schlagen.

Die zusätzliche Messung zu PPAFP zeichnet ein anderes Bild (Tab. E.5). Der verwendete Server enthält eine andere CPU als in der ersten Messung, was den Unterschied in der Anzahl der Playouts insgesamt erklärt. Unklar ist, wieso PPAFP mit 10s Zeit weniger Playouts schafft, als mit 7s. Mögliche Erklärungen sind, dass zufällig Züge so gewählt werden, dass sehr lange Playouts entstehen. Je länger die einzelnen Playouts, desto weniger können durchgeführt werden. Eine temporäre Verlangsamung des Servers (z.B. aufgrund hoher Last) scheint eher unwahrscheinlich, da die Anzahl der Playouts für UCT wie erwartet deutlich steigt. Der genaue Grund für die niedrige Anzahl an Playouts für PPAFP in dem Versuch mit 10s Bedenkzeit ist aus den aufgezeichneten Ergebnissen nicht erkenntlich. Im Lichte dessen, dass die Differenz der Playoutzahl beider Algorithmen größer wird, ergibt es Sinn, dass PPAFP im zweiten Versuch mit längerer Bedenkzeit schlechter abschneidet als zuvor.

Für MAST trifft die am Anfang der Sektion genannte Vermutung ebenfalls zu: der relative Unterschied zwischen der Zahl der Playouts wird immer geringer, je mehr Zeit für einen Zug zur Verfügung steht (Tab. 4.9 / Tab. 4.12). Dadurch kann UCT kaum noch Spiele für sich entscheiden und die Gewinnrate von MAST steigt fast linear mit der Zeit. Dass gleichzeitig die Anzahl der Unentschieden und die Länge eines Spiels sinken (Tab. E.3), zeigt, dass MAST die Spiele deutlich schneller für sich entscheiden kann. MAST scheint also auch in frühen Spielphasen nun besser abzuschneiden, während die Vermutung nahe liegt, dass es in vorherigen Versuchen mit weniger Zeit tendenziell erst ab einer höheren Zugzahl die Überhand gewinnen konnte. Das wird auch der Grund sein, warum MAST in diesen Versuchsreihen in den meisten Fällen das schnellste Spiel für sich entscheiden kann, obwohl dieses in vorherigen Versuchen beinahe ausnahmslos von UCT gewonnen wurde.

Insgesamt scheint sich die Vermutung zu bestätigen, dass eine kleinere relative Differenz in der Anzahl der Playouts positiv für den jeweiligen PPA-Algorithmus auswirkt. Theoretisch sollte diese Differenz mit zunehmender Bedenkzeit sinken. Wie aber vor allem in der Tabelle E.5 zu sehen ist, ist das nicht zwingend der Fall, z.B. aufgrund von schwankenden Laufzeiten für den Code oder unterschiedlich langen Playouts.

5.9. Feste Playoutzahl

Die in der vorherigen Sektion aufgestellte Vermutung, dass mit einer ausreichend niedrigen Differenz in der Zahl der Playouts jeweils der PPA-Algorithmus den Großteil der Spiele für sich entscheiden kann, bestätigt sich in dieser Versuchsreihe.

Für diesen Vergleich wurde eine feste Playoutzahl bestimmt, die zu erreichen ist, und nach welcher die Zugsuche abgebrochen wird, auch wenn noch Zeit zur Verfügung steht. Dadurch entsteht ein unausgewogener Vergleich, denn langsamere Algorithmen wie PPAFP können mehr der Ressource Zeit nutzen. Es ist zu sehen, dass alle PPA-Algorithmen sehr gut gegen UCT mit zufälligen Playouts abschneiden (Tab. F.1). UCT kann sich keinen Vorteil durch eine Vielzahl von Playouts verschaffen und dieser Versuch bestätigt, dass die PPA-Algorithmen einen höheren statistischen Wert aus den durchgeführten Playouts ziehen. Wo UCT nur die Besuchs- und Gewinnanzahl vermerkt und in der Selektionsphase nutzt, haben die anderen Algorithmen jeweils die Policy Tabelle, welche nach jedem Playout angepasst wird. Ein Playout geleitet von dieser Tabelle hat einen qualitativ höheren Wert als ein zufälliges.

Dieser Versuch ist als einziger gut mit Cazenaves Artikel[4] zu vergleichen, da auch dort nicht eine Zeit pro Zug, sondern eine Anzahl von Playouts festgelegt wurde. Es ist allerdings zu beachten, dass in dem Artikel nur eine Playoutzahl von 8000 festgelegt wurde, während hier mit 20000 Playouts getestet wird. Theoretisch sollte ein Mehr an Playouts eine gute Policy Tabelle weiter verstärken (d.h. konkret die PPA-Algorithmen noch besser gegen UCT abschneiden lassen). In Cazenaves Fall schnitten alle drei hier getesteten Algorithmen (bzw. die Variante ohne Penalty für PPAP und PPAFP) jeweils sehr gut gegen UCT ab, ähnlich wie es in diesem Versuch der Fall ist. In allen vorherigen Versuchen mit gleichem Zeitbudget unterschieden sich die Ergebnisse stark von Cazenaves Artikel, weil die Algorithmen sehr unterschiedlich schnell sind.

Der in diesem Versuch zu betrachtende Vorsprung von PPAFP zu PPAP (Abb. 4.9) ist vergleichbar mit Cazenaves Ergebnissen[4], wo PPAFP häufig am besten abschnitt. Dass MAST allerdings auch mit gleicher Playoutzahl die höchste Gewinnrate gegen UCT erzielt, war nicht zwingend zu erwarten. Möglicherweise braucht PPAFP im Schnitt länger, um ein Spiel für sich zu entscheiden und wurde durch das maximale Zuglimit begrenzt. Die durchschnittliche Länge eines Spiels ist für PPAFP länger als bei MAST (Tab. F.1). Die höhere Anzahl von Unentschieden bei dem Versuch UCT vs. PPAFP und die niedrigere Gewinnrate von UCT hier deuten ebenfalls darauf hin.

Dass UCT jeweils das schnellste Spiel verliert (Tab. F.1), liegt vermutlich daran, dass die anderen Algorithmen geleitet von der Policy Tabelle schnell bessere Entscheidungen treffen können als UCT, auch wenn der Suchgraph noch nicht allzu weit ausgebaut wurde.

Obwohl MAST gegen UCT die höchste Gewinnrate erzielen kann, verliert es im direkten Vergleich knapp gegen PPAFP (Abb. 4.10). Es gibt außerdem eine hohe Anzahl an Unentschieden. Es scheint, dass mit 20000 Playouts PPAFP tendenziell eine bessere Policy Tabelle (für Xiangqi) erstellen kann als MAST. Auch die Ergebnisse in Cazenaves Artikel[4] deuten darauf hin, dass PPAFP für mehrere Spiele tendenziell die beste Policy Tabelle erstellt. Insgesamt scheint es, dass ein guter Teil der Unentschieden bei PPAFP vs. UCT in einem Gewinn für PPAFP geendet hätte, hätte das maximale Zuglimit noch höher gelegen[2].

PPAFP benutzt als einziger PPA-Algorithmus nicht nur die Zugkodierung für die Indexierung der Policy Tabelle, sondern auch zwei binäre Features. Es scheint, dass die Information, ob es sich um einen Schlagzug handelt und ob das Zielfeld vom Gegner bedroht wird, sehr relevant für eine Policy Tabelle in Xiangqi ist. Dadurch, dass es in PPAFP mehr mögliche Zugcodes gibt, müssen tendenziell mehr Playouts durchgeführt werden, um eine aussagekräftige Statistik zu erstellen. Es scheint weiterhin, dass 20000 Playouts dafür ausreichen.

Gegen PPAP schneidet MAST sehr gut ab. Beide Algorithmen nutzen nur die Zugkodierung in der Policy Tabelle, aber MAST scheint die Gewichte je Zug aussagekräftiger zuzuteilen. Da auch in Cazenaves Artikel[4] die Algorithmen unterschiedlich gut in den verschiedenen Spielen abschneiden, liegt hier nahe, dass MAST einfach besser für die Suchgraphstruktur von Xiangqi geeignet ist als PPAP.

Im direkten Vergleich ist es nach den vorherigen Ergebnissen keine Überraschung, dass PPAFP mehr als die Hälfte der Spiele gegen PPAP gewinnt (Abb. 4.11). Es bestätigt sich, dass PPAFP die qualitativ beste Policy Tabelle erstellen kann. Nur kann es in fairen Versuchen mit den gleichen Ressourcen für jeden Algorithmus nicht ausreichend Playouts durchführen, um die Policy genügend zu adaptieren.

Insgesamt liegt die Vermutung nahe, dass mit steigender Playoutzahl PPAFP seinen Vorsprung ausbauen kann. Das wurde in dieser Arbeit allerdings nicht getestet, da PPAFP selbst für 20000 Playouts teilweise bis zu über 35s pro Zug benötigt und eine aussagekräftige Versuchsreihe sehr lange dauern würde, vor allem wenn Spiele im Durchschnitt über einhundert Züge lang sind.

[2]Es ist aber zu beachten, dass das Zuglimit für diese Versuchsreihe mit 170 Zügen bereits bei fast der doppelten Länge eines durchschnittlichen Xiangqi-Spiels liegt.

5.10. PPA-Algorithmen vs. ABS

Eine große Schwäche von UCT-basierten Algorithmen, welche wahrscheinlich der Hauptgrund für deren schlechtes Abschneiden (Abb. 4.12) gegen ABS in diesem Versuch ist, ist die Unfähigkeit, sogenannte „Trap States“ zuverlässig zu erkennen[12]. Ein Trap State bezeichnet einen Zustand, in dem eine Sequenz von Zügen existiert, mit der die GegnerIn garantiert gewinnen kann. In Schach sind solche Zustände nicht selten, was die Vermutung nahe legt, dass sie auch im Chinesischen Schach häufiger vorkommen. UCT hat Probleme, Züge zu identifizieren, die in solche Trap States führen, und weist ihnen oft ähnliche Bewertungen zu wie dem besten Zug im aktuellen Zustand. ABS dagegen erkennt einen Trap State immer, wenn die Suchtiefe über der Tiefe des Trap States[3] liegt, sprich wenn der Trap State bis zum garantierten Ausgang erkundet werden kann. Ebenso erkennt die ABS unter dieser Bedingung immer den Pfad, der in einem Trap State für den UCT-basierten Algorithmus zu ihrem garantierten Gewinn führt.[12]

Es kommt hinzu, dass UCT oft Zeit mit der Simulation tieferer Playouts in Trap States verschwendet. Je tiefer die Falle, desto mehr Zeit wird prozentual auf die Erkundung von Pfaden im Suchgraph verwendet, die bei einem optimalen Spiel der GegnerIn nicht eintreten werden.[12]

Diese Schwäche gefährliche Züge zu übersehen zeigt sich bereits vor dem Versuch gegen ABS in den teils extrem kurzen schnellsten Spielen (z.B. Tab. D.6), wo die UCT-basierten Algorithmen teilweise Züge ausführen, die aktiv dem gegnerischen Algorithmus helfen. Die UCT-basierten Algorithmen scheinen gerade anfangs im Spiel mit relativ wenig erkundeten Knoten eher zufällig Entscheidungen zu treffen, vor allem wenn wenige Playouts ausgeführt werden konnten. Das erklärt, warum PPAFP bei einer Bedenkzeit von 0,5s jedes einzelne Spiel gegen ABS verliert: Der Algorithmus kann hier nur knapp über 600 Playouts pro Zugsuche durchführen (Tab. G.1) und übersieht Gefahren für den eigenen König.

Doch auch viele Playouts bedeuten nicht zwingend, dass der Algorithmus besser abschneidet: Für drei von den verglichenen Algorithmen zeigt sich in Abb. 4.12, dass der größte Anteil von Unentschieden bei 2s pro Zugsuche vorliegt, während bei 7s ABS wieder ein paar mehr Spiele gewinnt. Das hängt eventuell mit der Suchtiefe von ABS zusammen. Wahrscheinlich gibt es eine Zeit, in welcher ABS im Schnitt gerade so eine tiefere Suchtiefe nicht erreicht, in welcher die anderen Algorithmen vergleichsweise stärker werden. Denn diese sollten linear von mehr Zeit profitieren, da sie immer mehr Playouts durchführen und ihre Statistik verbessern können. ABS ist dagegen in dieser Arbeit so implementiert, dass Iterative Deepening nur Züge von abgeschlossenen Suchen zurückgeben kann (also keine Zwischenergebnisse von tieferen, angefangenen Suchen). ABS wird also stufenweise besser. Eine Implementierung mit dem Zurückgeben von Zwischenergebnissen ist möglich, aber in dieser Arbeit lag der Fokus auf der Optimierung der PPA-Algorithmen, da sich schon in frühen Tests eine Überlegenheit von ABS abzeichnete.

Der Grund, warum ABS nie ein Spiel in unter sechs Zügen gewonnen hat, ist vermutlich, dass sie von einem optimalen Spiel der GegnerIn ausgeht. Sie leitet also nicht die Zugsequenz ein, die auf die Hilfe des gegnerischen Algorithmus baut, um dessen König zu schlagen.

[3] Die Tiefe eines Trap States ist die Nummer von Zügen, die gespielt werden muss, bis der garantierte Gewinnzustand für die GegnerIn erreicht ist.

Bis auf ein einziges Spiel kann keiner der UCT-basierten Algorithmen ABS in einem fairen Spiel besiegen. Daher ist nicht genau zu definieren, welcher Algorithmus am besten gegen ABS abschneidet: dies wurde im Rahmen dieser Arbeit an der Spiellänge und der Zahl der Unentschieden gemessen. Ein längeres Spiel bedeutet, dass die Algorithmen nicht sofort verlieren und daraufhin anzunehmen ist, dass sie bessere Entscheidungen insgesamt treffen. Eine höhere Anzahl an Unentschieden bedeutet, dass ABS weniger Spiele gewinnt, allerdings tritt hier wieder das Problem der ungeklärten Ursache auf: Es ist nicht klar, wieso die unentschiedenen Spiele das Zuglimit erreicht haben. Die durchschnittliche Länge eines Xiangqi-Spiels ist 95 Züge. Die gesetzte Grenze, 200, liegt weit darüber. Es ist nicht klar, ob die beiden Algorithmen bei einem Unentschieden tatsächlich gleich gut gespielt haben und deswegen das Spiel nicht früher beendet wurde oder ob sie sich in einer Endlosschleife von wiederholenden Zügen verfangen haben.

Als letztes wurde ein ungleicher Versuch durchgeführt, in welchem ABS nur die Hälfte der Zeit der anderen Algorithmen für die Zugsuche erhalten hat (Tab. G.2). Es ist zu sehen, dass die Suchtiefe sich nicht deutlich verringert und die anderen Algorithmen wieder sehr schlecht abschneiden. PPAFP verliert sogar mehr Spiele. Das ist vermutlich auf eine langsamere CPU in diesem Versuch und demzufolge noch weniger Playouts zurückzuführen. Die anderen Algorithmen erreichen vor allem in den Versuchen mit längeren Bedenkzeiten nun deutlich mehr Playouts, aber nur PPAP kann zwei Spiele für sich gewinnen. Dass es dabei gleich auch das schnellste Spiel für sich entscheidet ist interessant, aber da es sich um ein einmaliges Vorkommnis handelt, steht nicht fest, ob die Beobachtung strukturelle Gründe hat.

Interessant ist außerdem, dass PPAP als einziger UCT-basierter Algorithmus gegen ABS gewinnen konnte, nachdem es in vorherigen Versuchen von allen Algorithmen tendenziell am schlechtesten abgeschnitten hat (z.B. Abb. 4.7). Vielleicht hat es zufällig früh einen guten Zug gefunden und war mithilfe der Policy Tabelle in der Lage, die Playouts in die richtige Richtung zu lenken. Theoretisch haben die UCT-basierten Algorithmen gegenüber ABS den Vorteil, dass sie tiefer suchen können. Wird die Suche also erst einmal in die richtige Richtung konzentriert, können sie potentiell weiter vorausschauen als ABS. Möglicherweise hat PPAP es so geschafft, die drei Spiele zu gewinnen. Wieso aber kein anderer UCT-basierter Algorithmus dasselbe erreichen konnte, ist nicht erkenntlich.

Insgesamt zeigen die Ergebnisse klar, dass die UCT-basierten Algorithmen in der Praxis nicht mit ABS mithalten können, vor allem wenn bedacht wird, dass ABS für diese Arbeit kaum optimiert wurde. Die Schwäche von UCT-basierten Algorithmen in Schach-ähnlichen Spielen ist bekannt; besser schneiden sie in Spielen wie Go mit weniger bis gar keinen Trap States ab[12]. Diese Versuchsreihe zeigt, dass auch Policy Adaptation nicht ausreicht, um einen UCT-basierten Algorithmus zu konfigurieren, der in Xiangqi gegen ABS bestehen kann.

5.11. Limitationen

Trotz des Versuchs, den Vergleich der Algorithmen so fair wie möglich zu gestalten, unterliegt diese Arbeit einigen Limitationen.

Beispielsweise wurden nie alle möglichen Parameterkombinationen untereinander verglichen. Es ist also möglich, dass Vergleiche mit anderen Konfigurationen auch andere Ergebnisse geliefert hätten. Da es außerdem unendlich mögliche Werte für die Parameter gibt, können immer noch genauere Anpassungen durchgeführt werden. Jeder weitere Versuch stellt aber einen bedeutenden Zeitaufwand dar, weswegen sich in dieser Arbeit auf die beschriebene Gegenüberstellung von Parametereinstellungen beschränkt wurde.

Genauso liefern mehr durchgespielte Spiele eine bessere Statistik und mehr Spiele enden mit einem klaren Gewinner, wenn die maximale Zugzahl gehoben wird. Aber beide Parameter steigern den Zeitaufwand bedeutend, während eine höhere maximale Zugzahl auch noch den benötigten Speicherplatz zum Ausführen des Versuchs erhöht. Aufgrund der Vielzahl an zu untersuchenden Parametern wurden also beide Werte möglichst gering gehalten.

Es fehlt außerdem in der aktuellen Implementation eine Mechanik, die Zugwiederholungen nicht nur in den Playouts, sondern auch im eigentlichen Spiel erkennt und beendet. In dieser Implementierung ist nicht zu erkennen, ob ein Unentschieden nach Erreichen des Zuglimits ein aktives Spiel oder nur einen Zyklus unterbrochen hat, was die Bewertung der Algorithmen vor allem im Vergleich zu ABS erschwert.

Es ist außerdem zu beachten, dass die Algorithmen in dieser Arbeit nicht vollständig optimiert sind. Es gibt viele Techniken, die ABS weiter verbessern – wie etwa durch neuronale Netze erlernte Bewertungsfunktionen[15], Ruhesuche, oder auch eine komplexere Zugsortierung[8] – die hier nicht umgesetzt wurden. Und auch für PPA gibt es weitere Ansätze, wie ein Warmstarten der Policy Tabelle[5]. Genauso ist es sicherlich möglich, die Implementierung effizienter zu gestalten, sodass etwa Playouts, `adapt()` und Zuggeneration schneller laufen, was allen Algorithmen mehr Zeit für die Suche gibt und sie weiter verbessern sollte.

Alle für diese Arbeit umgesetzten Algorithmen sind von einem Menschen ohne viel Vorwissen zu Xiangqi auch bei längeren Bedenkzeiten für den Algorithmus schlagbar. Insgesamt handelt es sich dementsprechend nicht um eine wettbewerbsfähige Xiangqi-KI, die mit Großmeisterniveau-KIs wie z.B. ELP[6] mithalten könnte.

6. Fazit

In dieser Arbeit wurde ein empirischer Vergleich von Alpha-Beta Suche und Playout Policy Adaptation Algorithmen anhand des Spiels Xiangqi vorgenommen und eine Antwort auf die Frage gefunden, ob PPA-Algorithmen die Spielstärke von ABS erreichen oder sogar übertreffen können. Zudem wurden die PPA-Algorithmen untereinander verglichen und ihre Parameter für Xiangqi angepasst.

Es hat sich herausgestellt, dass im Vergleich aller UCT-basierten Algorithmen MAST (mit angepassten Parametern) am besten für Xiangqi geeignet ist und als einziges gegen UCT mit zufälligen Playouts in über der Hälfte der Fälle gewinnen kann.

Bei dem Vergleich mit ABS stellte sich heraus, dass keiner der UCT-basierten Algorithmen gut abschneiden konnte. In mehreren hundert Spielen verlor ABS nur drei Spiele gegen PPAP, selbst als sie mit der Hälfte des Zeitbudgets agierte.

Es stellt sich die Frage, ob mit weiteren Optimierungstechniken, wie einem Warmstart der Policy Tabelle, detaillierteren Parameteranpassungen oder völlig neuen Ideen einer der PPA-Algorithmen so weit verbessert werden kann, dass er in der Lage ist, in Xiangqi mit ABS mitzuhalten.

Im Lichte dessen, dass ABS selbst noch viel weiter optimiert werden kann, und UCT-basierte Algorithmen eine bekannte Schwäche in Schach-ähnlichen Spielen zeigen, scheint dies aber unwahrscheinlich. Es ist also kein Wunder, dass bisher die meisten Xiangqi-KIs auf Großmeisterniveau auf der Alpha-Beta Suche basieren.

Literaturverzeichnis

[1] Benjamin Blankertz and Vera Röhr. Algorithmen und datenstrukturen. Kommentar zum Skript: TU Berlin, 2023, VL Algorithmen und Datenstrukturen.

[2] Denis Breuker, J. W. H. M. Uiterwijk, and H. J. Van Den Herik. Information in transposition tables. *Advances in Computer Chess*, 8:199–211, 1997. PDF.

[3] Cameron Browne. Bitboard methods for games. *International Computer Games Assiciation journal*, 32(2):67–84, Juni 2014. DOI.

[4] Tristan Cazenave. Playout policy adaptation with move features. *Theoretical Computer Science*, 644:43–52, September 2016. DOI.

[5] Tristan Cazenave and Eustache Diemert. Memorizing the playout policy. In *Computer Games*, volume 818 of *Communications in Computer and Information Science*, page 96–107. Springer, Februar 2018. DOI.

[6] Jr-Chang Chen and Shun-Chin Hsu. Elp wins chinese chess tournament. *International Computer Games Association journal*, 24(3):182–184, September 2001. DOI.

[7] Ariel Felner, Sarit Kraus, and Richard E. Korf. Kbfs: K-best-first search. *Annals of Mathematics and Artificial Intelligence*, 39(1):19–39, September 2003. DOI.

[8] Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975. DOI.

[9] Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, März 1985. DOI.

[10] Juntong Lin, Zhichao Shu, and Yu Chen. Mastering chinese chess ai(xiangqi) without search. *arXiv preprint arXiv:2410.04865*, Oktober 2024. DOI.

[11] Lutz Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10):23–29, Oktober 2000. DOI.

[12] Raghuram Ramanujan, Ashish Sabharwal, and Bart Selman. On adversarial search spaces and sampling-based planning. *Proceedings of the International Conference on Automated Planning and Scheduling*, 20(1):242–245, Mai 2010. DOI.

[13] Christopher D. Rosin. Nested rollout policy adaptation for monte carlo tree search. *Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, 2011:649–654, Juli 2011. DOI.

[14] Jonathan Schaeffer. The history heuristic and alpha-beta search enhancements in practice. *IEEE transactions on pattern analysis and machine intelligence*, 11(11):1203–1212,

November 1989. DOI.

[15] Wen-Jie Tseng, Jr-Chang Chen, I-Chen Wu, and Tinghan Wei. Comparison training for computer chinese chess. *IEEE Transactions on Games*, 12(2):169–176, Januar 2019. DOI.

[16] Shi-Jim Yen, Jr-Chang Chen, Tai-Ning Yang, and Shun-Chin Hsu. Computer chinese chess. *International Computer Games Association journal*, 27(1):3–18, März 2004. DOI.

[17] Berk Yilmaz, Junyu Hu, and Jinsong Liu. Deep reinforcement learning xiangqi player with monte carlo tree search. *arXiv preprint arXiv:2506.15880*, Juni 2025. DOI.

[18] Albert L. Zobrist. A new hashing method with application for game playing. *International Computer Games Association journal*, 13(2):69–73, Juni 1990. DOI.

[19] Maciej Świechowski, Konrad Godlewski, Bartosz Sawicki, and Jacek Mańdziuk. Monte carlo tree search: a review of recent modifications and applications. *Artificial Intelligence Review*, 56:2497–2562, Juli 2022. DOI.

Anhang

A. PPAF vs. PPAFP

Metrik	PPAF	PPAFP	PPAF	PPAFP	PPAF	PPAFP
p	0,5		1		2	
Gewinne	27	33	28	32	27	33
Rate [%]	45	**55**	46,67	53,33	45	**55**
Playouts	1552	1804	1576	1776	1653	1817
Schn. Sp.	5			4	5	
Elo-Diff.		35		23		35
LOS [%]	21,93	78,07	30,28	69,72	21,93	78,07
Unent.	0		0		0	
Züge/Sp.	15		14		16	

Tabelle A.1.: PPAF vs. PPAFP mit variierendem p nach 60 Spielen.
$\alpha = 1$, $c = 1{,}414$, Zeit pro Zug = 2s, max. Zugzahl pro Spiel = 120.

B. α-Variation

Metrik	UCT	PPAP	UCT	PPAP	UCT	PPAP	UCT	PPAP	UCT	PPAP	UCT	PPAP
α	0,04		0,16		0,32		0,64		1		2,56	
Gewinne	48	6	46	4	46	3	47	4	47	5	47	4
Rate [%]	80	**10**	76,67	6,57	76,67	5	78,33	6,67	78,33	8,33	78,33	6,67
Playouts	140587	7037	136798	7525	136000	9079	136865	8204	141818	8502	141374	9948
Schn. Sp.	8		7		10		9		7		5	
Elo-Diff.	301		301		313		313		301		313	
LOS [%]	100	0	100	0	100	0	100	0	100	0	100	0
Unent.	6		10		11		9		8		9	
Züge/Sp.	38		43		46		43		46		46	

Tabelle B.1.: UCT vs. PPAP mit variierendem α nach 60 Spielen.
$p = 1$, $c = 1{,}414$, Zeit pro Zug = 2s, max. Zugzahl pro Spiel = 120.

Metrik	UCT	PPAFP	UCT	PPAFP	UCT	PPAFP	UCT	PPAFP	UCT	PPAFP	UCT	PPAFP
α	0,04		0,16		0,32		0,64		1		2,56	
Gewinne	56	2	51	5	48	9	54	5	50	2	45	8
Rate [%]	93,33	5	85	8,33	80	**15**	90	8,33	83,33	3,33	75	13,33
Playouts	188912	1941	191568	3593	184490	3264	211290	2907	159560	3527	165500	3960
Schn. Sp.	5		3		4		4		4		7	
Elo-Diff.	483		352		269		398		382		250	
LOS [%]	100	0	100	0	100	0	100	0	100	0	100	0
Unent.	1		4		3		1		8		7	
Züge/Sp.	24		29		27		21		34		34	

Tabelle B.2.: UCT vs. PPAFP mit variierendem α nach 60 Spielen.
$p = 2$, $c = 1{,}414$, Zeit pro Zug = 2s, max. Zugzahl pro Spiel = 120.

C. k-Variation

Metrik	UCT	MAST	UCT	MAST	UCT	MAST	UCT	MAST	UCT	MAST	UCT	MAST	UCT	MAST
k	1		4		16		64		128		256		512	
Gewinne	15	9	19	11	9	29	19	18	18	14	23	11	22	13
Rate [%]	25	15	31,67	18,33	15	**48,33**	31,67	30	30	23,33	38,33	18,33	36,67	21,67
Playouts	97264	34628	107517	48381	98796	71056	93685	63334	90309	59511	99486	60336	87228	55946
Schn. Sp.	9		3		10		14		3		4		9	
Elo-Diff.	35		47			120	6		23		70		53	
LOS [%]	89	11,03	92,79	7,21	0,06	99,94	56,53	43,47	76,03	23,98	98,02	1,98	93,59	6,41
Unent.	36		30		22		23		28		26		25	
Züge/Sp.	95		86		71		80		83		77		79	

Tabelle C.1.: UCT vs. MAST mit variierendem k nach 60 Spielen.
c = 1,414, Zeit pro Zug = 2s, max. Zugzahl pro Spiel = 120.

Metrik	UCT	MAST	UCT	MAST	UCT	MAST
k	8		16		32	
Rate [%]	16	44	22	48	23	33
Playouts	71189	43019	61986	39682	60855	37924
Schn. Sp.	10			14	3	
Elo-Diff.		100		92		35
LOS [%]	0,02	99,99	0,09	99,91	9,07	90,93
Unent.	40		30		44	
Züge/Sp.	103		87		104	

Tabelle C.2.: UCT vs. MAST mit variierendem k nach 100 Spielen (zusätzliche Messungen).
c = 1,414, Zeit pro Zug = 2s, max. Zugzahl pro Spiel = 150.

D. c-Variation

Dieser Anhang liefert einen ausführlichen Vergleich der PPA-Algorithmen mit UCT mit einer variierenden Explorationskonstante c. Es soll also in der Auswahlphase der Algorithmen eine gute Balance zwischen der Exploration neuer Züge und dem Ausnutzen bekannter, guter Züge gefunden werden. Die Werte wurden ausgehend von dem Defaultwert $\sqrt{2}$ gewählt.

In Tabellen farblich markierte Werte für c kennzeichnen den für zukünftige Versuche ausgewählten Wert für den Algorithmus.

CPU: Intel(R) Xeon(R) Gold 6326 CPU @ 2.90GHz

Anders als in den Versuchen zu p, α und k beeinflusst der c-Parameter auch UCT mit zufälligen Playouts (Alg. 2, Zeile 14). Es wurde je der gleiche Wert für c für beide zu vergleichenden Algorithmen gesetzt.

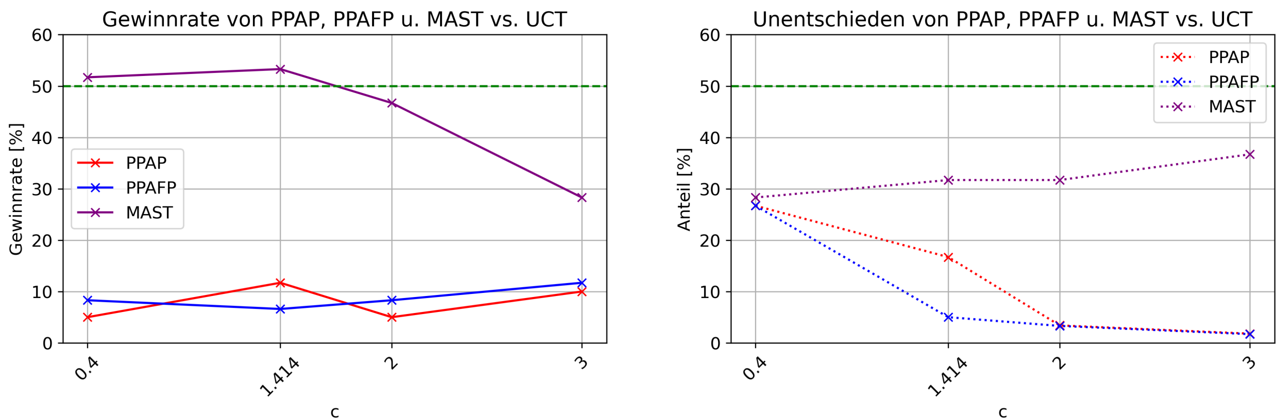

Abbildung D.1.: Vergleich der Gewinnraten und Unentschieden von PPAP, PPAFP u. MAST gegen UCT mit variierendem c nach 60 Spielen.
p (PPAP) = 1, p (PPAFP) = 2, α (PPAP) = 0,04, α (PPAFP) = 0,32, k = 16, Zeit pro Zug = 2s, max. Zugzahl pro Spiel = 120.

Wie in Abbildung D.1 zu sehen, liegen die Gewinnraten für PPAP und PPAFP für jeden Wert von c deutlich unter 50%. Einen Peak gibt es für PPAP bei c = 1,414 und für PPAFP bei c = 3. Bei beiden Algorithmen ist außerdem eine hohe Zahl an Unentschieden vor allem für kleinere c-Werte zu beobachten.

In Tabellen D.1 und D.2 wird ersichtlich, dass im Vergleich von PPAP und PPAFP vs. UCT die Elo-Differenz für jeden Wert von c hoch ist und die LOS immer bei 100% zugunsten von UCT liegt. Das schnellste Spiel wurde ebenfalls immer von UCT gewonnen, jeweils in sehr wenigen Zügen. Die Anzahl der durchschnittlichen Züge pro Spiel insgesamt sinkt stark mit

wachsendem c. Trotz der sehr eindeutigen LOS wurden die Versuche für Werte mit einer hohen Zahl an Unentschieden wiederholt, sowie eine neue Messung für $c = 4$ hinzugefügt, um verfolgen zu können, ob der Aufwärtstrend der Gewinnrate von 2 zu 3 sich weiterhin fortsetzt.

Metrik	UCT	PPAP	UCT	PPAP	UCT	PPAP	UCT	PPAP
c	0,4		1,414		2		3	
Gewinne	41	3	43	7	55	3	53	6
Rate [%]	68,33	5	71,67	11,67	91,67	5	88,33	10
Playouts	125184	7622	124166	6679	146568	5617	131549	5727
Schn. Sp.	3		6		5		5	
Elo-Diff.	260		241		458		366	
LOS [%]	100	0	100	0	100	0	100	0
Unent.	16		10		2		1	
Züge/Sp.	79		42		29		26	

Tabelle D.1.: UCT vs. PPAP mit variierendem c nach 60 Spielen.
$p = 1$, $\alpha = 0{,}04$, Zeit pro Zug = 2s, max. Zugzahl pro Spiel = 120.

Metrik	UCT	PPAFP	UCT	PPAFP	UCT	PPAFP	UCT	PPAFP
c	0,4		1,414		2		3	
Gewinne	39	5	53	4	53	5	52	7
Rate [%]	65	8,33	88,33	6,67	88,33	8,33	86,67	11,67
Playouts	155447	6467	185196	2513	180281	2553	148470	2165
Schn. Sp.	7		6		7		5	
Elo-Diff.	223		389		382		338	
LOS [%]	100	0	100	0	100	0	100	0
Unent.	16		3		2		1	
Züge/Sp.	57		26		25		23	

Tabelle D.2.: UCT vs. PPAFP mit variierendem c nach 60 Spielen.
$p = 2$, $\alpha = 0{,}32$, Zeit pro Zug = 2s, max. Zugzahl pro Spiel = 120.

Wie in vorherigen Versuchen gelingt es MAST für einen Teil der Werte, eine Gewinnrate von über 50% zu erreichen. Dies ist für $c = 0{,}4$ und 1,414 der Fall. Dort liegt die Elo-Differenz für MAST bei über 100 (Tab. D.3). Für alle Werte außer 3 erreicht MAST außerdem eine LOS von fast 100%. Trotzdem gewinnt UCT für jeden Wert von c das schnellste Spiel. Die durchschnittliche Spiellänge ändert sich für MAST kaum mit variierendem c.

Auch für MAST sind hohe Anteile von Unentschieden zu beobachten. Es werden die Messungen, die besonders erfolgversprechend scheinen, also für $c = 0{,}4$ und 1,414, wiederholt, sowie ein neuer Wert für c zwischen diesen eingeführt.

In der zweiten Versuchsreihe für c gelten die gleichen Werte wie zuvor, mit Ausnahme des maximalen Zugslimits. Dieses wurde auf 170 Züge erhöht, unter der Annahme, dass damit der prozentuale Anteil der Unentschieden sinkt.

Metrik	UCT	MAST	UCT	MAST	UCT	MAST	UCT	MAST
c	0,4		1,414		2		3	
Gewinne	12	31	9	32	13	28	21	17
Rate [%]	20	51,67	15	53,33	21,67	46,67	35	28,33
Playouts	117894	87107	103587	78389	97762	72642	97427	62864
Schn. Sp.	3		12		10		3	
Elo-Diff.		114		140		89	23	
LOS [%]	0,19	99,81	0,02	99,98	0,96	99,04	74,18	25,82
Unent.	17		19		19		22	
Züge/Sp.	70		75		71		71	

Tabelle D.3.: UCT vs. MAST mit variierendem c nach 60 Spielen.
$k = 16$, Zeit pro Zug = 2s, max. Zugzahl pro Spiel = 120.

CPU: Intel(R) Xeon(R) Gold 6326 CPU @ 2.90GHz

CPU (MAST): Intel(R) Xeon(R) Gold 6240L CPU @ 2.60GHz

Für PPAP zeigt sich, dass der Wert 4 für c eine vergleichsweise gute Gewinnrate liefert (Haupttext, Abb. 4.6). Allerdings ist auch sichtbar, dass nun mit einem höheren Zuglimit pro Spiel das Ergebnis für $c = 0{,}4$ insgesamt sowohl die beste Elo-Differenz, als auch die höchste Gewinnrate liefert (Tab. D.4). Die relative Anzahl an Unentschieden ist wie erhofft gesunken (Tab. D.4).

Auffällig ist, dass PPAP es für zwei der Werte schafft, das schnellste Spiel für sich zu entscheiden, wo es in vorherigen Versuchen diese immer gegen UCT verloren hat. Die Länge eines durchschnittlichen Spiels nimmt wieder für ein steigendes c stark ab. Für 0,4 liegt sie nah an dem Durchschnitt für Xiangqi, für einen Wert von 4 für c ist ein Spiel mit im Schnitt 24 Zügen sehr kurz.

Sowohl die beste Gewinnrate, als auch Elo-Differenz liegen für PPAP bei $c = 0{,}4$ vor. Dieser Wert wird für folgende Versuche gewählt.

Metrik	UCT	PPAP	UCT	PPAP	UCT	PPAP
c	0,4		1,414		4	
Rate [%]	67	**15**	81	7	88	11
Playouts	134492	11426	128823	11762	104956	5570
Schn. Sp.		15	5			5
Elo-Diff.	200		330		355	
LOS [%]	100	0	100	0	100	0
Unent.	18		12		1	
Züge/Sp.	84		51		24	

Tabelle D.4.: UCT vs. PPAP mit variierendem c nach 100 Spielen (zusätzliche Messungen).
$p = 1$, $\alpha = 0{,}04$, Zeit pro Zug = 2s, max. Zugzahl pro Spiel = 170.

PPAFP erreicht im zweiten Lauf ebenfalls für $c = 0{,}4$ die beste Gewinnrate (Haupttext, Abb. 4.6), auch wenn die Elo-Differenz größer ist als im ersten Versuch für den Wert 0,4 für c (Tab.

D.2). Der c-Wert 4 führt zu sehr kurzen Spielen, von denen UCT 93% für sich entscheiden kann (Tab. D.5). Es wird 0,4 für c verwendet, aufgrund der relativ hohen Gewinnrate im zweiten Testlauf und der längeren Spiele.

Sowohl PPAP als auch PPAFP sind laut der LOS von rund 100% mit hoher Wahrscheinlichkeit für jeden getesteten Wert von c UCT unterlegen.

Metrik	UCT	PPAFP	UCT	PPAFP
c	0,4		4	
Rate [%]	76	**12**	93	7
Playouts	160123	3168	132800	2099
Schn. Sp.	7		4	
Elo-Diff.	263		449	
LOS [%]	100	0	100	0
Unent.	12		0	
Züge/Sp.	58		21	

Tabelle D.5.: UCT vs. PPAFP mit variierendem c nach 100 Spielen (zusätzliche Messungen). $p = 2$, $\alpha = 0{,}32$, Zeit pro Zug = 2s, max. Zugzahl pro Spiel = 170.

Für alle im zweiten Versuch getesteten Werte für c erreicht MAST eine Gewinnrate über 50%, auch wenn der Anteil der Unentschieden immer noch bei rund einem Drittel liegt (Haupttext, Abb. 4.6). Ein Peak zeigt sich bei dem Wert $c = 0{,}9$, wo MAST eine Gewinnrate von 64% und eine Elo-Differenz von 215 erreicht (Tab. D.6). Dieser Wert wird im Folgenden für MAST für c weiter verwendet.

Die LOS liegt für die im zweiten Lauf getesteten Werte immer bei rund 100%, zugunsten von MAST. Die durchschnittliche Spiellänge liegt nahe am Durchschnitt für Xiangqi.

Obwohl MAST zufolge der LOS mit großer Wahrscheinlichkeit in mehreren Konfigurationen UCT überlegen ist, gewinnt UCT in diesem Versuchsabschnitt immer das schnellste Spiel, jedes Mal in nur drei Zügen.

Metrik	UCT	MAST	UCT	MAST	UCT	MAST
c	0,4		0,9		1,414	
Rate [%]	23	56	9	**64**	16	57
Playouts	81383	51399	81233	53818	78572	51541
Schn. Sp.	3		3		3	
Elo-Diff.		119		215		151
LOS [%]	0,01	99,99	0	100	0	100
Unent.	21		27		27	
Züge/Sp.	93		94		98	

Tabelle D.6.: UCT vs. MAST mit variierendem c nach 100 Spielen (zusätzliche Messungen). $k = 16$, Zeit pro Zug = 2s, max. Zugzahl pro Spiel = 170.

E. Zeit pro Zug Variation

Metrik	UCT	PPAP	UCT	PPAP	UCT	PPAP	UCT	PPAP
Zeit/Zug [s]	0,5		2		4		7	
Gewinne	37	7	41	2	49	3	51	3
Rate [%]	61,67	11,67	68,33	3,33	81,67	5	85	5
Playouts	34840	2226	132425	9944	303018	17435	551805	26134
Schn. Sp.	16		18		14		12	
Elo-Diff.	191		269		352		382	
LOS [%]	100	0	100	0	100	0	100	0
Unent.	16		17		8		6	
Züge/Sp.	89		83		66		67	

Tabelle E.1.: UCT vs. PPAP mit variierender Zeit pro Zug nach 60 Spielen. $p = 1$, $\alpha = 0{,}04$, $c = 0{,}4$, max. Zugzahl pro Spiel = 150.

Metrik	UCT	PPAFP	UCT	PPAFP	UCT	PPAFP	UCT	PPAFP
Zeit/Zug [s]	0,5		2		4		7	
Gewinne	54	7	44	5	36	7	32	9
Rate [%]	90	10	73,33	8,33	60	11,67	53,33	**15**
Playouts	60698	592	170821	3384	303134	10539	526749	22920
Schn. Sp.	4		7		11		9	
Elo-Diff.	382		269		183		140	
LOS [%]	100	0	100	0	100	0	99,98	0,02
Unent.	0		11		17		19	
Züge/Sp.	21		61		80		90	

Tabelle E.2.: UCT vs. PPAFP mit variierender Zeit pro Zug nach 60 Spielen. $p = 2$, $\alpha = 0{,}32$, $c = 0{,}4$, max. Zugzahl pro Spiel = 150.

Metrik	UCT	MAST	UCT	MAST	UCT	MAST	UCT	MAST
Zeit/Zug [s]	0,5		2		4		7	
Gewinne	7	32	6	37	8	36	6	46
Rate [%]	11,67	53,33	10	61,67	13,33	60	10	**76,67**
Playouts	29366	19979	113451	81631	224032	174314	420586	372427
Schn. Sp.		14		13	3			14
Elo-Diff.		154		199		176		280
LOS [%]	0	100	0	100	0	100	0	100
Unent.	21		17		16		8	
Züge/Sp.	101		90		82		72	

Tabelle E.3.: UCT vs. MAST mit variierender Zeit pro Zug nach 60 Spielen. $k = 16$, $c = 0{,}9$, max. Zugzahl pro Spiel = 150.

Metrik	UCT	PPAP	UCT	PPAP	UCT	PPAP
Zeit/Zug [s]	0,5		2		10	
Rate [%]	63	7	70	**15**	88	5
Playouts	31033	1990	140762	12676	785108	41433
Schn. Sp.	9		9		3	
Elo-Diff.	220		215		413	
LOS [%]	100	0	100	0	100	0
Unent.	30		15		7	
Züge/Sp.	83		71		54	

Tabelle E.4.: UCT vs. PPAP mit variierender Zeit pro Zug nach 100 Spielen (zusätzliche Messungen).
$p = 1$, $\alpha = 0{,}04$, $c = 0{,}4$, max. Zugzahl pro Spiel = 150.

Metrik	UCT	PPAFP	UCT	PPAFP	UCT	PPAFP
Zeit/Zug [s]	4		7		10	
Rate [%]	61	7	63	8	74	5
Playouts	210624	5984	294245	10670	405730	9131
Schn. Sp.	3		7		3	
Elo-Diff.	210		215		295	
LOS [%]	100	0	100	0	100	0
Unent.	32		29		21	
Züge/Sp.	77		77		66	

Tabelle E.5.: UCT vs. PPAFP mit variierender Zeit pro Zug nach 100 Spielen (zusätzliche Messungen).
$p = 2$, $\alpha = 0{,}32$, $c = 0{,}4$, max. Zugzahl pro Spiel = 150.

Metrik	UCT	MAST	UCT	MAST
Zeit/Zug [s]	0,5		10	
Rate [%]	21	46	8	74
Playouts	30148	20892	683026	647092
Schn. Sp.		11		9
Elo-Diff.		89		275
LOS [%]	0,11	99,89	0	100
Unent.	33		18	
Züge/Sp.	95		71	

Tabelle E.6.: UCT vs. MAST mit variierender Zeit pro Zug nach 100 Spielen (zusätzliche Messungen).
$k = 16$, $c = 0{,}9$, max. Zugzahl pro Spiel = 150.

F. Feste Playoutzahl

Metrik	UCT	PPAP	UCT	PPAFP	UCT	MAST
Rate [%]	29	46	6	54	9	61
Schn. Sp.		8		10		13
Elo-Diff.		60		182		200
LOS [%]	2,48	97,52	0	100	0	100
Unent.	25		40		30	
Züge/Sp.	91		111		103	

Tabelle F.1.: UCT vs. PPAP, PPAFP u. MAST mit 20000 Playouts nach 100 Spielen. p (PPAP) = 1, p (PPAFP) = 2, α (PPAP) = 0,04, α (PPAFP) = 0,32, k = 16, c (PPAP/PPAFP) = 0,4, c (MAST) = 0,9, max. Zugzahl pro Spiel = 170.

Metrik	MAST	PPAP	MAST	PPAFP
Rate [%]	67	16	26	36
Schn. Sp.		16	11	
Elo-Diff.	196			35
LOS [%]	100	0	10,2	89,8
Unent.	17		38	
Züge/Sp.	80		106	

Tabelle F.2.: MAST vs. PPAP u. PPAFP mit 20000 Playouts nach 100 Spielen. p (PPAP) = 1, p (PPAFP) = 2, α (PPAP) = 0,04, α (PPAFP) = 0,32, k = 16, c = 0,65, max. Zugzahl pro Spiel = 170.

Metrik	PPAP	PPAFP
Rate [%]	15	54
Schn. Sp.	17	
Elo-Diff.		143
LOS [%]	0	100
Unent.	31	
Züge/Sp.	93	

Tabelle F.3.: PPAP vs. PPAFP mit rund 20000 Playouts nach 100 Spielen. p = 1,5, α = 0,18, c = 0,4, max. Zugzahl pro Spiel = 170.

G. PPA-Algorithmen vs. ABS

Metrik	PPAP			PPAFP			MAST			UCT		
Zeit/Zug [s]	0,5	2	7	0,5	2	7	0,5	2	7	0,5	2	7
Rate [%]	0	**1**	0	0	0	0	0	0	0	0	0	0
Elo-Diff.	424	246	301	undef.	478	330	402	315	338	463	288	372
Unent. [%]	16	37	30	0	12	26	18	28	25	13	32	21
Playouts	14923	45734	112786	612	3648	27431	26999	167428	855265	21752	95964	601950
Züge/Sp.	64	108	97	25	63	91	74	93	94	60	96	82
ABS Tiefe	5–6	6–7	7–8	5–6	6–7	7–8	5–6	6–7	7–8	5–6	6–7	7–8

Tabelle G.1.: ABS vs. PPAP, PPAFP, MAST u. UCT mit variierender Zeit pro Zug nach 100 Spielen.
p (PPAP) = 1, p (PPAFP) = 2, α (PPAP) = 0,04, α (PPAFP) = 0,32, k = 16, c (PPAP/PPAFP) = 0,4, c (MAST) = 0,9, c (UCT) = 1,414, max. Zugzahl pro Spiel = 200.

Metrik	PPAP			PPAFP			MAST			UCT		
Zeit/Zug [s]	0,5	2	7	0,5	2	7	0,5	2	7	0,5	2	7
Rate [%]	0	0	**2**	0	0	0	0	0	0	0	0	0
Elo-Diff.	512	282	301	undef.	576	478	531	295	252	478	363	308
Unent. [%]	10	33	26	0	7	12	9	31	38	12	22	29
Playouts	9543	75090	236051	407	2241	17806	14548	205894	892312	43184	290818	2056672
Züge/Sp.	56	95	97	23	47	63	62	96	115	58	82	91
ABS Tiefe	5	6	7	5	6	7	5	6	7	5	6	7

Tabelle G.2.: Biased Vergleich von ABS zu PPAP, PPAFP, MAST u. UCT mit variierender Zeit pro Zug nach 100 Spielen. ABS bekommt hier jeweils die Hälfte der angegebenen Zeit.
p (PPAP) = 1, p (PPAFP) = 2, α (PPAP) = 0,04, α (PPAFP) = 0,32, k = 16, c (PPAP/PPAFP) = 0,4, c (MAST) = 0,9, c (UCT) = 1,414, max. Zugzahl pro Spiel = 200.