Otto-von-Guericke-Universität Magdeburg

Fakultät für Informatik

Lehrstuhl für Intelligente Systeme

Monte Carlo Tree Search in 4X-Spielen

Bachelorarbeit

Autor:

Eric Neumann - Emanuel Thon

Betreuende Professorin:

Prof. Dr.-Ing. habil. Sanaz Mostaghim

Betreuer:

Carlo Nübel, M.Sc.

Magdeburg, 18.6.2025

Inhaltsverzeichnis

1 Kurzfassung **3**

Akronyme & Fachbegriffe **4**

2 Einleitung **5**
2.1 Motivation . . . 5
2.2 Ziel . . . 5
2.3 Outline . . . 6

3 Theoretische Grundlagen **7**
3.1 Künstliche Intelligenz (KI)-Algorithmen in Strategiespielen . . . 7
3.2 Monte Carlo Tree Search . . . 8
3.2.1 Algorithmus . . . 8
3.2.2 Selection . . . 9
3.2.3 Expansion . . . 11
3.2.4 Simulation . . . 12
3.2.5 Backpropagation . . . 12
3.2.6 Beispiel . . . 13

4 Verwandte Arbeiten **15**
4.1 Monte Carlo Tree Search . . . 15
4.2 Action Space Pruning . . . 16
4.3 Verwandte Spiele . . . 17
4.4 MCTS mit Memory . . . 17

5 Quick Civ **19**
5.1 Spielbeschreibung . . . 19
5.2 Komplexität . . . 21
5.3 Performance . . . 23

6 Methodik **24**
6.1 Random und Greedy . . . 24
6.2 Vanilla-MCTS . . . 27
6.3 Hyper Parameter Optimization . . . 31
6.4 Action Space Pruning . . . 31
6.4.1 Only Military Overwrites Pruning . . . 32
6.4.2 Greedy Nodes Only Pruning . . . 34
6.4.3 Only Build Near Last Action Pruning . . . 35
6.4.4 In Late No Early Buildings Pruning . . . 35
6.5 Memory MCTS . . . 35

7 Experimente und Ergebnisse **37**
7.1 Hyper Parameter Optimization . . . 37
7.1.1 Parameterergebnisse . . . 37
7.1.2 Auswertung . . . 42
7.1.3 Spielweise . . . 45
7.2 VanillaMCTSBot versus Benchmarks . . . 47
7.3 Qualität des MCTS . . . 47
7.4 Action Space Pruning und Memory MCTS . . . 48
7.4.1 Greedy Nodes Only Pruning . . . 49
7.4.2 In Late No Early Building Pruning . . . 50
7.4.3 Only Build Near Last Action Pruning . . . 52
7.4.4 Only Military Overwrites Pruning . . . 54
7.4.5 Auswertung . . . 55
7.5 Memory MCTS . . . 56
7.6 Der beste Memory-MCTS . . . 58

8 Fazit **61**
8.1 Zusammenfassung . . . 61
8.2 Ausblick . . . 62

Anhang **63**
A. Bot-Initialisierung . . . 63
B. Gebäudeliste . . . 64

Literaturverzeichnis **65**

1 Kurzfassung

[EN] Entscheidungsfindung in großen Suchräumen ist ein zentrales Thema der KI. Eine bewährte Methode zur effizienten Erkundung solcher Räume ist der Monte Carlo Tree Search oder Monte Carlo Tree Search Algorithmus (MCTS). Allerdings stößt auch MCTS bei Domänen mit sehr hohem Verzweigungsfaktor an seine Grenzen. Im Rahmen dieser Bachelorarbeit wurde ein eigenes Strategiespiel als Testumgebung entwickelt, um die Eignung und Leistungsfähigkeit von MCTS in einer solchen komplexen Entscheidungsdomäne zu untersuchen. Experimente und eigenes Spielgefühl zeigten, dass sich MCTS gut für das verwendete Strategiespiel anbietet, wenn man bei der Entwicklung einen großen Fokus auf Performanceoptimierung legt, damit der Algorithmus ausreichend viele Spielzustände simulieren kann. Zur Verbesserung der Performance wurden ausgewählte Pruning-Verfahren sowie eine eigens konzipiertes Speicherkomponente (*Memory*) in den Algorithmus integriert. Die Ergebnisse zeigen, dass sowohl die Pruning-Strategien als auch das *Memory* zu einer Steigerung der Effizienz und Lösungsqualität des MCTS beitragen können.

Akronyme & Fachbegriffe

KI Künstliche Intelligenz

MCTS Monte Carlo Tree Search oder Monte Carlo Tree Search Algorithmus

RTS Real Time Strategy

4X Subgenre der Strategiespiele (eXplore, eXpand, eXploit, eXterminate)

UCB Upper Confidence Bound

EA Evolutionärer Algorithmus

HPO Hyper Parameter Optimization

2 Einleitung

[ET] „Denn eines hat Civ 6 dann eben doch mit *Civilization: Revolution* gemeinsam: die schwache KI. Denn die Computergegner bringen selbst auf den höchsten Schwierigkeitsgraden wenig zustande - vor allem im Krieg ziehen sie konfus hin und her oder schließen Frieden, obwohl sie eindeutig die Oberhand haben. Das heißt nicht, dass ich automatisch gewinne, mit ihren massiven Ressourcenboni sausen die hochstufigen KI-Widersachern rasant zum Kultur- oder Technologiesieg.“ [Gam].

2.1 Motivation

[ET] KI in Strategiespielen ist bekannt dafür, immer wieder starke Aussetzer zu haben, schlechte Entscheidungen zu treffen und dabei auch noch zu schummeln. Gerade KI in 4X-Strategiespielen, wie die der Spielereihe [Gam23], ist hauptsächlich bekannt dafür, der größte Kritikpunkt der dazugehörigen Spiele zu sein. Dies liegt größtenteils daran, dass Strategiespiele sehr komplex sind und die Algorithmen, die verwendet werden, häufig aus Budgetgründen relativ einfach gehalten sind. Meist werden dann Entscheidungsbäume verwendet. Diese sind zwar grundsätzlich schnell gebaut, müssen bei Änderungen am Spiel aber auch per Hand angepasst werden. In dieser Arbeit geht es um das selbst entwickelte Strategiespiel *Quick Civ* und die dazugehörige Gegner-KI. Dabei wird der MCTS-Algorithmus verwendet, um sowohl weniger fehleranfälliger zu sein, als auch um den Overhead, der durch das Anpassen an neue Spielmechaniken entsteht, zu minimieren.

Auch im größeren Kontext gibt es viele Szenarien, in denen jetzt schon KI-Systeme eingesetzt werden um komplexe Entscheidungen zu treffen. Durch die Anwendung bei kleineren Modellproblemen lassen sich sehr gut die Limitationen der Systeme erkennen. Hierfür eignen sich Strategiespiele sehr gut, da sie in ihrer Komplexität teilweise an die echte Welt heranreichen.

2.2 Ziel

[ET] Ziel dieser Arbeit ist das Entwickeln des Spiels *Quick Civ* und einer dazugehörigen Gegner-KI, welche ohne Hilfe von spielerischen Vorteilen in der Lage ist, einem menschlichen Spieler eine Herausforderung zu sein. MCTS ist ein Algorithmus, welcher zu jedem Zeitpunkt eine valide Aktion ausgeben kann und gegen die optimale Lösung konvergiert. Das heißt, der Algorithmus wird hauptsächlich durch Performanzverbesserungen stärker. Ein Teil davon ist das sogenannte Action Space Pruning, welches daraus besteht, Aktionen, die mit hoher Wahrscheinlichkeit schlecht sind, kategorisch auszuschließen. So wird die

verfügbare Rechenzeit für die vielversprechendsten Aktionen aufgehoben. Daraus ergeben sich folgende Forschungsfragen:

1. Wie gut funktioniert MCTS als KI-Gegner im Subgenre der Strategiespiele (eXplore, eXpand, eXploit, eXterminate) (4X)-Spielen und wie muss man die Parameter anpassen um eine optimale Performanz zu erreichen?
2. Wie stark lässt sich die Performanz des Algorithmus mit Action Space Pruning verbessern.
3. Kann die Performanz des Algorithmus durch ein selbst entwickeltes Gedächtnis (Memory) noch verbessert werden.

Hauptbeitrag: In dieser Arbeit geht es darum, ob man MCTS als KI-Algorithmus für Gegner in 4X-Spielen benutzen kann. Dies wird an einem relativ simplen selbst erstellten 4X-Spiel geprüft. Ein besonderes Augenmerk wird dabei auf das Problem eines großen Action-Spaces und Möglichkeiten diesen einzuschränken gelegt. Dafür wird auch ein MCTS spezifisches Memory implementiert, welches sich Zugbewertungen über mehrere Runden hinweg merken kann.

2.3 Outline

[ET] In Kapitel 3 geht es um die Theoretischen Grundlagen von KI in Videospielen sowie die Funktionsweise des, in dieser Arbeit verwendeten, MCTS-Algorithmus. Kapitel 4 beschäftigt sich mit anderen wissenschaftlichen Arbeiten zu den Themen MCTS, Action Space Pruning, KI in Videospielen und durch Speicher verstärkter MCTS. In Kapitel 5 wird das Spiel *Quick Civ* vorgestellt und die Probleme aufgezeigt, den Algorithmus darauf anzuwenden. Die Funktionsweise der *Quick Civ* Gegner-KI, des Action Space Prunings und des Memories werden in Kapitel 6 erklärt. In Kapitel 7 wird der Aufbau der Experimente und die daraus resultierenden Ergebnisse beschrieben und ausgewertet. Die Zusammenfassung und ein Ausblick auf aufbauende Forschungsfragen sind in Kapitel 8 zu finden.

3 Theoretische Grundlagen

[EN] Im ersten Abschnitt dieses Kapitels werden alternative Methoden, die anstelle von MCTS zur KI-Entwicklung in Spielen verwendet werden könnten, vorgestellt und die Gründe für die Entscheidung zugunsten von MCTS erläutert. Anschließend werden die vier einzelnen Phasen des MCTS detailliert erklärt: *Selection* (Auswahl eines vielversprechenden Knotenpunkts), *Expansion* (Hinzufügen neuer Knoten), *Simulation* (Durchspielen mehrerer Runden ausgehend von diesem Punkt) und *Backpropagation* (Rückführung der gewonnenen Informationen zur Optimierung zukünftiger Entscheidungen).

3.1 KI-Algorithmen in Strategiespielen

[EN] **Entscheidungsbäume**, oder auch *Decision Trees* genannt, sind eine Methode, bei der vordefinierte Bedingungen abgefragt werden und daraufhin entschieden wird, welche Aktion von der KI ausgeführt wird [DV13], [KS08], [RM05]. Diese Technik wird oft in Spielen mit klaren, regelbasierten Entscheidungsprozessen eingesetzt, bei denen es eine begrenzte Anzahl möglicher Zustände gibt. Ein bekanntes Beispiel ist die KI in *Pac-Man*, die für die Geister einfache Entscheidungsbäume nutzt. So bestimmt der Algorithmus anhand von Faktoren wie der Entfernung zum Spieler oder der aktuellen Spielphase, ob ein Geist angreifen oder fliehen soll [Mül]. Entscheidungsbäume sind für einen geringen Suchraum eine plausible Methode, jedoch geraten sie bei zunehmender Komplexität sehr schnell an ihre Grenzen. Zudem haben Entscheidungsbäume einen weiteren schwerwiegenden Nachteil während der Entwicklung eines Computerspiels. Beim Konstruieren eines Entscheidungsbaums ist man gezwungen, Domain-Wissen anzuwenden und anzupassen, wenn sich etwas im Spiel ändert. Da das Spiel *Quick Civ* noch weit von einer finalen Version entfernt ist, sind dynamische Algorithmen deutlich praktischer, da sie nicht mit jeder Spieländerung angepasst werden müssen.

Der **Minimax-Algorithmus** ist eine klassische Methode für rundenbasierte Spiele mit perfekter Information, wie Schach oder Tic-Tac-Toe [Mül]. Dabei wird ein Spielbaum generiert, in dem alle möglichen zukünftigen Spielzüge simuliert werden. Der Algorithmus geht davon aus, dass beide Spieler optimal agieren, und bewertet jeden möglichen Spielzustand mit einer Nutzenfunktion. Ziel ist es, den bestmöglichen Zug zu wählen, indem entweder der eigene Gewinn maximiert oder der potenzielle Gewinn des Gegners minimiert wird. Der Minimax-Algorithmus kommt in vielen Brettspielen zum Einsatz, darunter Schach (z. B. in der Stockfish-Engine [sto22]) und Dame ([CIK+23], [MDCB23], [EC18]). Allerdings kann er mit zunehmender Spielkomplexität schnell an seine Grenzen stoßen, weshalb er für das Spiel *Quick Civ* mit hoher Komplexität, siehe Abschnitt 5.2, nicht anwendbar ist.

Neuronale Netzwerke und Reinforcement Learning haben in den letzten Jahren große Fortschritte gemacht und werden in vielen modernen KI-gesteuerten Spielen eingesetzt. Anstatt sich auf explizite Entscheidungsbäume oder festgelegte Regeln zu stützen, lernen diese Methoden durch wiederholtes Spielen und Belohnungssysteme, welche Aktionen am vorteilhaftesten sind und trainieren durch das Spielen ein Neuronales Netz. Ein bekanntes Beispiel ist AlphaGo [Che16], das durch Deep Reinforcement Learning die besten Strategien für das komplexe Brettspiel Go erlernte und schließlich menschliche Großmeister besiegen konnte [alp16]. Auch in modernen Real Time Strategy (RTS) - Spielen wie *StarCraft II* wird diese Technik genutzt, um KI-Agenten zu trainieren, die sich dynamisch an verschiedene Spielsituationen anpassen können [Bli10]. Die Verwendung von Neuronalen Netzen könnte auch die Evaluierungsfunktion, siehe Gleichung 6.2, ersetzen bzw. verbessern. Dies ist aber frühestens für einen Nachfolgeteil des Spiels *Quick Civ* angedacht.

3.2 Monte Carlo Tree Search

[EN] **Monte Carlo Tree Search** ist der Algorithmus, der in dieser Arbeit verwendet wird, weil er sich in einigen Strategiespielen bewährt hat, wie in einigen Artikeln beschrieben wird [SCS09], [Roe12], [Sam18], [ZHR19], [JWF18], [RFT14], [AHH10], [CF10], [Win17]. Er ermöglicht hochwertige Aktionsauswahl, ohne einen vollständigen Baum generieren zu müssen – ein klarer Vorteil gegenüber dem Minimax-Algorithmus, der aus genau diesem Grund oft gemieden wird. Zudem ist MCTS ein *Any-Time-Algorithmus*, was bedeutet, dass er nach einer bestimmten Zeit terminiert werden kann und er daraufhin die bis dahin beste gefundene Aktion zurückgibt. Auf diese Weise sind sogar verschiedene Schwierigkeitsgrade für das Spiel trivial implementierbar, indem den verschiedenen Schwierigkeiten unterschiedlich lange *Denkphasen* gegeben werden. Auch wenn MCTS nicht immer die beste Entscheidung garantiert, ist das für ein Spiel der Komplexitätsklasse von *Quick Civ* unproblematisch. Der Aktionsraum ist so groß, dass es nicht darauf ankommt, die optimale Aktion zu finden – eine *gute* reicht völlig aus.

3.2.1 Algorithmus

[EN] MCTS ist ein simulationsbasierter Algorithmus zur Entscheidungsfindung in komplexen Umgebungen [Cou06], [BPW+12], [ŚGSM23]. Der Algorithmus kombiniert den klassischen Monte-Carlo-Ansatz, bestehend aus Zufallsstichproben und statistischer Auswertung, mit einer baumbasierten Suchtechnik. Grundlegend basiert der Algorithmus darauf, vom aktuellen Spielzustand einen Suchbaum aufzubauen, indem man Folgezustände simuliert und daraufhin bewertet. Er besteht aus vier Hauptschritten, die iterativ durchlaufen werden. Diese Unterpunkte werden im folgenden noch ausführlich erklärt. In der Abbildung 3.1 wird der Ablauf der Schritte grafisch dargestellt.

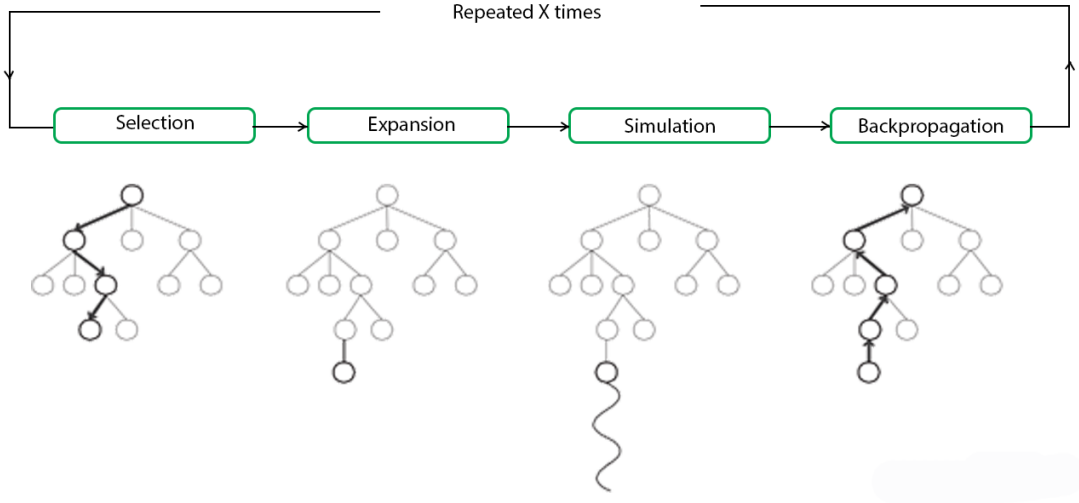


Abbildung 3.1: Monte Carlo Tree Search. Quelle: [Roy19]

3.2.2 Selection

[EN] In dieser Phase wird ein Knoten ausgehend vom Wurzelknoten, also dem einzigen Knoten ohne Elternknoten, des aktuellen Suchbaums anhand einer festgelegten Strategie ausgewählt, der weiter untersucht werden soll. Dabei spielt die Balance zwischen *Exploration* und *Exploitation* eine zentrale Rolle.

Exploration bedeutet, neue oder bislang wenig betrachtete Knoten zu erforschen, um potenziell bessere Entscheidungen zu entdecken. *Exploitation* hingegen bezeichnet die bevorzugte Selektion von Knoten, die bereits gute Ergebnisse geliefert haben, um auf bekannten, vielversprechenden Strategien aufzubauen und diese tiefer zu untersuchen.

MCTS muss beide Aspekte geschickt ausbalancieren: Zu viel Exploration kann zu ineffizienter Suche führen, während zu viel Exploitation dazu führen kann, dass bessere Alternativen übersehen werden und der Algorithmus in einem lokalen Maxima steckenbleibt. Deswegen haben sich mehrere Selektionsstrategien gebildet, welche nun genauer erläutert werden.

Die **Random-Selection**, siehe Code 3.1, ist sehr simpel. Ausgehend vom Startknoten wird zufällig ein Kindknoten ausgewählt, bis ein Blattknoten, also ein Knoten ohne eigene Kinder, erreicht wird oder ein Knoten, der noch nicht vollständig exploriert wurde. Vollständige Exploration bedeutet in diesem Kontext die Betrachtung aller möglichen Kindknoten, welche sich durch den aktuellen Spielzustand und alle legalen Aktionen ergeben.

Code 3.1: Random-Selection

```
def SELECT_RANDOM(node):
    while node is not Blattknoten:
        if node is not vollstaendig_expandiert:
            return EXPAND(node)
        else:
            node = random.choice(node.Children)
    return node
```

Die **Greedy-Selection**, siehe Code 3.2, wählt stets den Kindknoten mit dem höchsten Gewinnverhältnis (Erfolgsquote). Mit dieser Strategie wird stets Exploitation genutzt und nie exploriert.

Code 3.2: Greedy-Selection

```
def SELECT_GREEDY(node):
    while node is not Blattknoten:
        if node is not vollstaendig_expandiert:
            return EXPAND(node)
        else:
            node = BEST_CHILD_GREEDY(node)
    return node

def BEST_CHILD_GREEDY(node):
    return max(node.Children, key=lambda child: child.Value / child.Visits)
```

Wie in Code 3.3 zu sehen, nutzt die **Epsilon-Greedy-Selection** eine Mischung aus der Random- und der Greedy-Strategie. Mit Wahrscheinlichkeit ϵ wird ein zufälliger Kindknoten gewählt, ansonsten der beste. Mit dieser Strategie werden Exploitation und Exploration auf einfache Weise ausbalanciert. Dabei besteht jedoch das Problem, dass wenn Exploitation praktiziert wird, stets der allerbeste Knoten ausgewählt wird, auch wenn er schon vielfach häufiger ausgewählt wurde, als ähnlich gute Knoten.

Code 3.3: Epsilon-Greedy-Selection

```
def SELECT_EPSILON_GREEDY(node, epsilon):
    while node is not Blattknoten:
        if node hat nicht vollstaendig expandierte Kinder:
            return EXPAND(node)
        else:
            if random.random() < epsilon:
                node = RANDOM_CHOICE(node.Children)
            else:
                node = BEST_CHILD_GREEDY(node)
    return node
```

Die **Upper Confidence Bound (UCB)-Selection**, siehe Code 3.4, verwendet eine Balance zwischen Exploration und Exploitation basierend auf der UCB1-Formel [KS06]. Diese Selektionsstrategie verspricht eine gleichmäßigere Untersuchung aller Knoten, mit Fokus auf die erfolgversprechendsten.

$$UCB = \frac{w_i}{n_i} + C \cdot \sqrt{\frac{\ln N}{n_i}} \tag{3.1}$$

wobei:

- w_i die Anzahl der Gewinne des Kindes i ist,

- n_i die Anzahl der Besuche von Kind i ist
- N die Gesamtzahl der Besuche des Elternknotens ist
- C eine explorative Konstante ist

Diese Formel sorgt für eine ausgewogene Balance zwischen Exploration und Exploitation. Der erste Term $\frac{w_i}{n_i}$ stellt sicher, dass Knoten mit hohen Erfolgschancen bevorzugt werden (Exploitation). Der zweite Term $C \cdot \sqrt{\frac{\ln N}{n_i}}$ belohnt Knoten, die bisher weniger oft besucht wurden, um sicherzustellen, dass auch weniger bekannte Alternativen weiter untersucht werden (Exploration).

Code 3.4: UCB-Selection

```
def SELECT_UCB(node, C):
    while node is not Blattknoten:
        if node is not vollstaendig_expandiert:
            return EXPAND(node)
        else:
            node = BEST_CHILD_UCB(node, C)
    return node

def BEST_CHILD_UCB(node, C):
    UCB_VALUES = []
    for child in node.Children:
        UCB = (child.Value / child.Visits) + C * sqrt(log(node.Visits) / child.Visits)
        UCB_VALUES.Add((UCB, child))
    return node_of(max(UCB))
```

3.2.3 Expansion

[EN] In dieser Phase geht es darum, den Suchbaum zu erweitern. Nachdem in der *Selection*-Phase ein Knoten ausgewählt wurde, welcher weiter exploriert werden soll, wird dieser nun um einen Kindknoten erweitert. Dazu wird der Spielzustand, der in dem selektierten Knoten gespeichert ist, verwendet. Es wird eine zufällige legale Spielaktion für jeden Spieler ausgehend vom diesem Spielzustand ausgewählt und ausgeführt. Der resultierende Spielzustand wird nun in dem neu erstellten Kindknoten gespeichert.

```
def EXPAND(node):
  if node is terminal:
    return node #Keine Expansion moeglich

  possible Actions = getPossibleActions(node.gameState)

  for action in possibleActions:
    if not node.has_child_for_action(action):
      newGameState = getNewGameState(node.gameState, action)

      for player in players:
        possibleActions enemy = player.getPossibleActions(newGameState)
        action_enemy = RANDOM_CHOICE(possibleActions_enemy)
        newGameState = getNewGameState(newGameState, action_enemy)

      newChild = ERSTELLE_KNOTEN(newGameState, parent=node)
      node.Children.Add(newChild)
      return newChild
```

3.2.4 Simulation

[EN] Nachdem der neue Knoten erstellt wurde, braucht er noch eine Bewertung. Dazu werden im MCTS-Algorithmus Simulationen, auch genannt *Rollouts* verwendet. Dabei wird ein Spiel oftmals bis zum Spielende simuliert und in dem Knoten gespeichert, ob er die Simulation gewonnen hat. Diese Standardvariante wurde mit der Zeit jedoch aufgrund einiger Probleme in spezifischen Fällen weiterentwickelt. Zum Einen müssen die Simulationen nicht zufällig gespielt werden, sondern können auch nach dem Greedy-Algorithmus durchgespielt werden, um die Simulationen realistischer und damit wertvoller zu gestalten. Andererseits ist eine einzige Simulation oftmals nicht ausreichend, um eine aussagekräftige Bewertung abzugeben. Deswegen werden deutlich mehr als nur eine Simulation durchgeführt und der Quotient aus Siegen und gespielten Simulationen gebildet. Falls die Domaine keine Simulationen bis zum Spielende aufgrund von Performanceproblemen zulassen, müssen diese vorher terminiert werden. Zudem werden die Simulationen zunehmend ungenauer, je tiefer sie führen. Aus diesen Gründen verwenden einige Entwickler eine Evaluierungsfunktion, welche nach einer bestimmten Simulationstiefe auf den resultierenden Spielzustand angewandt wird, um den Knoten anhand von Heuristiken zu bewerten. Beispielsweise werden zehn Züge von der KI und den Gegnern ausgeführt und am Ende der Züge wird eine Evaluierungsfunktion benutzt, um den Spielzustand zu bewerten. Logischerweise werden dann nicht die Siege in dem Knoten abgespeichert, sondern der Wert, der bei der Evaluation zurückgegeben wird, gemittelt über die Anzahl der Simulationen.

3.2.5 Backpropagation

[EN] In dieser Phase wird die Rückführung der Bewertung zum Wurzelknoten vollzogen. Nachdem nun der neu erstellte Knoten durch die Simulationsphase bewertet wurde, wird sein Wert an seine Elternknoten weitergegeben, bis der Wurzelknoten erreicht wurde. Dadurch wird erreicht, dass Knoten mit jeder weiteren Iteration des Gesamtalgorithmus weiter erforscht werden und die Bewertung immer genauer wird.

3.2.6 Beispiel

[EN] In den folgenden Grafiken (Abbildung 3.2 und Abbildung 3.3) wird der Algorithmus anhand eines Beispiels anschaulich erklärt.

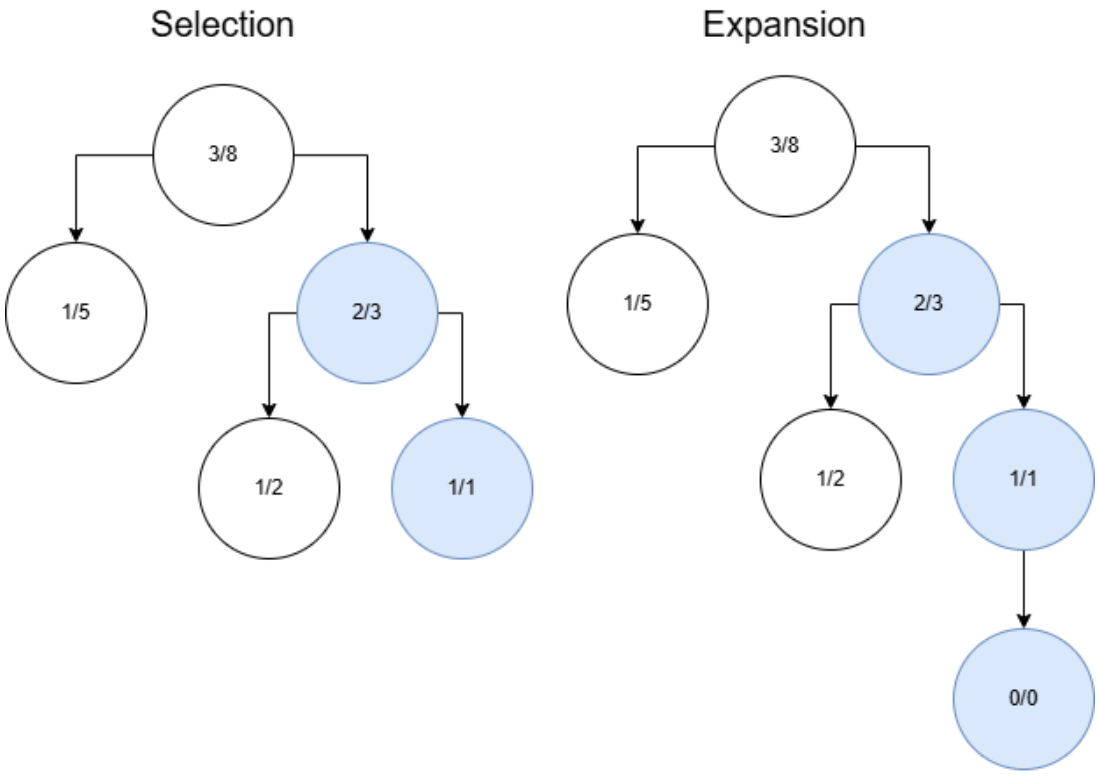


Abbildung 3.2: Selection und Expansion

In diesem Beispiel wird als Selektionsstrategie die Greedy-Selektion verwendet. Der Knoten mit dem größten Quotienten aus Siegen und Spielen wird so lange ausgewählt, bis ein Blattknoten oder ein nicht vollständig explorierter Knoten erreicht wird. In der Expansion wird eine zufällige Aktion vom ausgewählten Knoten ausgewählt und ausgeführt. Der resultierende Knoten wird in den Suchbaum eingehängt.

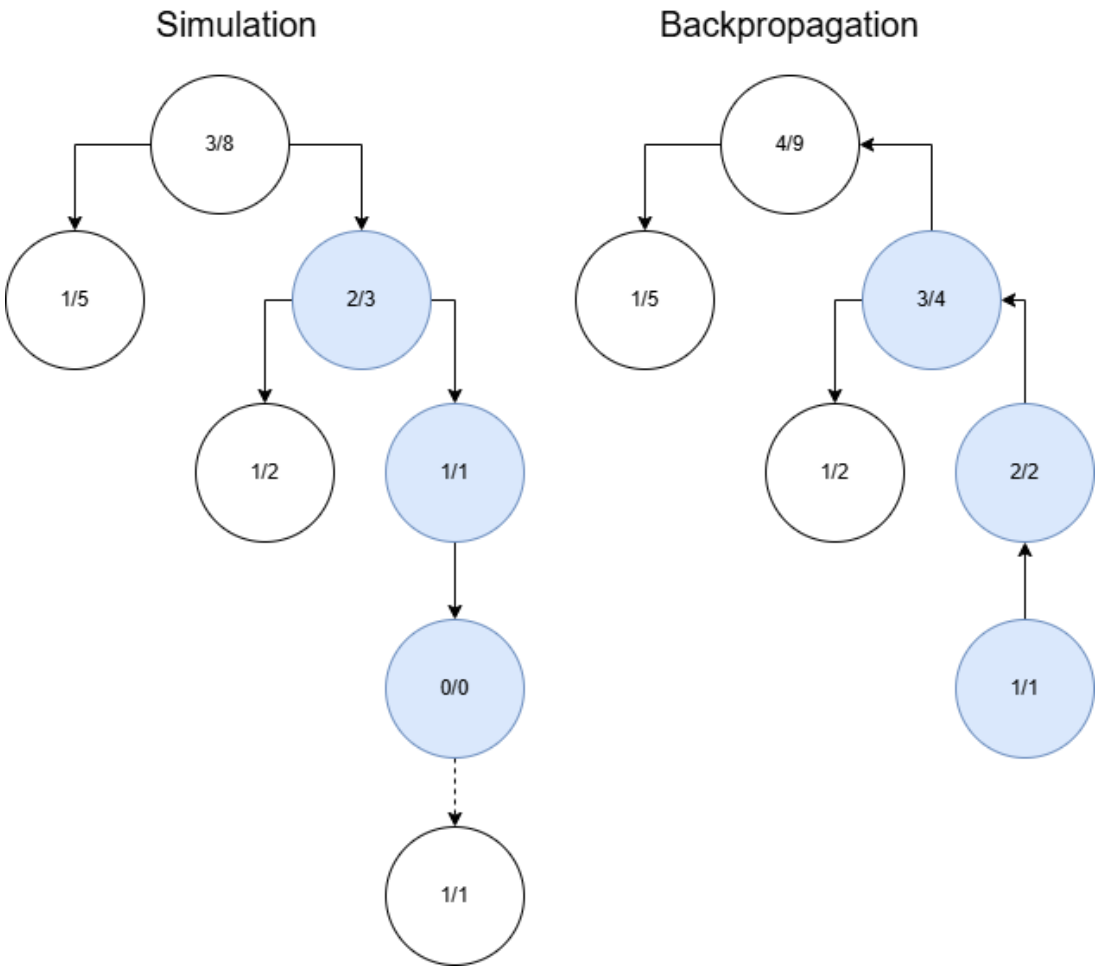


Abbildung 3.3: Simulation und Backpropagation

In diesem Beispiel wird ein Spiel zufällig bis zum Spielende simuliert und der Spieler gewinnt das Spiel. In der *Backpropagation*-Phase wird sowohl der Sieg, als auch das gespielte Spiel bis zum Wurzelknoten zurückgegeben und eine neue Iteration beginnt. Der Algorithmus läuft entweder für eine festgelegte Iterationsanzahl oder eine bestimmte Zeit. Anschließend wird unabhängig von der gewählten Selektionsstrategie der Kindknoten mit dem besten Quotienten aus Siegen und Spielen gewählt. Der MCTS für diese Bachelorarbeit bedient sich dieser Standardimplementierung, hat jedoch einige gravierende Änderungen. Die genauere Implementierung und welche Anpassungen vollzogen wurden, wird in Abschnitt 6.2 beschrieben.

4 Verwandte Arbeiten

[EN] In diesem Kapitel werden Arbeiten und Studien betrachtet, welche sich mit dem Thema MCTS und explizit dem *Pruning* des Aktionsraumes beschäftigen. Aufgrund der Größe und Relevanz des Themas werden ausschließlich bedeutende Paper sowie Studien berücksichtigt, die sich mit ähnlichen Fragen wie diese Bachelorarbeit befassen. Zudem werden verwandte Spiele analysiert und die Methoden zur Entwicklung der KI in diesen Spielen betrachtet.

4.1 Monte Carlo Tree Search

[EN] In der Vergangenheit hat sich MCTS in vielen klassischen Strategiespielen als bewährte Methode für einen Suchbaum etabliert [SCS09], [Roe12], [Sam18], [ZHR19], [JWF18], [RFT14], [AHH10], [CF10], [Win17]. Nach erstmaliger Erwähnung in [Cou06] wurden die größten und bekanntesten Erfolge des Algorithmus im Spiel Go, Schach und Shogi (die japanische Schach-Variante) erreicht [SHS+17], [SHS+18], [SHM+16], indem der Computer jeweils gegen die besten Spieler dieser Domäne mit Leichtigkeit gewann. Einer der größten Meilensteine war dabei der erstmalige Sieg eines Computers gegen den damaligen Großmeister Lee Sedol in einem vom 9. bis zum 15. März 2016 stattgefundenen 5-Spiele-Schach-Turnier. Seitdem ist die Weiterentwicklung des MCTS-Algorithmus stark gewachsen.

Für die unterschiedlichen Phasen des Algorithmus haben sich verschiedenste Varianten herauskristallisiert, von denen sich einige als dominant erwiesen haben. Das Paper [KS06] befasste sich das erste Mal mit der Einführung des UCB, um *Exploitation* und *Exploration* in der *Selection*-Phase gewissenhaft abzuwägen. Diese Form der Selektionsstrategie hat sich im Verlaufe der Zeit als praktikabel erwiesen.

Für die Simulationen, oder auch *Rollouts* genannt, sind bei performanceintensiven Spielen frühzeitige Terminierungen wichtig, da Simulationen bis zum Spielende zu zeitaufwendig wäre. Wie in Abschnitt 5.3 beschrieben, ist *Quick Civ* ebenfalls ein Spiel, welches keine Simulationen bis zum Spielende ermöglicht. In dem Paper [Lor15] wird dieses Thema intensiv betrachtet. Die *Rollouts* des MCTS-Algorithmus werden nach wenigen Zügen terminiert und der resultierende Spielzustand aufgrund von Domainwissen evaluiert.

Andere Arbeiten haben für die Evaluierung des Spielzustandes ein neuronales Netz verwendet, welches den Zustand einliest und eine Bewertung ausgibt [GSLL16], [SHS+17], [SHS+18], [CGDNV+21].

Neben der bereits angesprochenen langen Berechnungszeit ist ein sehr breiter Suchbaum eine weitere Hürde für die Qualität des MCTS-Algorithmus. [GS11] befasst sich mit der Heuristik *All-Moves-As-First*, die jeder Aktion einen Wert zuweist, welcher aus dem

Durchschnitt aller Simulationen, in denen die Aktion verwendet wird, berechnet wird. Dieser Wert wird dann verwendet, um zu entscheiden, welche Aktion in der *Selection*-Phase ausgewählt wird. Dies setzt voraus, dass die Qualität einer Aktion unabhängig davon ist, zu welchem Zeitpunkt jene Aktion ausgeführt wird, was auch für das Spiel *Quick Civ* zutrifft. In dieser Arbeit wird sich auf *Pruning* und die neue Methode *MCTS-Memory* beschränkt, um das Problem des breiten Suchbaums anzugehen.

Da MCTS ein breites Forschungsgebiet ist, gibt es unzählige Anpassungen, um die Performance je nach Domäne zu steigern. In den Papern [BPW+12] und [ŚGSM23] werden deswegen als Überblick die verschiedensten Anpassungen des Algorithmus zusammengefasst.

4.2 Action Space Pruning

[EN] In Abschnitt 5.2 wird ausführlich beschrieben, dass der MCTS auf Action Space Pruning angewiesen ist, um bessere Ergebnisse zu erzielen. Das Problem eines großen Suchraums ist bekannt und wurde bereits in einigen Studien bearbeitet. Da Pruning zumeist Domainwissen erfordert, ist jedes dieser Paper lediglich Inspirationen und können nicht trivial übernommen werden.

[JTTR14] befasst sich mit dem **Clustering** von Einheiten in dem RTS-Spiel *StarCraft*. Aktionen werden in dieser Pruning-Variante nicht einzelnen Einheiten zugeordnet, sondern gleichzeitig auf eine größere Gruppe an Einheiten angewandt. Durch diese Änderung wird der Aktionsraum deutlich verkleinert, was eine tiefere Baumsuche ermöglicht. Diese Anpassung ließe sich im Spiel *Quick Civ* zwar theoretisch umsetzen, indem man mehrere Felder gruppiert und bei allen das gleiche Gebäude baut, jedoch würde dies zu erheblichen Nachteilen führen, da es für die Effektivität der Gebäude sehr wichtig ist, welche anderen Gebäude daneben gebaut werden. Deshalb ist das Clustering der Felder nicht vorteilhaft.

In dem Paper [MMLN18] wird die **NaiveSampling**-Methode behandelt. In RTS-Spielen ist es oft der Fall, dass der Spieler eine große Menge an Einheiten kontrolliert. Durch NaiveSampling werden die einzelnen Entscheidungen, welche Aktionen die Einheiten ausführen sollen, als alleinstehende *Multi-Armed-Bandit*-Probleme behandelt, welche durch einen globalen *Multi-Armed-Bandit* verbunden sind. Dadurch wird verhindert, dass ein riesiger Aktionsraum entsteht, bei dem alle Einheiten mit all ihren Aktionen betrachtet werden. Diese Methode des NaiveSampling wird in dem Paper durch Domainwissen noch erweitert und dadurch verbessert, indem tendenziell schlechtere Aktionen herausgefiltert werden. Die Methode lässt sich für *Quick Civ* nicht anwenden, da in dem Strategiespiel keine Parallelität, wie in den RTS-Spielen, herrscht. Anstatt mehreren Einheiten Aktionen zuzuordnen, bedarf es in *Quick Civ* lediglich einer einfachen Aktionsauswahl aus einer sehr großen Menge an Bauaktionen.

Action Abstraction via Scripts ist eine weitere bekannte Pruning Methode, welche darauf beruht, anhand von *Highlevel-Scripts* zu entscheiden, welche Aktionen in der aktuellen Spielsituation betrachtet werden. Beispielsweise wird in RTS durch die *Scripts* entschieden, ob der Spieler gerade angreifen soll, oder nicht. Je nachdem, wofür das *Script* ausschlägt, werden demnach Aktionen zum Angreifen herausgefiltert. [ML18],

[DHGJ+21]. Diese Methode erfordert Domainwissen und ließe sich auch für *Quick Civ* umsetzen.

Move Pruning ist eine Methode, um bestimmte Aktionen anhand von Domainwissen auszuschließen, um damit die Größe des Suchraums zu verringern [OSM20]. Da diese Methode auf das Spiel explizit angepasstes Wissen benötigt, sind andere Arbeiten auf diesem Bereich wenig relevant für die Ausarbeitung geeigneter Pruningverfahren für den MCTS dieser Bachelorarbeit. Jedoch ist dieser Punkt hier zu erwähnen, da auch für *Quick Civ* Move-Pruning Methoden erstellt wurden, wie in Kapitel 6 ausführlich aufgezeigt wird.

4.3 Verwandte Spiele

[ET] 4X-Spiele sind häufig für ihre schlechte KI bekannt. Das liegt hauptsächlich daran, dass die meisten dieser Spiele viele komplexe Systeme haben und eine optimierte KI nicht im Fokus der Entwicklung steht. Deshalb und weil die Unternehmen ihre KI-Entwicklungen nicht veröffentlichen müssen, gibt es relativ wenig Forschung auf diesem Gebiet. Trotzdem lässt sich aus Interviews und durch Reverse Engineering (dem Analysieren des Programmcodes) zumindest schätzen, wie die KI in den großen Vorreitern funktioniert. Zum Beispiel wurden in der Civilization-Reihe in Teil 2 Utility-AI ([Wikb]) verwendet und in Teil 6 eine Mischung aus Utility-AI und *Decision Trees* (genauer beschrieben in Kapitel 3) [CIV]. Europa Universalis IV (kurz EUIV) dagegen benutzt für die Wirtschaftsentscheidungen einen greedy-Algorithmus, hat dafür aber ein ausgeprägteres Diplomatiesystem, welches versucht den stärksten Spieler aufzuhalten und eine Kampfberechnung, welche in der Lage ist, auch fähige Angriffskriege zu führen [EUI]. Sowohl Civilization VI als auch EUIV geben der KI auf höheren Schwierigkeitsgraden diverse Boni, um die schlechte KI auszugleichen. Man kann daran gut erkennen, dass diese Algorithmen hauptsächlich dafür da sind, einen interessanten Gegner zu kreieren und dafür gerne auf tatsächliche KI-stärke verzichten. Für das deutlich weniger komplexe Strategiespiel *Free Civ* wurde auch schon ein Knowledge Based Reinforcement Learning-Verfahren angewendet [VNSB20]. Die resultierende KI war auch deutlich stärker als die gegebene Computer-KI. Für simplere Strategiespiele wie Schach und Go wurden schon viele Algorithmen angewendet. Darunter auch häufig MCTS, aber auch Reinforcement Learning durch Neuronalen Netze wie z.B. bei Alpha Zero. Diese Spiele bieten allerdings keinen besonders sinnvollen Vergleich, da der Komplexitätsunterschied zu groß ist (siehe Abschnitt 5.2, Abschnitt 5.3).

4.4 MCTS mit Memory

[ET] Die Idee, den MCTS mit Hilfe von Speicher zu optimieren, ist in einigen Quellen zu finden. Dabei gibt es verschiedene Ansätze wo man mit Memory optimieren kann. Einige Quellen beschäftigen sich damit, bestimmte Zwischenwerte des MCTS abzuspeichern und so die Performance zu erhöhen, aber die meisten davon sind entweder nicht auf *Quick Civ* anwendbar oder zumindest sehr verschieden zu dem Memory-MCTS Algorithmus, der in dieser Arbeit Anwendung findet. Eine weitere relativ häufig verwendete Memory-Optimierung ist das wiederverwenden des Suchbaums. Dabei wird sich der

Suchbaum nach den Berechnungen gespeichert und kann dann im nächsten Spielzug wiederverwendet werden, in dem man mit dem Suchbaum der Aktion startet, für die sich im vorherigen Spielzug entschieden wurde. Dann muss man zumindest diesen Teil des Baums nicht erneut berechnen, da er identisch zum alten wäre [ABCS20]. Dieser Ansatz ist bei *Quick Civ* aber faktisch nicht umsetzbar, da es sowohl für den Spieler als auch die Gegner so viele Aktionen gibt, dass der exakte GameState des nächsten Zuges entweder nicht im Suchbaum auftaucht oder der dazugehörige Baum so klein ist, dass dies vernachlässigbare Einsparungen bringt. Ein anderer Ansatz evaluiert States, indem es sie mit den schon berechneten States vergleicht und ihnen entsprechend Score zuweist, je nach dem wie ähnlich sich die States sind [XMM18]. In einem letzten Paper werden Aktionen wiederverwendet, die innerhalb des Suchbaums schon einmal evaluiert wurden [XNP+24]. Dieser Ansatz ist dem Memory-MCTS dieser Arbeit am ähnlichsten, jedoch mit dem Unterschied, dass der Memory-MCTS die Evaluierung über mehrere Runden speichert.

5 Quick Civ

[ET] *Quick Civ* ist ein 4X-Strategiespiel, welches im Rahmen dieser Bachelorarbeit entwickelt wurde. Es handelt sich dabei um ein von [Gam23] inspiriertes Rundenstrategiespiel, mit dem entscheidenden Unterschied, dass das Spiel deutlich einfacher und schneller als die Vorlage ist. Der Fokus dieser Arbeit soll zwar auf der Gegner-KI im Spiel liegen, aber um diese und die entsprechenden Anpassungen verstehen zu können, werden in diesem Kapitel die Mechaniken von *Quick Civ* erklärt.

5.1 Spielbeschreibung

[ET] *Quick Civ* ist ein rundenbasiertes 4X-Strategiespiel. Das heißt es spielen mehrere Spieler (menschlich oder KI) auf einer großen Karte, um am Ende die meisten Punkte zu sammeln. Das Spielfeld besteht aus vielen Sechseckfeldern, auf denen sich jeweils 0-3 Naturelemente befinden. Aktuell gibt es zwei verschiedene Naturelemente: Wald (grünes Quadrat) und Berg (graues Quadrat).

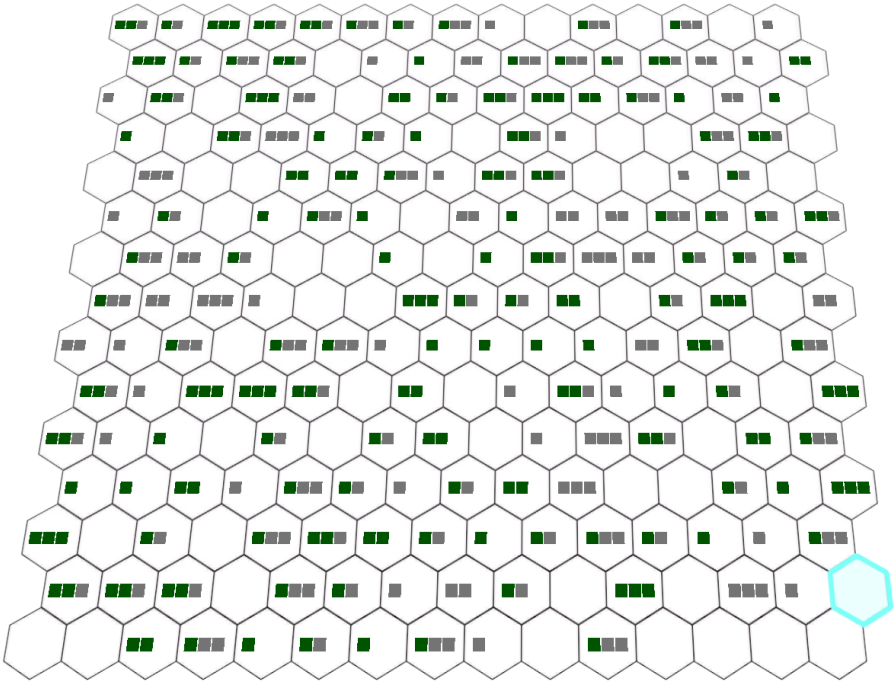


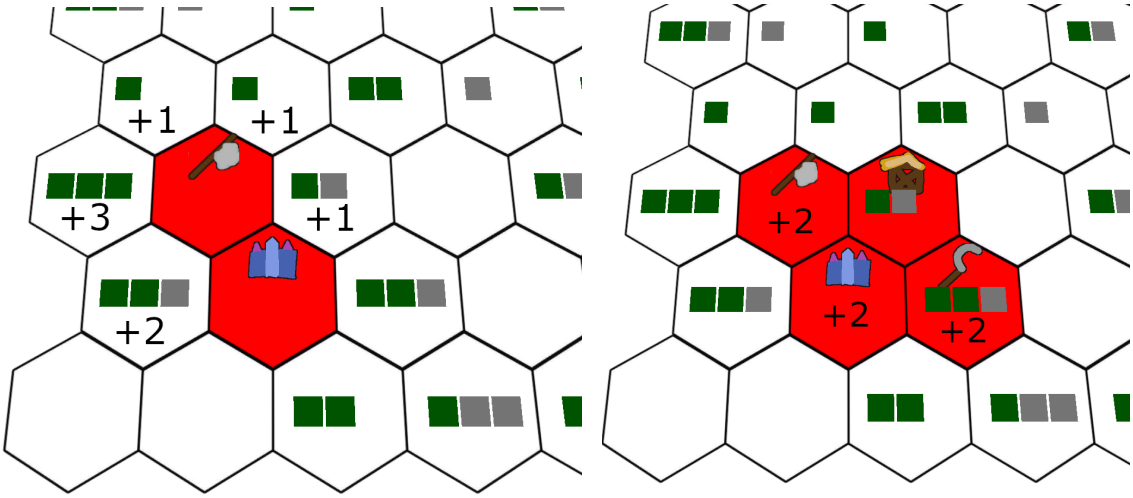
Abbildung 5.1: Spielfeld

Zum Start des Spiels beginnt jeder Spieler mit einem Stadtzentrum in einer Ecke. Von dort aus versuchen sich die Spieler weiter auszubreiten. Da das Spiel möglichst einsteigerfreundlich sein soll, gibt es nur eine Aktion, die die Spieler machen können: Das Bebauen

eines Hex-Feldes. Hierfür existieren 28 verschiedene Gebäude, die man bauen kann. Ein Überblick über die Gebäude befindet sich in Abschnitt B..

Es gibt im Spiel drei verschiedene Ressourcen, die verschiedene Aufgaben erfüllen. Nahrung wird benötigt, um unbebaute Felder das erste mal zu bebauen. Je weiter das Feld von einem Stadtzentrum entfernt ist, desto höher sind die Nahrungskosten. Material wird immer benötigt, wenn man ein Gebäude baut. Hierbei wird der Preis nur von der Art des Gebäudes bestimmt. Punkte sind die letzte Ressource. Sie werden nicht für das eigentliche Spielen gebraucht, sondern sind am Ende für die Auswertung wichtig. Der Spieler, der nach 100 Runden die meisten Punkte hat, gewinnt das Spiel.

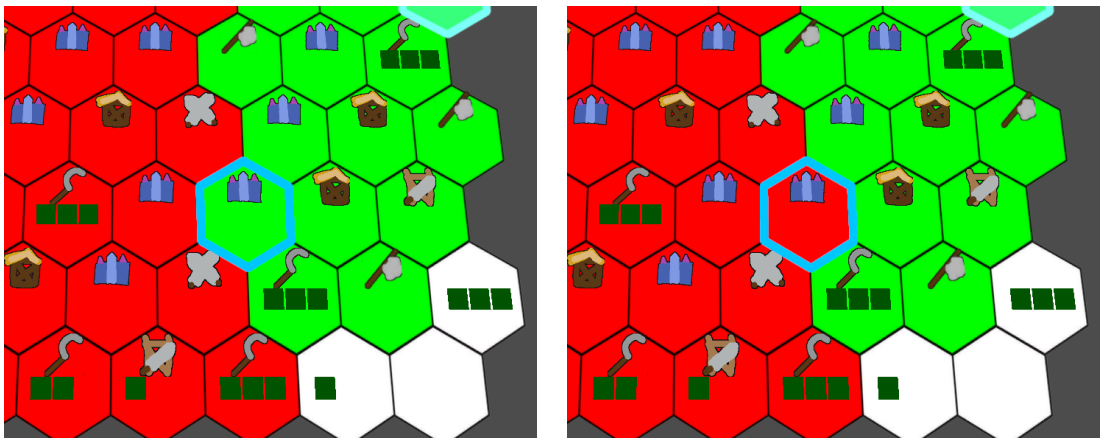
Die Gebäude geben alle verschiedene Boni und produzieren die verschiedenen Ressourcen. Wie viel die Gebäude produzieren, hängt maßgeblich davon ab, wo sie gebaut werden. Ein Holzfäller produziert z.B. für jeden angrenzenden Wald ein Material pro Runde. Ein Wohnviertel produziert dagegen für jedes angrenzende bebaute Feld 2 Material, wie in Abbildung 7.10 zu sehen. Die Komplexität des Spiels entsteht also hauptsächlich durch die Vielzahl an verschiedenen Gebäuden, welche durch ihre Platzierungen alle miteinander interagieren können.



(a) Holzfällerboni **(b)** Hausboni

Abbildung 5.2: Boni für verschiedene Gebäude

Kommt es zu einer Konfrontation mit einem anderen Spieler bieten sich zwei Optionen: Erstens kann man friedlich mit dem anderen Spieler koexistieren und mit dem Handelsposten ein Gebäude bauen, welches von fremden Feldern profitiert. Zweitens kann man gegen den anderen Spieler kämpfen, indem man Militärgebäude baut und versucht damit seine Felder einzunehmen. Das Militärsystem funktioniert dabei so, dass Militärgebäude militärischen Druck auf angrenzende Felder ausüben. Übt man über drei Runden mehr Druck auf ein Feld aus als sein Besitzer, übernimmt man die Kontrolle des Feldes wie in Abbildung 5.3 zu sehen ist.



(a) Vor dem Konflikt **(b)** Nach dem Konflikt

Abbildung 5.3: Militär

5.2 Komplexität

[ET] MCTS ist ein vielfach eingesetzter Algorithmus, der auch schon bei anderen Strategiespielen Anwendung gefunden hat. Allerdings ist hier eine Einordnung zwingend notwendig, da sich verschiedene Strategiespiele unterschiedlich gut für den Algorithmus eignen. Eine Stärke des MCTS liegt zwar darin, aus einem Suchbaum die vielversprechendsten Aktionen schnell rauszufinden und dann tiefer zu erforschen, aber das funktioniert nur so lange, wie jede Startmöglichkeit auch ausreichend exploriert werden kann. Bei *Quick Civ* ist dies später im Spiel häufig nicht mehr möglich, da der Action Space zu groß wird.

Um dieses Problem nochmal zu verdeutlichen folgt ein Vergleich zwischen Go (MCTS wurde vielfach für Go eingesetzt) und *Quick Civ*:

	Go	Quick Civ
Anzahl mögliche Spielzustände	19×19 Felder mit jeweils 3 Zuständen (schwarz, weiß, leer) ergibt $1,74 \times 10^{172}$	15×15 Felder mit jeweils 113 Zuständen (28 Gebäude je 4 Spieler, unbebaut) ergibt $6,26 \times 10^{265}$
Durchschnittsanzahl an Aktionen pro Runde	Hälfte aller Felder: 180,5	ca. 50 Felder und ca. 20 Gebäude: 1000
größtmögliche Anzahl an Aktionen pro Runde	alle Felder: 361	ca. 150 Felder und 28 Gebäude: 4200

Tabelle 5.1: Komplexitätsvergleich

Die möglichen Spielzustände für *Quick Civ* sind nicht ganz akkurat sondern nur eine obere Abschätzung. Sie sagen aber ohnehin nur bedingt etwas über die Komplexität aus. Die durchschnittlichen Anzahl Aktionen pro Runde sind deutlich wichtiger und diese zeigen deutlich, dass MCTS bei *Quick Civ* deutlich mehr Aktionen zu beurteilen hat, als bei Go. Dies wird sogar noch wichtiger, da der Aufwand nicht linear, sondern exponentiell mit

der Anzahl an Aktionen steigt. Das kommt daher, dass pro Schritt den der Algorithmus tiefer geht, er theoretisch wieder den gesamten Action Space überprüfen muss, bis er tiefer gehen darf.

5.3 Performance

[ET] Als weitere Schwierigkeit erwies sich die Performance von *Quick Civ*. Während bei simpleren Strategiespielen wie Go die Zustandsberechnung relativ einfach ist, müssen für *Quick Civ* bei jeder Aktion Ressourceneinkommen, Militärwerte usw. berechnet werden. Während Spiele normalerweise so optimiert werden, dass sie in Echtzeit spielbar sind, wurde *Quick Civ* so stark optimiert, sodass jetzt über 10000 Zustände pro Sekunde berechnet werden können. Diese Optimierung ist auch zwingend notwendig, um MCTS auf ähnliche Spiele oder Projekte anzuwenden. Deswegen sollte es gut bedacht sein, ob man MCTS für Spiele-KI einsetzt, da dann die Performance definitiv eine deutlich größere Rolle spielen wird. Außerdem muss das Spiel ein Forward-Model besitzen. Das heißt man muss aus dem aktuellen Zustand alle möglichen Aktionen extrahieren und mit diesen neue Spielstände simulieren können, ohne den originalen Zustand zu verändern. Da dies in den meisten Spielen nicht notwendig ist, wurde das Spiel hier auch deutlich anders entwickelt als das für gewöhnlich getan wird.

6 Methodik

[EN] Um die Forschungsfragen in Hinblick auf die Leistung des MCTS zu beantworten, bedarf es einiges an Vorleistungen. Wie in Kapitel 5 ausführlich beschrieben, wurde im Rahmen dieser Bachelorarbeit der Grundbaustein eines 4X-Spiels mit dem Namen *Quick Civ* entwickelt. Damit nicht-menschliche Spieler das Spiel spielen können, wurde die Funktion *getNextActions()* bereitgestellt, welche jede Runde für jeden teilnehmenden Spieler, egal ob von Mensch oder Maschine gesteuert, aufgerufen wird. Menschliche Spieler können anschließend durch Eingabegeräte ihre Aktionen deklarieren, wohingegen die nicht-menschlichen Spieler ihren Algorithmus nutzen, um entsprechende Aktionen auszuwählen. Zur Verfügung stehen allen Spielern der aktuelle Spielzustand mit allen möglichen Aktionen, die Ressourcen (-produktion) jedes Spielers und ein *Forward-Modell*. Dieses Modell beschreibt eine Übergangsrelation von einem Spielzustand, über eine Menge an Aktionen, zu einem resultierenden Spielzustand. Auf diese Weise können die nicht-menschliche Spieler ermitteln, wie das Spiel nach bestimmten Aktionen aussieht und daraufhin Entscheidungen und Bewertungen treffen. Dieses *Forward-Modell* ist der menschlichen Vorstellungskraft gleichzusetzen, wenn der Spieler nachdenkt, was passieren wird, wenn er oder die Gegner bestimmte Aktionen ausführen.

Mit der Schnittstelle *getNextActions()* wird also gewährleistet, dass sowohl menschliche, als auch nicht-menschliche Spieler mit dem Spiel interagieren können. Im weiteren Verlauf dieses Kapitels wird aufgezeigt, welche Arten von nicht-menschlichen Spielern implementiert wurden, um schlussendlich die Forschungsfragen beantworten zu können.

6.1 Random und Greedy

[EN] Zur Bewertung verschiedener Herangehensweisen nicht-menschlicher Spieler werden zwei einfache Strategien als Benchmark verwendet: die *Random*-Strategie und die *Greedy*-Strategie. Sie dienen als Referenz für den Vergleich mit komplexeren Entscheidungsverfahren und ermöglichen eine erste Einschätzung der Leistungsfähigkeit von Algorithmen in diesem Spiel.

Die **Random**-Strategie stellt die einfachste Form der Entscheidungsfindung dar. Hierbei wählt der nicht-menschliche Spieler in jeder Runde zufällig eine Aktion aus den verfügbaren Aktionen aus. Diese Strategie erfordert keinerlei Bewertung oder Planung, sondern basiert ausschließlich auf Zufall. Obwohl sie nicht darauf ausgelegt ist, das Spiel effektiv zu gewinnen, dient sie als Referenzpunkt für den Vergleich mit intelligenteren Strategien.

Code 6.1: Random-Bot

```
function get_next_actions():
    result = []
    possibleActions = getPossibleActions(currentGameState)
  if possibleActions is not empty:
      randomAction = random.Choice(possibleActions)
        result.Add(randomAction)

      return result
```

Ein Vorteil der *Random*-Strategie ist ihre einfache Implementierung und geringe Rechenanforderung. Sie ermöglicht es, grundlegende Mechaniken des Spiels zu testen und bietet eine untere Leistungsgrenze für andere Strategien. In den durchgeführten Simulationen zeigt sich logischerweise jedoch schnell, dass sie keine wertvollen Ergebnisse zeigt.

Die **Greedy**-Strategie verfolgt einen deterministischen Ansatz, indem sie in jeder Runde die kurzfristig beste Aktion wählt. Dies geschieht anhand einer Evaluierungsfunktion, die den unmittelbaren Nutzen einer Aktion bewertet. Da Stadtzentren für das Spiel essentiell sind, jedoch keinen unmittelbar spürbaren Nutzen mit sich bringen, erhalten *Greedy-Bots* einen Counter, welcher dafür sorgt, dass alle x-Runden ein Stadtzentrum gebaut wird.

Code 6.2: Greedy-Bot

```
function get_next_actions(cityCentreCounter):
    result = []

    if(cityCentreCounter is 0):
        cityCentreCounter = turnsForCityCentre
        return result.Add(buildCityCentreAction)
    else:
        cityCentreCounter.decrement()

      possibleActions = getPossibleActions(currentGameState)

  if possibleActions is not empty:
    for action in possibleActions:
        newGameState = getNewGameState(currentGameState, action)
        value = getValue(newGameState)
        if value > bestValue:
            bestValue = value
            bestAction = action

  if bestAction is not None:
      result.Add(bestAction)

return result
```

Ein zentraler Aspekt der *Greedy*-Strategie ist die Wahl einer geeigneten Bewertungsfunktion. Diese Bewertungsfunktion, im Pseudocode mit *getValue* bezeichnet und in Gleichung 6.2 zu sehen, ist nicht nur für den *Greedy-Bot* essentiell, sondern auch später für den MCTS-Algorithmus, welcher auf eine Bewertungsfunktion zurückgreift. Das interessante der Bewertungsfunktion ist die Abwägung zwischen Produktionsverbesserung und Punktegenerierung. Wie in Kapitel 5 beschrieben, sind die Punkte zwar essentiell, um das Spiel zu gewinnen, jedoch braucht man genug Ressourcen, um sein Reich zu vergrößern und so die spätere Punktegenerierung zu gewährleisten. Aus diesem Grund ist eine Bewertungsfunktion entstanden, die sowohl die Ressourcenproduktion, als auch die Punktegenerierung abwägt, und zwar abhängig zum Verlauf des Spielfortschritts, wie in

Gleichung 6.1 zu sehen ist. Dadurch wird gewährleistet, dass die Spieler am Anfang des Spiels die Ressourcenproduktion und dann ab einem bestimmten Zeitpunkt immer mehr Punktegenerierung priorisieren.

$$\text{gameProgression} = \text{clamp}\left(\frac{\text{currentRound} \cdot \text{pointInc}}{\text{finalRound}}, 0, 1\right) \tag{6.1}$$

$$\begin{aligned} \text{E(s)} = (\text{mp} \cdot \text{material} + \text{fp} \cdot \text{food} + \text{bp} \cdot \text{material} \cdot \text{food} + \text{tp} \cdot \text{tiles}) \cdot (1 - \text{gameProgression}) \\ +(\text{pp} \cdot \text{points}) \cdot \text{gameProgression} \end{aligned} \tag{6.2}$$

Dabei stehen die Parameter für folgende Werte:

- **gameProgression**: Ein Wert zwischen 0 und 1, der den Spielfortschritt anhand der aktuellen Runde bestimmt. Durch **pointInc** kann der Wert schneller auf 1 gelangen, um schneller nur noch Punktegebäude zu bauen.
- **material**, **food**, **points**: Ressourcenproduktion des Spielers.
- **mp**, **fp**, **pp**: Parameter für die Gewichtung der Einzelbestandteile
- **tiles**, **tp**: Anzahl an bebauten Feldern des Spielers und dazugehöriger Parameter.
- **bp**: Parameter, um Gleichgewichtung von **material** und **food** zu belohnen.

Durch die einzelnen Parameter kann jeder Bot individuell auf verschiedene Taktiken eingestellt werden. Bots mit einem hohen Tile-Parameter (**tp**) bebauen vor allem neue Felder, anstatt alte zu überbauen und erobern gerne andere Spieler. Ein hoher **pointInc** sorgt dafür, dass man bereits früh nur noch ausschließlich Punktegebäude selektiert. Nach mehrfachen Testspielen und analysieren der Ergebnisse hat sich herausgestellt, dass ausgeglichene Parameter (alle Parameter haben den Wert 1, außer **PointInc** mit einem Wert zwischen 1.5 und 2) die beste Strategie für einen *Greedy-Bot* gewährleisten. Zudem sollte er alle 10-20 Runden ein Stadtzentrum bauen, was wie im Code 6.2 gezeigt, durch den Parameter (turnsForCityCentre) eingestellt werden kann. Durch diese Parameterwerte verfolgt der *Greedy-Bot* einen einfachen Plan: am Anfang baut er gleichmäßig Material- und Nahrungsgebäude und ab circa der Hälfte des Spiels ausschließlich Punktegebäude. So wird sichergestellt, dass er stets genug Ressourcen für den Ausbau der Punktegenerierung hat, ohne dabei zu spät mit dem Bau der Punktegebäude anzufangen. Da der *Greedy-Bot* nur als ein Benchmark für andere Algorithmen dient, genügt eine grobe Überlegung und das analysieren einiger Testspiele als Entscheidungsgrundlage für die Parameter.

Ein wesentlicher Nachteil dieser Methode ist die fehlende Weitsicht: Da immer nur der unmittelbar beste Zug gewählt wird, kann es dazu kommen, dass langfristige Vorteile zugunsten kurzfristiger Gewinne ignoriert werden. Dies führt in einigen Spielsituationen zu ineffizienten Entscheidungen, insbesondere wenn ein bestimmtes Gebäude erst durch die Kombination mit anderen Bauwerken seine Effektivität steigert. Außerdem hat der *Greedy-Ansatz* einen weiteren großen Schwachpunkt - das Militärsystem. Da Militärgebäude keinen direkten Vorteil in der Ressourcenproduktion bieten, werden sie von *Greedy-Bots* ignoriert. Dies führt dazu, dass sie im Spiel gegen bessere Spieler, egal ob

menschlich oder nicht-menschlich, regelrecht überrannt werden, da sie sich nicht verteidigen.

6.2 Vanilla-MCTS

[EN] Vorneweg muss erwähnt werden, dass die *MCTS-Bots* lange nicht so viele Simulationen in der Simulationsphase durchlaufen können, wie es zumeist üblich ist. Dies hat einfache Gründe, die bereits in Abschnitt 5.3 beschrieben wurden. Deswegen ist die Wahl der Parameter und die Entscheidung für die richtigen Strategien in den einzelnen Phasen des Algorithmus noch entscheidender, da nicht leichtsinnig x-beliebig viele Simulationen durchgeführt werden können, sondern stets Rechenzeit gegen Leistung abgewogen werden muss. In diesem Abschnitt wird erklärt, welche genaue Implementierung des MCTS umgesetzt wurde. Dabei entstand zumeist eine Vielzahl an Variationen, welche dann durch einen *Evolutionären Algorithmus*, näher beschrieben in Abschnitt 6.3, evaluiert wurden.

Um die optimale Strategie für die **Selection**-Phase herauszufinden, wurden alle in Unterabschnitt 3.2.2 beschriebenen Methoden implementiert, dazu gehören die *Random*-, *Greedy*-, *EpsilonGreedy*-, und die *UCB*-Strategie, da von Anfang an nicht offensichtlich war, welche der Strategien die effektivste ist. Zudem sind alle nötigen Parameter (z.B. *epsilon* für die Epsilon-Greedy Methode oder C für die UCB-Methode) modular veränderbar, sodass sie durch den *Evolutionären Algorithmus* angepasst werden können.

Um die UCB-Strategie korrekt anwenden zu können, ist eine Normalisierung der Werte erforderlich, da UCB nur für Werte im Intervall $[0, 1]$ zuverlässig funktioniert. Als Grundlage dient die Methode aus [VSS17], insbesondere Abschnitt 4.4. Die Normalisierung erfolgt vor der Anwendung der UCB-Formel, indem für jeden Knoten der beste und der schlechteste Wert unter seinen Kindknoten ermittelt wird. Dabei wird nicht direkt der gespeicherte Wert eines Kindes betrachtet, sondern der Mittelwert aus gesammeltem Wert und Besuchen, also der Quotient in Gleichung 6.3

$$Q(n) = \frac{\mathrm{V}(n)}{\mathrm{visits}(n)} \tag{6.3}$$

Die Normalisierung eines Wertes $Q(c)$ innerhalb eines Knotens erfolgt dann nach durch Gleichung 6.4.

$$Q_{\mathrm{norm}}(n) = \frac{Q(n) - Q_{\min}}{Q_{\max} - Q_{\min}} \tag{6.4}$$

wobei:

- $Q(n)$ der durchschnittliche Evaluierungswert eines Kindknotens ist,
- $Q_{\min}$ der schlechteste Durchschnittswert unter den Kindknoten ist,
- $Q_{\max}$ der beste Durchschnittswert unter den Kindknoten ist.

Durch diese Transformation werden alle Werte in den Bereich $[0, 1]$ skaliert, sodass die UCB-Formel korrekt angewendet werden kann.

Die **Expansion**-Phase erfordert wenig Anpassung, da sie relativ klar definiert ist. In dem verwendeten Algorithmus werden durch eine Funktion alle möglichen Aktionen des Spielzustands ermittelt und anschließend zufällig eine von ihnen ausgewählt und ausgeführt. Danach werden für jeden Gegner ebenfalls eine Aktion ausgewählt und auf den neuen Spielzustand angewandt. Der resultierende Spielzustand wird in den bestehenden Baum eingehängt und die selektierte Aktion wird aus den möglichen Aktionen des ursprünglichen Knotens gestrichen, damit sie nicht erneut in der Expansion-Phase betrachtet wird.

Die **Simulation**-Phase stellt den komplexesten und variabelsten Abschnitt des MCTS-Algorithmus dar. Wie bereits zuvor erwähnt, ist es nicht praktikabel, mehrere Hundert Simulationen bis zum Spielende durchzuführen. Daher wird die Anzahl der Simulationen begrenzt und nur bis zu einer bestimmten Tiefe ausgeführt, wobei anschließend die Evaluierungsfunktion $E(s)$ (siehe Gleichung 6.2) angewendet wird. Allerdings wird die Evaluierungsfunktion nicht nur auf den Spielzustand nach der Simulation angewandt, sondern die Differenz zum Ausgangszustand im Wurzelknoten des aktuellen Suchbaumes berechnet, siehe Gleichung 6.5

$$\Delta E(s) = E(s_{\text{end}}) - E(s_{\text{root}}) \tag{6.5}$$

Dadurch ergibt sich eine relative Veränderung des Spielzustands anstelle eines absoluten Werts. Dies hat zur Folge, dass die gespeicherten Werte in den Knoten, unabhängig von der Phase des Spiels, prozentual ähnlich bleiben.

Ein Beispiel verdeutlicht diesen Ansatz:

- **Aktuelle Spielrunde:** 20
- **Wert der Evaluierungsfunktion für den Wurzelknoten** $E(s_{\text{root}})$: 300
- **Wert der Evaluierungsfunktion nach einer Simulation, ausgehend von Spielzustand s** $E(s_{\text{end},1})$: 350 (z. B. durch den Bau eines vorteilhaften Gebäudes)
- **Wert der Evaluierungsfunktion nach einer alternativen Simulation, ausgehend von Spielzustand s** $E(s_{\text{end},2})$: 280 (z. B. durch den Bau eines ungünstigen Gebäudes)
- **Veränderung relativ zum Wurzelknoten:**
  - $\Delta E_1 = 350 - 300 = +50$
  - $\Delta E_2 = 280 - 300 = -20$

Vergleicht man stattdessen nur die absoluten Werte der Evaluierungsfunktion der Endzustände, so ergibt sich ein prozentualer Unterschied:

$$\frac{350}{280} \approx 1,25 \quad \text{(d. h. } +25\%\text{)}$$

Dieser relative Unterschied nimmt mit fortschreitendem Spielverlauf zwangsläufig ab, da einzelne Aktionen am Anfang des Spiels die Produktion verdoppeln können, während sie am Ende des Spiels maximal einen Vorteil von wenigen Prozenten bringen. Betrachtet man jedoch die Differenz zum Wurzelknoten, bleibt die Skala der Werte stabil und ermöglicht eine konsistente Bewertung über verschiedene Spielphasen hinweg.

Zusätzlich fließen die Evaluierungsfunktionen der Gegner ebenfalls in die Bewertung mit ein, um zu belohnen, den Gegnern zu schaden. Dazu wurde folgende Formel benutzt, siehe Gleichung 6.6.

$$V = \begin{cases} \Delta E(s) & \text{falls } w = 0 \\ \Delta E(s) - \Delta E^*(s) & \text{falls } w = 1 \\ \Delta E(s) - w \cdot \Delta E^*(s) - \sum\limits_{i \neq \text{self}, i \neq *} \frac{(1-w)}{N} \Delta E_i(s) & \text{sonst} \end{cases} \tag{6.6}$$

wobei:

- $\Delta E(s)$ die Evaluierungsfunktion vom Spielzustand des Spielers ist.
- $\Delta E^*(s)$ die Evaluierungsfunktion vom Spielzustand des besten Gegners ist.
- $\Delta E_i(s)$ die Evaluierungsfunktionen vom Spielzustand für andere Gegner sind.
- $w \in \mathbb{R}, \quad w \in [0, 1]$ der *WinnerBias-Parameter*, das Gewicht für den stärksten Gegner ist.
- N die Anzahl der Spieler ist.

Dieser Ansatz erlaubt eine differenzierte Bewertung der Simulationsergebnisse, indem nicht nur die eigene Veränderung im Spielzustand betrachtet wird, sondern auch die relative Entwicklung der Gegner einbezogen wird. Ein hoher w-Wert priorisiert den Vergleich mit dem stärksten Gegner, während kleinere Werte eine gleichmäßigere Berücksichtigung aller Spieler ermöglichen.

Die Simulationen können auf unterschiedliche Weise ausgeführt werden. Für die Umsetzung wurden verschiedene Varianten gewählt, die jeweils durch Parameter gesteuert werden können – sowohl für den Spieler als auch für die Gegner:

- Random: Eine zufällige Aktion wird aus allen möglichen Aktionen ausgewählt.
- Greedy: Die kurzfristig beste Aktion wird aus allen möglichen Aktionen ausgewählt.
- RandomWithoutOverwrite: Eine zufällige Aktion wird gewählt, wobei Aktionen ausgeschlossen werden, bei denen ein bereits bebautes Feld überbaut wird. Dies verstärkt die Simulation der Ausbreitung des Spielers bzw. der Gegner im Vergleich zur reinen Random-Strategie.
- RandomWithoutOverwriteWithMilitary: Diese Variante entspricht der Strategie *RandomWithoutOverwrite*, erlaubt jedoch das Überbauen von Militärgebäuden oder das Ersetzen bestehender Gebäude durch Militärgebäude. Dies trägt der Tatsache Rechnung, dass es im Spielverlauf häufig sinnvoll ist, Militärgebäude später zu überbauen oder andere Gebäude durch sie zu ersetzen.

- JustWaiting: In dieser Varianten werden keine Aktionen ausgeführt, sondern nur gewartet. Dies hat zur Folge, dass das Ergebnis von militärischen Prozessen abgewartet wird und das Ergebnis daraus evaluiert werden kann

Mit steigender Komplexität der Simulationsstrategie erhöht sich die Aussagekraft der einzelnen Simulationen, da diese realistischer an den tatsächlichen Spielverlauf angelehnt sind. Gleichzeitig führt eine höhere Komplexität jedoch zu längeren Rechenzeiten. Dadurch können auch einfachere *Random-Rollouts* effektiv sein, da sie aufgrund der geringeren Rechenzeit eine größere Anzahl an Simulationen ermöglichen.

Nach der Simulations-Phase liegen mehrere Ergebnisse aus den durchgeführten Simulationen vor. In der anschließenden **Backpropagation**-Phase werden diese Ergebnisse gemittelt und an die übergeordneten Knoten weitergegeben. Der durchschnittliche Wert aus den Simulationsergebnissen berechnet sich wie folgt in Gleichung 6.7

$$\bar{V} = \frac{\sum_{i=1}^{M} V_i}{M} \tag{6.7}$$

wobei:

- M die Anzahl der durchgeführten Simulationen ist.
- V die endgültige Bewertung des Spielzustandes ist.

Dieser gemittelte Wert wird dann entlang des Suchbaumes propagiert. Der Wert eines Knotens $V(n)$ wird dabei gemäß Gleichung 6.8 aktualisiert:

$$V(n)_{\text{neu}} = V(n)_{\text{alt}} + \bar{V} \cdot \frac{1}{\max(\text{vanishingBackprop}, 0.1) \cdot d} \tag{6.8}$$

und die Besuchszahl $\text{visits}(n)$ eines Knotens wird entsprechend Gleichung 6.9 angepasst:

$$\text{visits}(n)_{\text{neu}} = \text{visits}(n)_{\text{alt}} + \frac{1}{\max(\text{vanishingBackprop}, 0.1) \cdot d} \tag{6.9}$$

wobei:

- d die Tiefe der Propagierung bezeichnet.
- vanishingBackprop ein Parameter ist, der bestimmt, wie stark der Wert abgeschwächt wird. Dabei steht ein Wert von 1 für eine sehr starke Abschwächung. Dieser Wert ist mindestens 0.1, da der Algorithmus nur für vanishingBackprop > 0 funktioniert.

Anschließend wird die gleiche Aktualisierung für den Elternknoten durchgeführt, wobei die Tiefe d um eins erhöht wird, bis die Wurzel des Suchbaums erreicht ist. Auf diese Weise werden die Simulationsergebnisse sukzessive in den gesamten Baum zurückgeführt, wobei weiter entfernte Knoten mit einem entsprechend skalierten Einfluss aktualisiert werden.

6.3 Hyper Parameter Optimization

[EN] Wie anfangs erwähnt, existieren eine Vielzahl an Variationen für jede der vier zentralen MCTS-Phasen. Um herauszukristallisieren, welche Methode nun die effektivste ist und welche Parameterkonstellationen sich am erfolgversprechendsten darstellen, wird ein Evolutionärer Algorithmus (EA) verwendet [ESES15]. Zusammengefasst besteht ein EA aus vier wesentlichen Abschnitten, welche, mit Ausnahme der Initialisierung, solange iteriert werden, bis eine Terminierungsbedingung erfüllt ist. Folgender EA wurde verwendet, um eine gute Kombination aus Parametern für den *Vanilla MCTS* zu extrahieren.

- **Initialisierung:** Die Hälfte der Population wird mit *Greedy-Bots* initialisiert, die andere Hälfte besteht aus *MCTS-Bots*. Bei der Initialisierung werden die Parameter zufällig initialisiert, innerhalb vordefinierter Grenzen. Die genauen Werte sind im Abschnitt A. nachzuschlagen
- **Fitnessfunktion:** Die Bots spielen eine festgelegte Anzahl an *Matches* gegeneinander. Ein *Match* ist ein Runde *Quick Civ* bis Runde 100. Der gewonnene Fitnesswert nach einem *Match* berechnet sich auf Grundlage von Gleichung 6.10.

$$\Delta Fitness(i) = \frac{\text{Points}(i)}{\text{Points}_{\text{max}}} \tag{6.10}$$

  , wobei $\text{Points}_{\text{max}}$ die Höchstpunktzahl des gespielten *Matches* beschreibt
- **Selektion:** Elitismus - die x besten Individuen überleben die Generation.
- **Variation:** Die überlebenden Individuen werden durch *Crossover* reproduziert und anschließend mutiert, bis die Ausgangspopulationsgröße erreicht ist. Dabei wird ein *uniformes Crossover* verwendet. Diese Methode erstellt das Kindindividuum aus den zwei Elternindividuen, für jedes Allel mit einer Wahrscheinlichkeit von 50% das Gen entweder vom ersten oder vom zweiten Elternteil übernimmt. Eine Ausnahme dazu bilden bestimmte Parameter (hier: *epsilon* für die *Greedy*-Selektionsstrategie und C für die *UCB*-Selektionsstrategie), die nur dann vom Elternteil übernommen werden, wenn sie auch die jeweilige Selektionsstrategie übernommen haben. Die *Mutation* bewirkt eine geringfügige Änderung jeden Gens, zu einem bestimmten Prozentsatz. Auch dabei werden nur die eben genannten Parameter mutiert, sofern das Individuum auch die jeweilige Selektionsstrategie geerbt hat.

Da durch die Variationsphase exakt die Anfangspopulationsgröße erreicht wird, benötigt diese Hyper Parameter Optimization (HPO) keine Umweltselektion.

6.4 Action Space Pruning

[EN] Wie bereits in Abschnitt 5.2 beschrieben, ist der Action-Space zu groß für den MCTS-Algorithmus, um erfolgreich die beste Aktion zu finden. Action Space Pruning ist eine Methode, um den Suchraum zu verringern, indem wenig erfolgversprechende Aktionen

nicht betrachtet werden. In dieser Bachelorarbeit werden zwei unterschiedliche Kategorien von Action Space Pruning betrachtet. Zum einen existiert der Ansatz, durch Domainwissen schwache Aktionen herauszufiltern. Die alternative Methode ist der neue Ansatz des *Memory-MCTS*, der in Abschnitt 6.5 näher beschrieben wird. Um Domainwissen zum Filtern von Aktionen in dem Spiel *Quick Civ* anzuwenden, wurden vier verschiedene Ansätze implementiert, die im Folgenden genauer beschrieben werden. In Kapitel 7 wird dann überprüft, wie effektiv jene Pruning-Ansätze sind. Tabelle 6.1 bietet einen Überblick über die verschiedenen Pruningvarianten.

Name	Beschreibung
OnlyMilitaryOverwrites	Lediglich Miitär- und Punktgebäude dürfen überbaut werden oder andere Gebäude überbauen
OnlyGreedyNodes	Es werden nur Aktionen betrachtet, die eine Verbesserung des Ergebnisses der Bewertungsfunktion bewirken
OnlyBuildNearLastAction	Der Action Space filtert alle Aktionen, die zu weit von der zuletzt-ausgeführten Aktion entfernt sind.
InLateGameNoEarlyGameBuildings	Sobald ein Gebäude der teuersten Gebäudeklasse gebaut wird, schließt dieses Pruning alle Gebäude der günstigsten Gebäudeklasse aus.

Tabelle 6.1: Überblick Pruningvarianten

6.4.1 Only Military Overwrites Pruning

[EN]

Diese Pruning-Methode filtert alle *Überbauen*-Aktionen heraus, welche keine Militär- oder Punktgebäude beinhalten. Dabei ist egal, ob mit einem Militär- bzw. Punktgebäude überbaut wird, oder ein Feld auf dem bereits ein Militär-bzw. Punktgebäude steht, überbaut wird. Diese Methode beruht auf der Annahme, dass wenn man (vor allem zu Spielbeginn) effektiv sein Reich aufbaut, kein Grund besteht, bebaute Felder erneut zu überbauen. Militärgebäude bilden eine Ausnahme, da sie in Angriffsszenarien schnell geändert werden müssen, falls man den Gegner angreifen will, sich verteidigen möchte oder sein Reich wieder auf Produktion auslegen möchte, wenn der Krieg vorbei ist. Punktgebäude sind zusätzlich von dem Pruning ausgenommen, da man zu Spielende seine Produktion von Nahrung und Material auf Punkte ändern möchte. An dem folgenden Beispiel erkennt man, wie viele Aktionen durch dieses Pruning herausgefiltert werden können.

In Abbildung 6.1 ist ein Spielstand in Runde 45 dargestellt, mit folgenden Werten:

- 27 bebaute Felder
- Davon 8 mit Punkt- oder Militärgebäuden bebaut

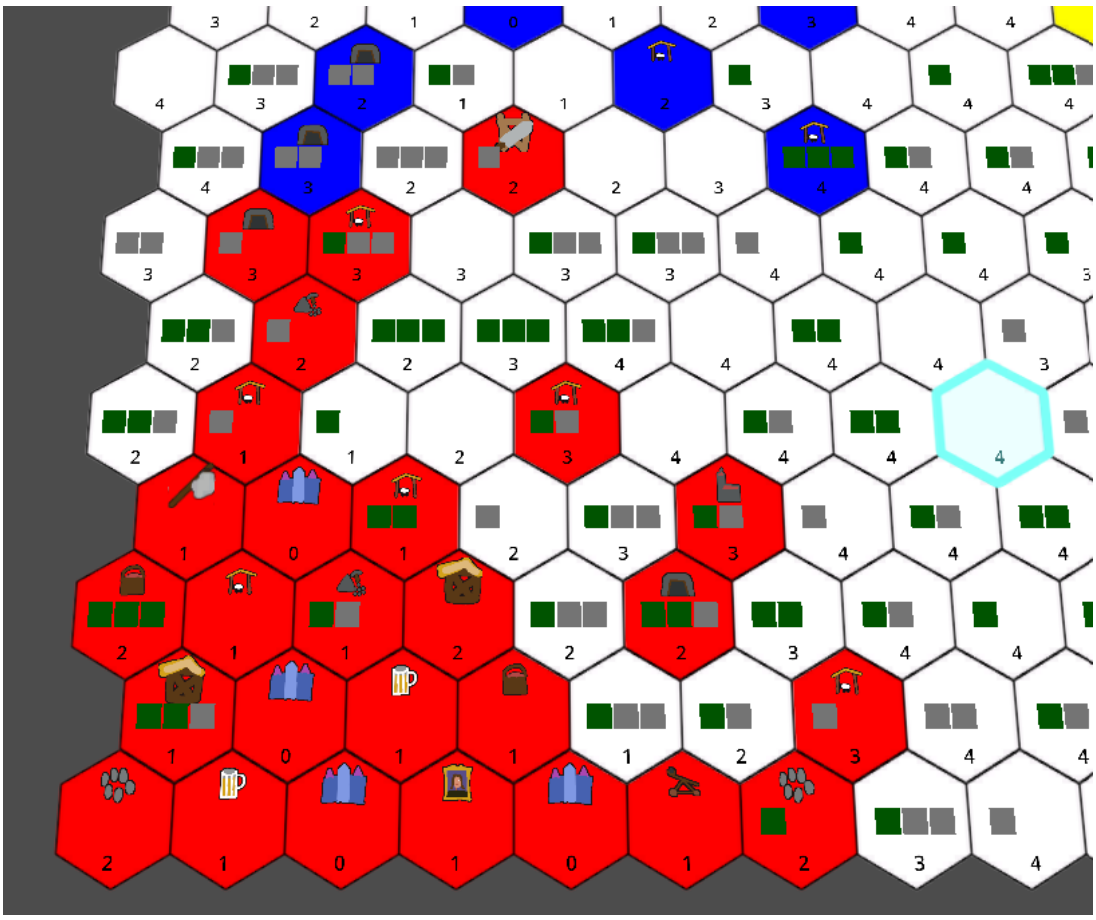


Abbildung 6.1: Quick Civ, Runde 45

- 22 Felder, die sich in vier Feldern Reichweite eines Stadtzentrums befinden und damit bebaut werden können
- Ausreichend Material und Nahrung, um alle diese Felder mit allen verschiedenen Gebäuden zu bebauen

Dadurch ergibt sich folgende Berechnung, um die Anzahl aller verfügbaren Aktionen **ohne Pruning** zu bestimmen:

$$A_{\text{ohne}} = (22 \cdot 28) + (27 \cdot 28) = 1372$$

Unter Berücksichtigung von **Pruning** (es existieren 12 Gebäude, die in die Kategorie Punkte-/Militärgebäude fallen) ergibt sich folgende Berechnung:

$$A_{\text{mit}} = (22 \cdot 28) + (8 \cdot 28) + (19 \cdot 12) = 1068$$

wobei:

- $22 \cdot 28$ die möglichen Aktionen für die 22 freien Felder darstellt,
- $8 \cdot 28$ die möglichen Aktionen für die 7 bereits bebauten Punkt-/Militärgebäude darstellt,

- $19 \cdot 12$ die möglichen Aktionen für die 20 anderen bebauten Felder mit den 12 Punkt-/Militärgebäuden darstellt.

Die Formel zur Berechnung der Anzahl der geprunten Aktionen lässt sich mit Gleichung 6.11 generalisieren.

$$A_{\text{geprunet}} = (F - F_{\text{militär}} - F_{\text{punkt}}) \cdot (G - G_{\text{militär}} - G_{\text{punkt}}) \tag{6.11}$$

wobei:

- $(F - F_{\text{militär}} - F_{\text{punkt}})$ die bereits bebauten Felder **ohne** Militär-bzw. Punktgebäude darstellt,
- $(G - G_{\text{militär}} - G_{\text{punkt}})$ die verfügbaren Gebäude **ohne** Militär-bzw. Punktgebäude darstellt

Im Verlauf des Spiels steigt die Anzahl der gepruneten Aktionen deutlich an, vor allem bei Strategien, die erst spät im Spielverlauf Punktegebäude priorisieren.

6.4.2 Greedy Nodes Only Pruning

[EN] Diese Pruning-Methode filtert von Anfang an alle Aktionen aus dem Action Space heraus, welche eine schlechtere Evaluierung des Spielzustands hervorrufen, als ohne die Aktion ausgeführt zu haben. Demnach werden nur Aktionen betrachtet, die die Evaluierungsfunktion steigern. Militärgebäude bilden eine Ausnahme für diese Methode, da diese keinen kurzfristigen Nutzen haben und damit gefiltert werden würden, was zu großem Nachteil führen würde. Durch diese Methode werden vor allem Aktionen, die niemals die beste Aktion sein können, herausgefiltert. Der Nachteil dieses Ansatzes ist die erhöhte Rechenzeit, da der Algorithmus, wie in Code 6.3 zu sehen, durch jede mögliche Aktion iterieren muss, um sie mit der aktuellen Evaluierungsfunktion zu vergleichen.

Code 6.3: OnlyGreedyActions

```
def filterOnlyGreedyActions(possibleActions, gameState):
  baseValue = getValue(gameState)
  result = []

  for action in possibleActions:
    if action.build.type == "military":
      result.Add(action)
      continue
    newGameState = getNewState(action, gameState)
    currentValue = getValue(newGameState)

    if currentValue > baseValue:
      result.Add(action)

  return result
```

6.4.3 Only Build Near Last Action Pruning

[ET] Diese Pruning Methode merkt sich immer die zuletzt ausgeführte Aktion und entfernt alle Aktionen aus dem Action Space, die sich nicht in einem Radius von 2 Feldern um diese Aktion befinden. Die Idee hierbei ist, den Aktionsraum drastisch einzuschränken und dem Algorithmus so immer nur einen kleinen Bereich zu geben, indem er arbeiten kann. Dadurch soll der Baum deutlich tiefer werden und komplexere Entscheidungen in diesem Bereich treffen und so eine optimiertere Gesamtlösung finden, welche aber erst nach und nach berechnet wird.

6.4.4 In Late No Early Buildings Pruning

[ET] Diese Pruning Methode sorgt dafür, dass, sobald einmal eins der teuersten Gebäude für 250 Material gebaut wurde, keine Gebäude für 10 Material mehr gebaut werden können. Der Gedanke ist hier, dass die frühen Gebäude sehr ineffizient sind und sobald man einmal ein 250-Material-Gebäude gebaut hat, man sich wahrscheinlich immer auch zumindest ein 50-Material-Gebäude leisten kann. Damit werden später im Spiel, wenn die Performance am kritischsten wird, ca. ein Drittel aller Aktionen herausgefiltert. Eine Ausnahme bildet das Stadtzentrum, welches trotzdem gebaut werden darf, da man es in jeder Spielsituation gebrauchen kann, um sich weiter auszubreiten.

6.5 Memory MCTS

[ET] Das Memory ist eine eigene Implementierung und wird in Kombination mit MCTS im folgenden Memory-MCTS genannt. Dabei fungiert es als eine weitere Pruning-Methode welche sich die Annahme zu nutze macht, dass Aktionen über mehrere Runden generell ähnliche Evaluierungswerte haben. Grundsätzlich bleiben schlechte Aktionen schlecht und gute Aktionen gut. Im Folgenden werden die Begriffe *LearningRate*, *TournamentSelectionSize*, *ResetStrength*, *ResetPoint* und *ExplorationParameter* verwendet. Dabei handelt es sich um Memory spezifische Parameter. Bei dem Memory-MCTS werden die Evaluierungen der simulierten Aktionen in einem Dictionary gespeichert. Während der Backpropagation werden die Werte dann nicht nur in den Knoten geschrieben, sondern überschreiben teilweise das Memory. Bei jeder neuen Backpropagation berechnet sich der Wert für das Memory mit Gleichung 6.12.

$$Memory[i] = Memory[i] * (1 - \alpha) + Value * \alpha \tag{6.12}$$

Hierbei ist α die *LearningRate*, die bestimmt, wie schnell der neue Wert in das Memory übernommen wird. In der *Selection* wird jetzt außerdem nicht mehr jede Aktion ausprobiert, sondern erst wird bestimmt, ob ein bestehendes Kind exploited werden soll oder ob neue Kinder explored werden sollen (mit einer Wahrscheinlichkeit von *ExplorationParameter*). Sobald neue Kinder explored werden sollen, werden aus den möglichen Aktionen zufällig eine *TournamentSelectionSize* große Gruppe an Aktionen ausgewählt und die Aktionen mit dem stärksten Memory-Wert wird dann genommen. Damit neue Aktionen überhaupt genommen werden, werden alle noch nicht evaluierten Aktionen mit

dem höchsten Wert von 1 initialisiert. Außerdem wird das Memory nach jeder Runde auch wieder ein wenig zurückgesetzt, damit ältere Aktionen, die zu Beginn eine schlechte Evaluierung bekommen haben, eine weitere Chance bekommen. Diese Rechnung ist in Gleichung 6.13 zu sehen.

$$Memory[i] = Memory[i] * (1 - ResetStrength) + ResetPoint * ResetStrength \quad (6.13)$$

Hierbei bestimmt der Parameter *ResetStrength*, nach wie vielen Runden der Memorywert wieder bei *ResetPoint* ist. *ResetPoint* sollte für den gewünschten Effekt gleich 1 sein, also dem maximal möglichen Evaluationswert. Hier ist er aber per Parameter anpassbar.

Nach ein paar Testläufen hat sich eine weitere Anpassung als sinnvoll herausgestellt. Am Ende jeder Runde werden sich alle Felder angeschaut, bei denen sich entweder der Besitzer oder das Gebäude geändert hat. Alle Aktionen auf diesen und den benachbarten Feldern werden wieder zum Ausgangswert von 1 zurückgesetzt. Da sich auf diesen Feldern gerade etwas geändert hat, sind die Evaluationen der Nachbarfelder mit hoher Wahrscheinlichkeit nicht mehr akkurat und werden daher zurückgesetzt. Dies hilft vor allem in Kriegsszenarien mit anderen Spielern.

7 Experimente und Ergebnisse

[EN] In dem folgenden Kapitel werden die Experimente und die dazugehörigen Ergebnisse abschnittsweise beschrieben. Zu Beginn werden durch eine HPO die Parameter des *VanillaMCTS* optimiert, sodass der resultierende *Bot* eine gute Spielqualität aufweist. Dieser *VanillaMCTS-Bot* dient als Benchmark, um in den anstehenden Experimenten die verschiedenen Pruning-Methoden und das Memory zu testen. Dazu spielt der gleiche *VanillaMCTS-Bot* jeweils **ohne** und **mit** der jeweiligen Pruning-Variante gegeneinander.

7.1 Hyper Parameter Optimization

[EN] Wie im Einführungstext beschrieben, dient diese Optimierung zur Erstellung eines *VanillaMCTSBots* mit einer guten Spielperformance. Der grundlegende Aufbau der HPO wurde bereits ausführlich in Abschnitt 6.3 beschrieben. Der Optimierungsalgorithmus wurde mit folgenden Parametern gestartet:

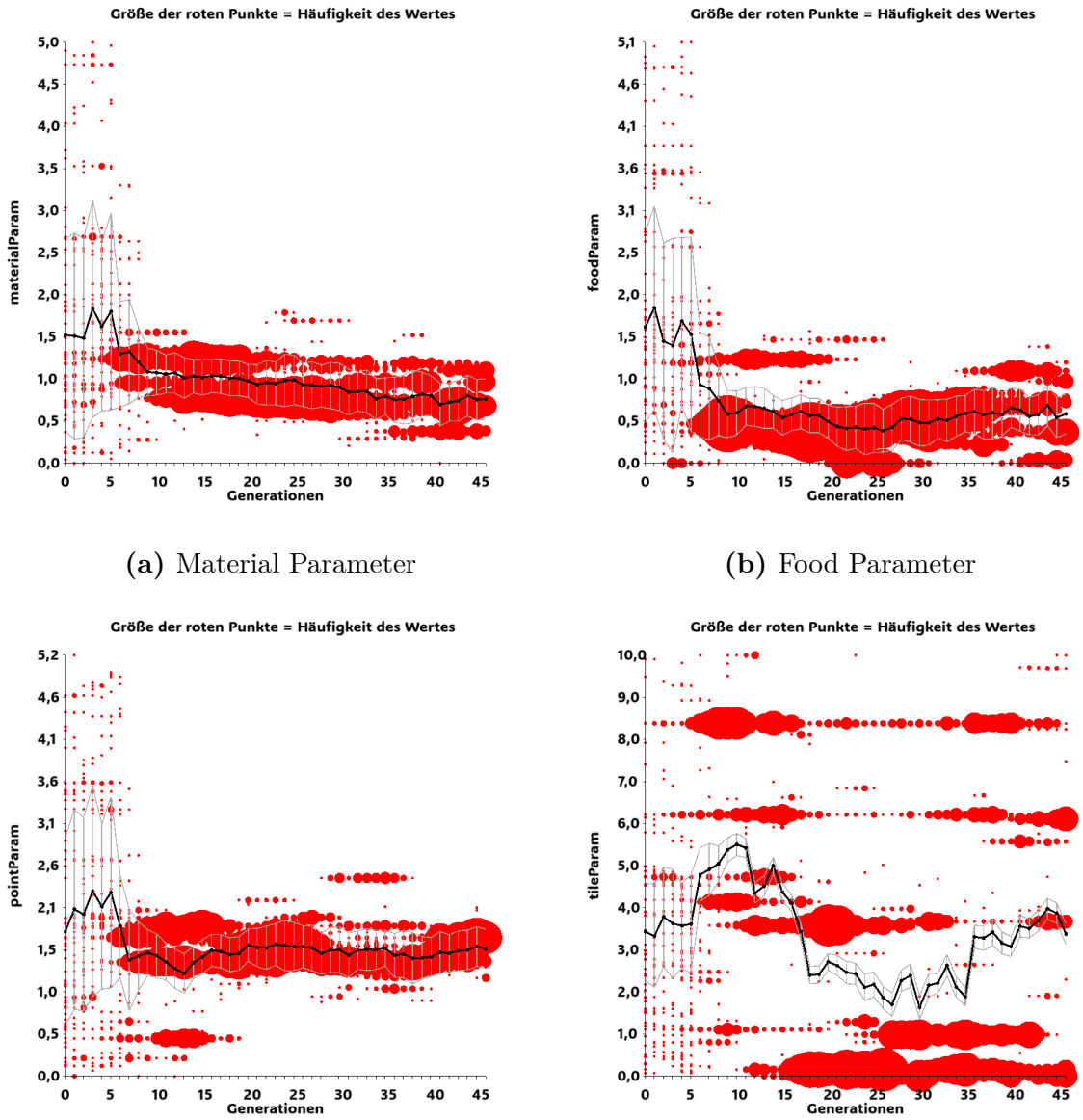
- Zeit pro Zug: 2 Sekunden
- Populationsgröße: 52
- Anzahl Generationen: 50
- Mutationsrate: 1
- Matches pro Individuum: 4
- Siegergröße: 50%, daraus folgt, dass die 26 besten Individuen unbehandelt in die nächste Generation übernommen werden und miteinander *Crossover* vollziehen, um die andere Hälfte der Population zu erstellen.

Wie in Abschnitt 6.3 beschrieben, startet die Population zur Hälfte mit *Greedy-Bots* und *MCTS-Bots*. Bereits nach acht Generationen haben sich die *MCTS-Bots* durchgesetzt und es existiert kein *Greedy-Bot* mehr in der Population.

7.1.1 Parameterergebnisse

[EN] Um den Optimierungsalgorithmus auswerten zu können, wird eine Visualisierung verwendet. Dadurch lassen sich alle Parameter, die optimiert werden, im Verlauf der Generationen in einem Diagramm darstellen. Jeder rote Punkt auf den Diagrammen steht für ein Individuum, welches diesen Parameterwert in dieser Generation besitzt. Je größer der rote Punkt, desto mehr Individuen haben exakt den gleichen Parameterwert. Die schwarze Linie stellt den Durchschnitt der Generation dar. Durch die Diagramme kann erkannt

werden, welche Parameter irrelevant sind, bzw. welchen Wert sie für optimale Performance annehmen sollten.



(a) Material Parameter

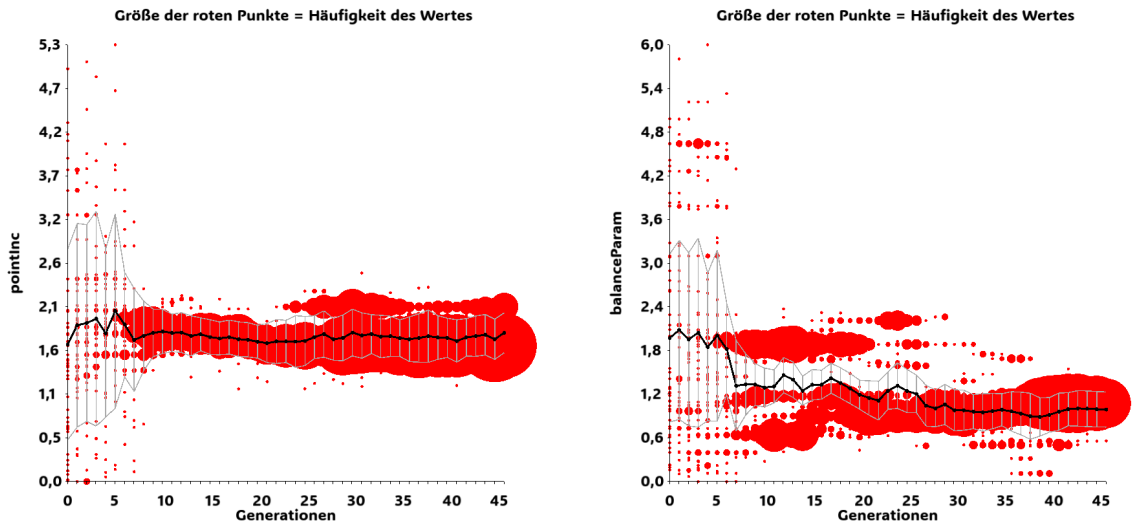
(b) Food Parameter

(c) Point Parameter

(d) Tile Parameter

Abbildung 7.1: Parameter: Evaluierungsfunktion 1

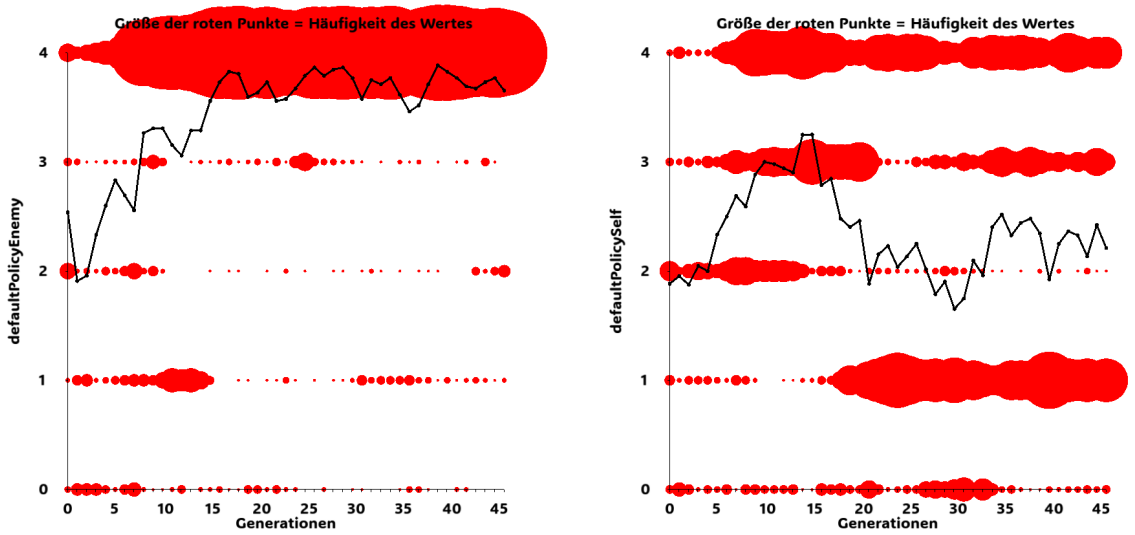
In Abbildung 7.1 sind die Basisparameter für die Evaluierungsfunktion (siehe Gleichung 6.2) dargestellt. Anhand der Visualisierung wird deutlich, dass sich sowohl *Material-* und *Food Parameter* bei einem Wert zwischen $0.5 - 1.0$ einpendelt. Der *Point Parameter* konvergiert nach den 50 Generationen zu einem Wert von circa $1.5 - 1.7$. Lediglich der *Tile Parameter* weist eine große Streuung auf und lässt sich deswegen nicht trivial auf einen optimalen Wert führen.



(a) PointIncrement Parameter **(b)** Balance Parameter

Abbildung 7.2: Parameter: Evaluierungsfunktion 2

In Abbildung 7.2 sind die beiden Parameter *Point Increment* und *Balance* für die Evaluierungsfunktion (siehe Gleichung 6.2) dargestellt. Der *Balance Parameter* hat nach einigen Generationen einen Wert von circa 1. Der *Point Increment Parameter* ist der entscheidendste Wert und konvergiert deswegen am schnellsten. Nach nur wenigen Generationen hat er stetig einen Wert um $1.6 - 1.7$. Er ist dafür verantwortlich, ab welcher Runde die Bots ausschließlich Punktegebäude bauen. Ein Wert von $1.6 - 1.7$ entspricht dabei einer Spielweise, bei der bis Runde 60 alle Gebäude gebaut werden und anschließend lediglich Punktegebäude favorisiert werden.



(a) Default Policy Enemy **(b)** Default Policy Self

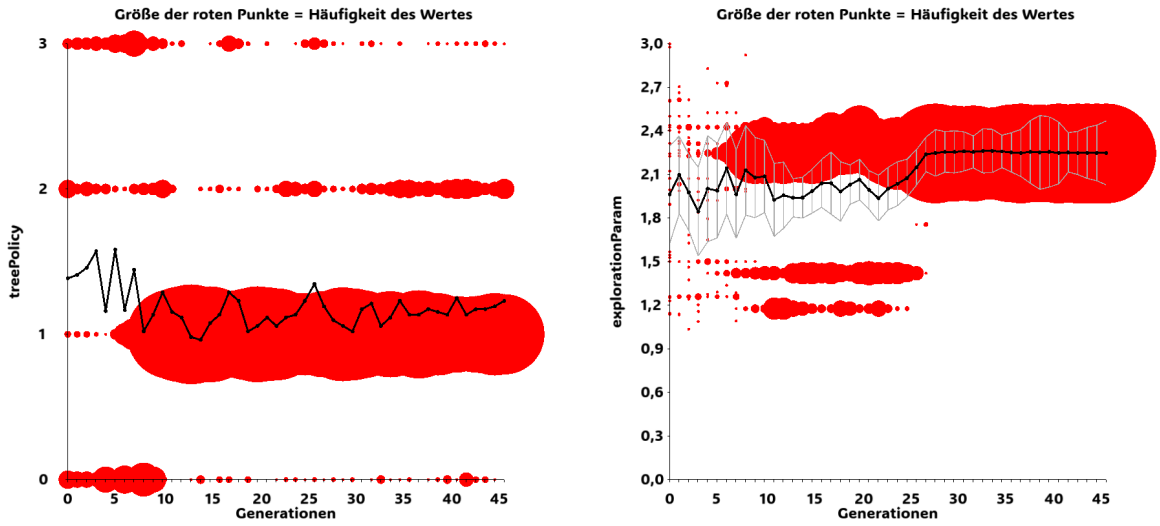
Abbildung 7.3: Parameter: Default Policy

In Abbildung 7.3 sind die *Default Policies* der *MCTS-Bots* dargestellt. Diese Policies sind verantwortlich dafür, welche Strategie entweder man selbst oder die Gegner während der

Simulationsphase durchführen. Dabei stehen die Zahlwerte für folgende Strategien. Die Strategien wurden bereits in Abschnitt 6.2 ausführlich beschrieben.

- 0: *Random*
- 1: *Greedy*
- 2: *RandomWithoutOverwrite*
- 3: *RandomWithourOverwriteWithMilitary*
- 4: *JustWaiting*

Dabei muss beachtet werden, dass sobald eine der beiden Policies *JustWaiting* ist, **alle** Spieler in den Simulationen nur warten. Demnach ist dann die andere Policy, welche **nicht** *JustWaiting* ist, irrelevant. Dieses Verhalten kann auch in den Diagrammen erkannt werden. Eine der zwei Policies konvergiert ziemlich schnell gegen *JustWaiting*, was dazu führt, dass die andere Policy irrelevant wird.



(a) Tree Policy

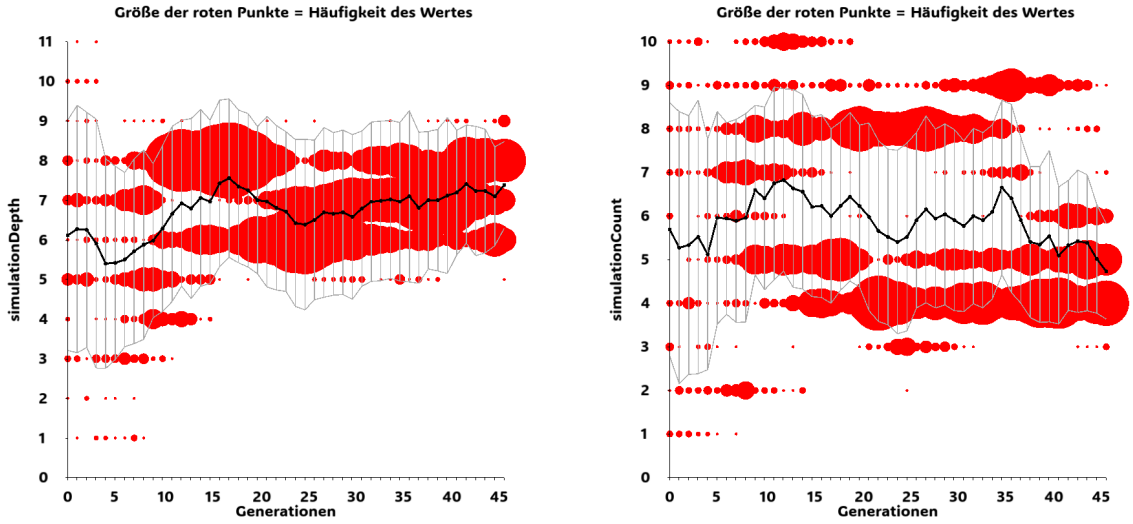
(b) Exploration Parameter

Abbildung 7.4: Parameter: Tree Policy

In Abbildung 7.4 wird die Auswahl der *Tree Policy* dargestellt. Diese Policy ist verantwortlich dafür, welcher Knoten in der Selektion als nächstes *exploritert* oder *exploited* wird. Dabei stehen die Zahlwerte für folgende Selektionsstrategien. Die Strategien wurden bereits in Abschnitt 6.2 ausführlich beschrieben.

- 0: *Random*
- 1: *Greedy*
- 2: *EpsilonGreedy*
- 3: *UCB*

In dem Diagramm ist klar erkennbar, dass die *Greedy*-Strategie die dominante Selektionsstrategie ist. In (b) ist der *Exploration*-Parameter für die **ucs!** (**ucs!**)-Strategie abgebildet. Anfangs wird deutlich, dass der Parameter zwischen den Werten $1 - 3$ pendelt. Dadurch, dass sich die *Greedy*-Strategie durchsetzt, bleibt der Parameter jedoch ziemlich schnell auf einem konstanten Niveau, da er nicht mutiert, wenn er nicht benutzt wird.



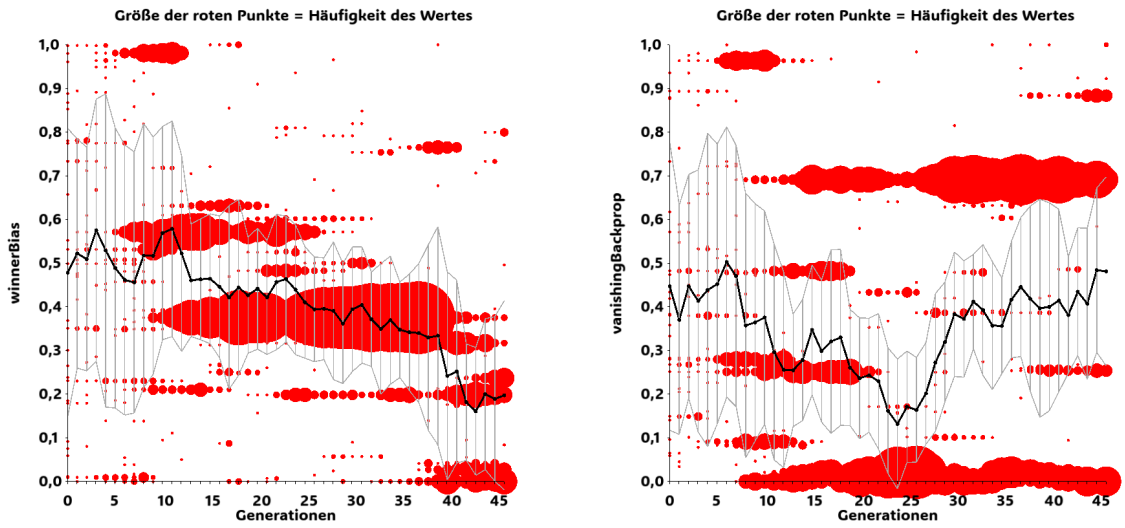
(a) Simulationstiefe **(b)** Simulationsanzahl

Abbildung 7.5: Parameter: Simulationen

In Abbildung 7.5 werden die Anzahl und die Tiefe der Simulationen visualisiert. Wohingegen die Simulationstiefe gegen einen Wert von zwischen $6 - 8$ konvergiert, ist die Simulationsanzahl sehr gestreut. Dies hat den Grund, dass sich die Default Policy *Just-Waiting* in Abbildung 7.3 durchgesetzt hat und bei dieser Strategie die Simulationsanzahl nicht relevant ist, da es keinen Grund gäbe, dieselbe Simulation mehrfach identisch abzuwarten.

In Abbildung 7.6 sind die letzten beiden entscheidenden Parameter für den MCTS-Algorithmus dargestellt. Der *Winner-Bias*-Parameter nähert sich einem Wert um circa 0.2 an, was bedeutet, dass alle Gegner circa gleichstark in die Gewichtung der Evaluierungsfunktion eingehen. Der *Vanishing-Backpropagation*-Parameter hat eine deutlich größere Streuung und kann nicht sofort auf einen eindeutigen Wert zurückgeführt werden.

Nach 50 Generationen wurde der Optimierungsalgorithmus terminiert. Der Bot, welcher für die kommenden Ergebnisse verwendet wird, wurde aus der letzten Generation extrahiert. Dieser Bot überdauerte 6 Generationen und hat in den dabei 24 Matches 6 Siege geholt und im Durchschnitt ein Ergebnis von 90% des besten Spielers des Matches erzielt. Der Bot hat eine Parameterkonstellation, die passend zu den Visualisierungen ist, da jeder Parameter in dem Bereich der Werte liegt, gegen den die Graphen konvergieren. Folgende Parameter bilden den Bot, ab jetzt *VanillaMCTSBot* der für die weiteren Ergebnisse verwendet wird (es werden nur Parameter aufgeführt, die auch benutzt werden):



(a) Winnerbias Parameter **(b)** Vanishing Backpropagation Parameter

Abbildung 7.6: Parameter: Simulationen

- **Simulationstiefe**: 6
- **Simulationsanzahl**: 4
- **Vanishing-Backpropagation**: 0.69
- **Winner-Bias**: 0.20
- **Tree-Policy**: *Greedy*
- **Simulationsstrategie**: *JustWaiting*
- **Material-Param**: 0.68
- **Food-Param**: 0.73
- **Point-Param**: 1.70
- **Tile-Param**: 6.11
- **Balance-Param**: 0.87
- **PointInc-Param**: 1.86

7.1.2 Auswertung

[EN] Nachdem im vorherigen Abschnitt der Benchmark-Bot mithilfe einer umfangreichen HPO bestimmt wurde, soll in diesem Abschnitt analysiert werden, weshalb sich gerade diese Konstellation von Parametern durchgesetzt hat. Auffällig ist, dass sich während des Optimierungsprozesses bestimmte Parameter relativ schnell stabilisiert haben, während andere größere Streuung zeigten oder sich als weniger entscheidungsrelevant herausstellten. Zuerst werden die Parameter genau erklärt, die eine schnelle Konvergenz zu einem

Wert vorweisen. Anschließend wird erläutert, wieso die restlichen Parameter eine große Streuung bieten.

Insbesondere Parameter wie die Tree-Policy **Greedy** und die Simulationsstrategie **JustWaiting** haben maßgeblich zum Erfolg des Bots beigetragen, da sie einen effizienten und deterministischen Suchprozess ermöglichen. Sie haben zwar den Nachteil, weniger aussagekräftige Simulationen zu erzeugen, da durch **JustWaiting** lediglich in den Simulationen gewartet wird, jedoch werden im gleichen Atemzug dadurch deutlich mehr Knoten erreichbar, da die Simulationen weniger Rechenzeit beanspruchen. Damit diese Simulationsstrategie jedoch wirklich effektiv ist, benötigt sie eine Simulationstiefe von mindestens drei, da nach drei Runden die militärischen Konflikte aufgelöst werden. Deswegen steigt der Parameter **Simulationstiefe** schnell an und bleibt dann stets konstant über drei. Die Tree-Policy **Greedy** hat sich als dominante Strategie abgezeichnet, entgegen der Erwartungen, dass UCB die stärkste sein wird. Die Erwartung entstand durch die aufstrebende Beliebtheit des UCB in der Vergangenheit, wie in Kapitel 4 beschrieben. Die Dominanz der Greedy-Tree-Policy lässt sich auf zwei Gründe zurückführen. Zum Einen benötigt sie weniger Rechenzeit als die relativ aufwendige Formel von UCB. Zum Anderen stellt diese Art der Selektionsstrategie sicher, dass weniger vielversprechende Knoten weniger Rechenzeit beanspruchen, da sie nicht exploriert werden. So können die zur Verfügung stehenden Iterationen des MCTS-Algorithmus auf die Knoten fokussiert werden, die eine höhere Bewertung mit sich bringen. Logischerweise wird auf diese Weise eher ein lokales Maximum getroffen, da weniger exploriert wird. Dies ist aber weniger von Bedeutung, da das globale Maximum realistisch gesehen unmöglich zu finden ist.

Die rasche Konvergenz des PointIncrement-Parameters lässt sich darauf zurückführen, dass es für den Bot für den Spielsieg von großer Bedeutung ist, ab wann er Punktgebäude favorisiert. Wie in Kapitel 7 beschrieben, nähert sich der Parameter schnell einem Wert von circa 1.6 an. Dies hat zur Folge, dass die Bots ab Runde 60 ausschließlich Punktgebäude bauen. Dieser Zeitpunkt scheint ein sehr guter TradeOff zwischen Konstruktion einer starken Wirtschaft und früher Punktegenerierung zu sein. Wenn der Parameter größer wäre, so würden die Bots zu spät mit der Punktegenerierung starten und könnten so den bereits erzielten Vorsprung anderer Spieler nicht mehr aufholen. Wenn der Parameter kleiner wäre, so hätten die Bots keine starke Produktion, um langfristig eine starke Punktegenerierung zu garantieren und sich gegen Feinde zu verteidigen.

Der Balance-Parameter konvergiert ebenfalls schnell zu einem festen Wert von circa 1. Dies bedeutet, dass *Food-* und *Materialgebäude* gleichstark bevorzugt werden. Er beeinflusst die Formel Gleichung 6.2 maßgeblich, sodass jene Spielweise belohnt wird, bei der die Produktion beider Ressourcen ähnlich hoch ist. Ein Wert von 1 zeigt demnach auch, dass beide Ressourcen ähnlich wichtig für den Spielerfolg sind.

Die Parameter für die Evaluierungsfunktion in Abbildung 7.1 nähern sich ebenfalls schnell einem ähnlichen Wert an, was darauf zurückzuführen ist, dass keine der Ressourcen stärker gewichtet werden soll. Lediglich der **Tile**-Parameter sticht heraus. Die große Streuung hat eine einfache Erklärung. In der Gleichung 6.2 ist der Tile-Parameter ein Faktor, jedoch ist der Faktor, mit dem der Parameter multipliziert wird, die Anzahl an bebauten Feldern. Dieser Faktor ist deutlich kleiner, als die Faktoren der anderen Produkte, wie zum Beispiel Nahrungsproduktion oder Materialproduktion. Demnach ist das Tile-Produkt immer deutlich kleiner als die anderen Produkte und wird nur hat daher auch nur einen relativ kleinen Einfluss.

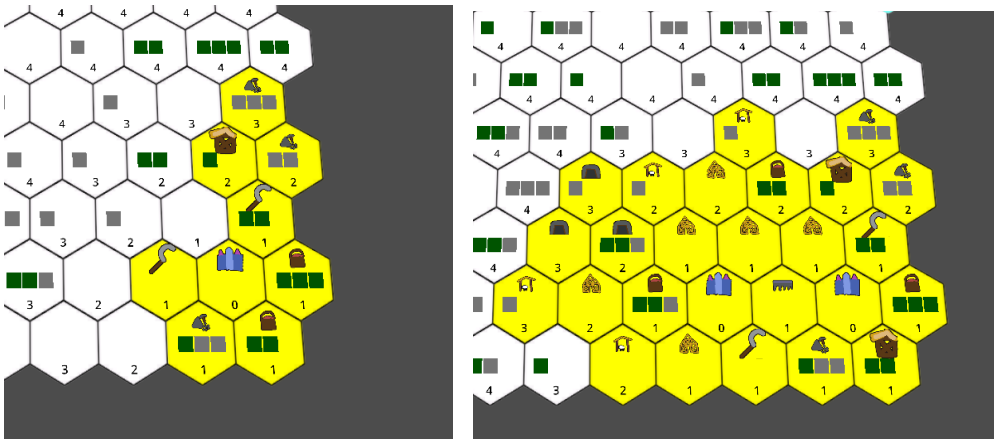
Wie bereits zuvor erwähnt sind die Parameter, die bis hierhin noch nicht genauer beschrieben wurden, weniger relevant. Die **Simulationsanzahl** ist nur von Bedeutung, wenn in der Simulationsstrategie auch richtige Aktionen simuliert werden, anders als bei dem *VanillaMCTSBot*, der in den Simulationen nur wartet. Man erkennt in den ersten Generationen einen starken Anstieg der Simulationsanzahl, da andere Simulationsstrategien von einer höheren Simulationsanzahl profitieren. Jedoch nach einer bestimmten Zeit, steigt sie nicht weiter sondern streut mit hoher Varianz. Dies liegt daran, dass sie ab dem Zeitpunkt, wo so gut wie alle Bots die *JustWaiting*-Simulationen benutzen, irrelevant ist.

Der **VanishingBackprop**-Parameter, der wie in Abschnitt 6.2 beschrieben für die Verringerung des Einflusses tieferer Knoten verantwortlich ist streut sehr stark und scheint deswegen wenig relevant zu sein. Um die Visualisierung jedoch in den richtigen Kontext zu rücken, muss erwähnt werden, dass wenn der Parameter einen Wert von 0 hat, er auf 0.1 erhöht wird, da die Formeln für die Backpropagation sonst nicht anwendbar sind.

WinnerBias ist der letzte Parameter und erfordert nur eine kurze Analyse. Der Wert weist eine Streuung auf, jedoch im unteren Bereich $[0.0, 0.5]$. Dies hat zur Folge, dass der Bot nicht den stärksten Spieler fokussiert, sondern alle Gegner ziemlich gleichmäßig. Der Grund dafür ist simpel. Der verwendete EA benutzt als Fitnessfunktion Summe aller Matches und der daraus gewonnen Punkte im Vergleich zum besten Spieler diesen Matches - nicht also die Spielsiege selbst. Demnach sind riskante Spielweisen, die den Führenden fokussieren, nicht so erfolgreich. Dafür ist es wichtiger, im Durchschnitt eine sehr hohe Punktzahl zu erreichen, um nah an den besten Spieler heranzukommen.

7.1.3 Spielweise

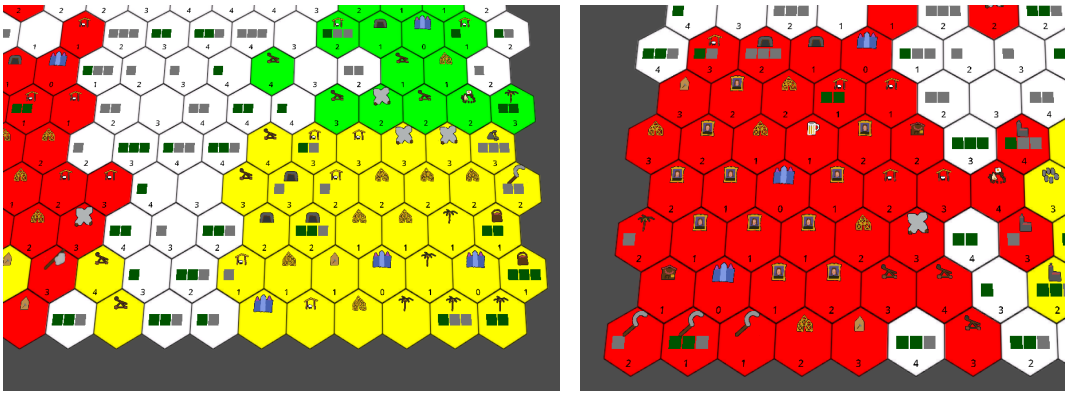
[EN] Um dem Leser eine bildliche Vorstellung zu geben, wie sich der *VanillaMCTSBot* in dem Spiel *Quick Civ* verhält, folgt nun eine Visualisierung eines typischen Spiels von diesem Bot. In den kommenden Experimenten wird außerdem statistisch dargestellt, welche Strategien er verfolgt.



(a) Anfangsgebäude **(b)** Produktionsverbesserung

Abbildung 7.7: Spielstart

In der Abbildung 7.7 erkennt man, wie der *VanillaMCTSBot* anfangs günstige Gebäude baut, um seine Nahrungs- und Materialproduktion gleichmäßig (aufgrund ausgewogener *Food-* bzw. *Material-Parameter* zu erhöhen. Durch eine größere Materialmenge kann er dann im Laufe des Anfangsspiels stärkere Gebäude bauen, um seine Produktion weiter zu maximieren.



(a) Militärischer Konflikt **(b)** Punktegenerierung

Abbildung 7.8: Midgame

In der Abbildung 7.8 wird deutlich, wie der *VanillaMCTSBot* reagiert, wenn er auf Feinde trifft. Aufgrund eines erhöhten *Tile-Parameter* bevorzugt er die aggressive Spielweise, um neue Felder zu erobern. Da der *Winner Bias* ziemlich gering ist, konfrontiert der Bot alle Gegner gleichstark und fokussiert nicht nur den bestplatzierten. Der *PointInc-Parameter* von 1.70 hat zur Folge, dass ab Runde 60 lediglich Punktegebäude gebaut werden bzw. erobert werden möchten. Dieses Verhalten ist bei dem roten Spieler klar erkennbar, da er

in seinem Reich einige *Museen* platziert, die für starke Punktegenerierung verantwortlich sind.

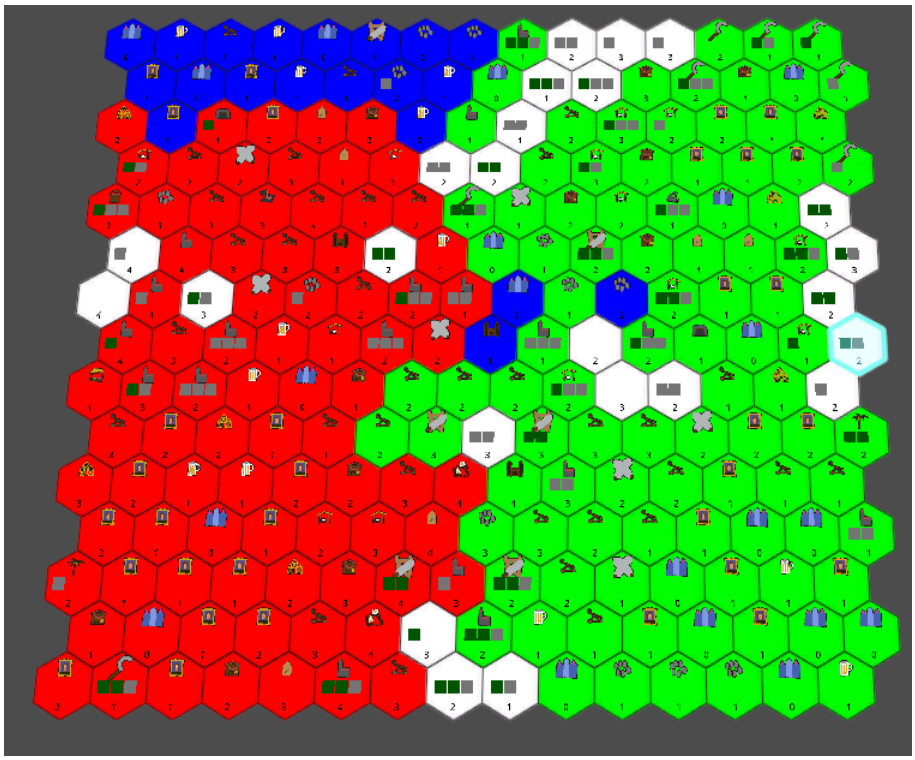
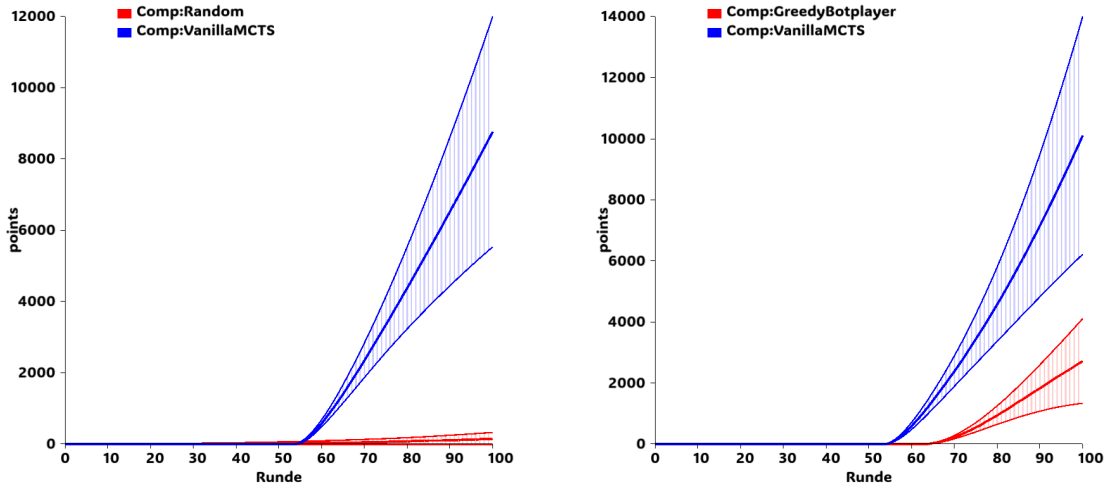


Abbildung 7.9: Spielende

Abbildung 7.9 zeigt das Spielende eines typischen Spiels des *VanillaMCTSBots*. Im Spielverlauf entstehen durch militärische Konfrontationen einige Unterschiede in der Größe des Reiches. In diesem Beispiel wurde ein Spieler sogar komplett eingenommen. An der Abbildung erkennt man außerdem, dass zu Spielende fast ausschließlich Punkte- und Militärgebäude existieren.

7.2 VanillaMCTSBot versus Benchmarks

[EN] Bevor die verschiedenen Pruningmethoden gegen den *VanillaMCTSBot* getestet werden, soll zunächst anhand von Abbildung 7.10 verdeutlicht werden, dass die in Kapitel 6 beschriebenen Benchmark-Bots *Random* und *Greedy* dem *VanillaMCTSBot* klar unterlegen sind. Die Abbildung zeigt deutlich, dass sie gegenüber der überlegenen Entscheidungsstrategie des *VanillaMCTSBots* keine ernstzunehmende Konkurrenz darstellen.



(a) VanillaMCTSBot vs. Random **(b)** VanillaMCTSBot vs. Greedy

Abbildung 7.10: VanillaMCTSBot vs. Benchmarks

7.3 Qualität des MCTS

[EN] Im Rahmen der Bachelorarbeit stellt sich die Forschungsfrage, inwiefern der MCTS-Algorithmus anwendbar für 4X-Spiele ist und welche Parameter besonders wichtig für eine gute Performanz sind. Zusammenfassend lässt sich sagen, dass die Qualität des MCTS-Algorithmus von verschiedenen Faktoren abhängt. Der bedeutendste Einflussfaktor ist die benötigte Rechenzeit für die Erstellung neuer Spielstände. Wie bereits ausführlich in Abschnitt 5.3 beschrieben, existieren Spiele bei denen es möglich ist, mehrere Millionen Spielstände in einer Sekunde zu berechnen. 4X-Spiele haben den Nachteil, oft sehr komplex zu sein und deswegen eine Menge Rechenzeit für die Erstellung neuer Spielstände zu beanspruchen. Aus diesem Unterschied folgt, dass für den MCTS-Algorithmus deutlich weniger Iterationen möglich sind, die zu einer geringeren Wahrscheinlichkeit führen, eine optimale Lösung zu finden. Aus diesem Grund ist die Verwendung des MCTS-Ansatzes für 4X-Spiele nur sinnvoll anwendbar, falls das Spiel über eine gute Balance zwischen Rechenzeit und benötigter Qualität der gefundenen Lösung aufweist. Das Spiel *QuickCiv* ist ein gutes Beispiel dafür, denn die Berechnung mehrerer 1.000 Spielstände ist möglich, ohne die Anforderung zu haben, dass der Algorithmus den besten Spielzug finden muss, da es dafür schlichtweg zu viele Aktionen gibt. Ein zweiter entscheidender Faktor für

die Anwendbarkeit des Algorithmus ist die richtige Parameterkonstellation. Der MCTS-Algorithmus lässt sich durch eine große Menge an Parametern verändern, jeweils mit einer starken Auswirkung auf die Qualität des Algorithmus. Deshalb empfiehlt sich, mehrere Tests durchzuführen, um die stärkste Parameterkonstellation herauszufinden. Dabei wurde in dieser Bachelorarbeit eine HPO mit großem Erfolg verwendet, da sich unerwartete Strategien als die beste Option heraus gestellt haben. So sind (*JustWaiting*-Simulationen stärker als *Random*-Simulationen). Der Algorithmus hat jedoch eine große Schwäche. Dadurch, dass er in 4X-Spielen keine Simulationen bis zum Spielende simulieren kann, fehlt ihm das langfristige Planen. Er ist gut darin, in der nahen Zukunft clevere Kombinationen zu finden, aber wenn es um langfristiges Planen geht, hat er einen großen Nachteil im Vergleich zu menschlichen Spielern. Auf diese Weise ist es einem Menschem möglich, durch langfristige Taktiken, wie das frühe Besiedeln des Territoriums der Gegner, mit Leichtigkeit gegen den Bot zu gewinnen.

Insgesamt lässt sich die Forschungsfrage also so beantworten, dass der MCTS-Algorithmus für ein 4X-Spiel anwendbar ist, wenn er zum einen die nötige Performanz aufweist, das nötige *Forward-Modell* besitzt und keine Bedingung existiert, den optimalen Spielzug zu finden. Für Spiele wie zum Beispiel Civilization [Gam23] ist der Algorithmus ungeeignet, da die Berechnung neuer Spielstände aufgrund der hohen Spielkomplexität zu lange dauert. Für kleinere Spiele, bei denen die Spielkomplexität überschaubar ist, lässt sich der Algorithmus gut verwenden, bedingt aber eine starke Priorisierung der Spieloptimierung in der Entwicklung, sowie das Kalkulieren einer adäquaten Parameterkonstellation. Lässt sich der Algorithmus auf das Spiel anwenden, so hat man den klaren Vorteil, dass man eine KI entwickelt hat, die keine spezifischen Anpassungen braucht, wenn sich das Spiel weiterentwickelt. Dies ist ein enormer Vorteil im Vergleich zu herkömmlichen KI-Methoden, wie *Decision-Trees*. Für das in dieser Bachelorarbeit verwendete Spiel *QuickCiv* lässt sich also klar sagen, dass der MCTS-Algorithmus gut anwendbar ist, wenn man die nötigen Vorkehrung wie eine gute Spieloptimierung und das Austesten der besten Parameterkonstellationen vollzieht.

7.4 Action Space Pruning und Memory MCTS

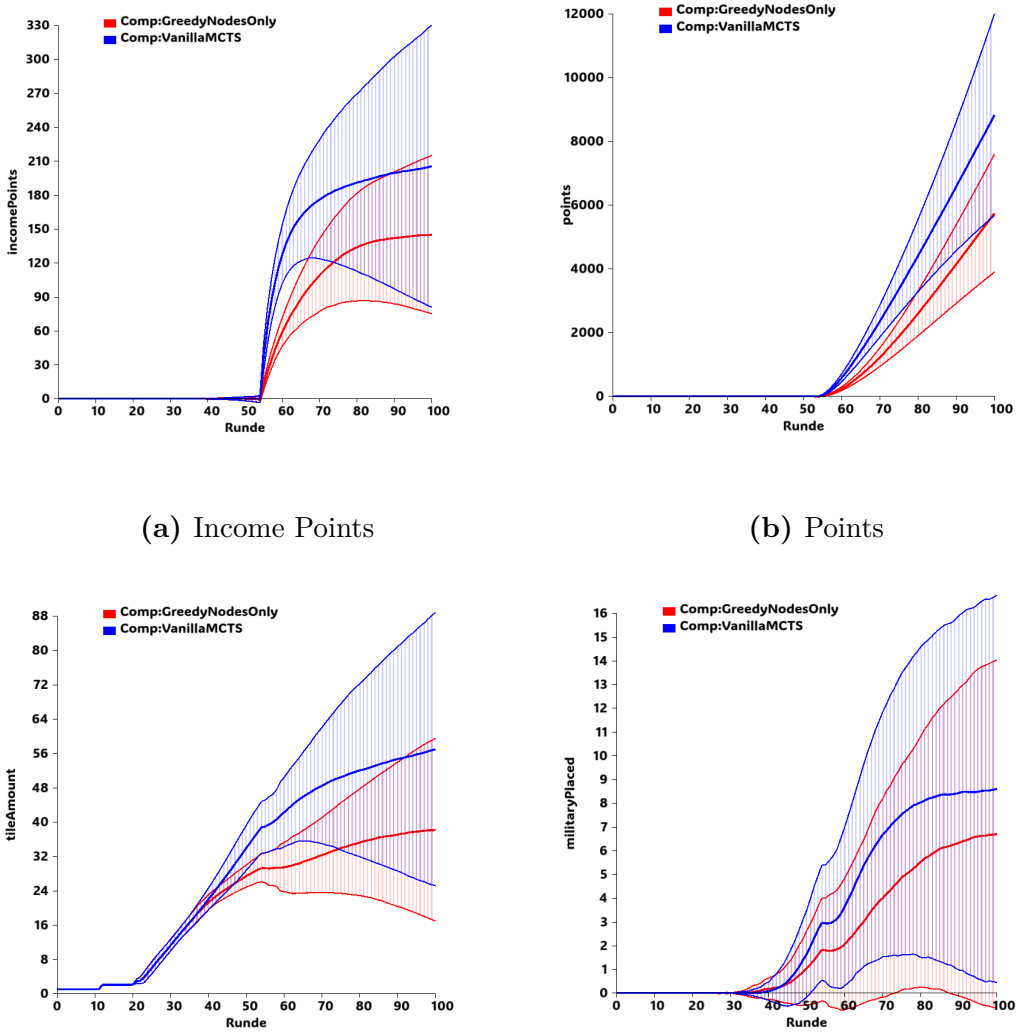
[ET]

Um die verschieden Verbesserungsmethoden gut vergleichen zu können, spielen in diesem Teil der Experimente immer 4 Bots gegeneinander: 2 mal der *VanillaMCTSBot* und 2 mal der *VanillaMCTSBot* mit einem Pruning oder Memory. Dieses Spiel wird 500 mal wiederholt und die Performanz der Bots in verschiedenen Metriken geplottet. Die Metriken die uns hier am meisten beschäftigen sind Points (wie viele Punkte die Bots in jeder Runde haben), IncomePoints (wie viele Punkte die Bots pro Runnde dazu verdienen), TileAmount (wie viele Spielfelder die Bots jede Runde haben) und MilitaryPlaced (wie viele Militärgebäude die Bots zu jedem Zeitpunkt haben). Da jeweils zwei Bots ein einem Spiel identisch sind werden sie in den Grafiken für eine bessere Übersicht zusammengefasst.

Um Die statistische Signifikanz der Experimente zu bestimmen wurde danach ein T-Test angewendet [Wika]. Dabei wurden für jede Optimierungsmethode jeweils die Menge der gesammelten Punkte am Ende des Spiels mit den Punkten der unoptimierten Variante verglichen.

7.4.1 Greedy Nodes Only Pruning

[ET]



(a) Income Points

(b) Points

(c) Tile Amount

(d) Military Placed

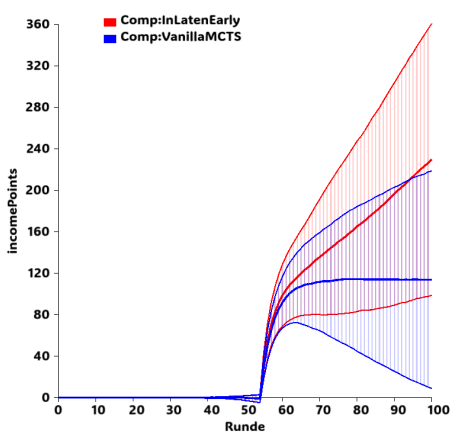
Abbildung 7.11: Greedy Nodes Only Pruning

Bei dem T-Test ist der t-Wert -26,64 mit einem Freiheitsgrad von 1612,04 und damit deutlich unter dem kritischen Wert von -1,96. Damit ist das Ergebnis eindeutig statistisch signifikant.

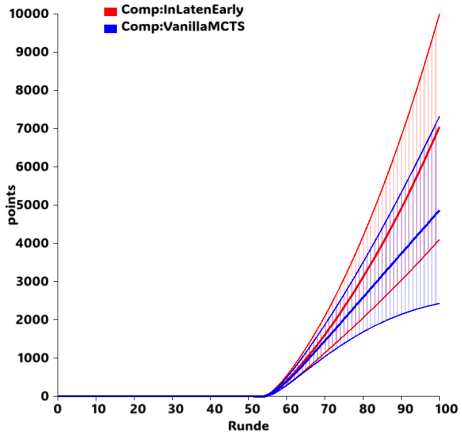
Alle Metriken sind spätestens in Runde 60 deutlich schlechter als bei dem Benchmark Bot wie man in Abbildung 7.11 sehen kann. Das heißt, das *Greedy Nodes Only Pruning* hat die Performance des MCTS stark verschlechtert, obwohl das Pruning nur gute Aktionen übrig lassen sollte. Allerdings wurde nach weiterer Analyse klar, dass die Berechnung zu einem großen Performanz Verlust führt. So konnte der Bot mit Pruning ca. 100 mal weniger Iterationen durchführen als der *VanillaMCTSBot*, in der selben Zeit. Dieser Performanzverlust kann durch das Pruning nicht ausgeglichen werden und der Algorithmus macht so insgesamt zu wenig Iterationen, um ein starkes Ergebnis zu erzielen

7.4.2 In Late No Early Building Pruning

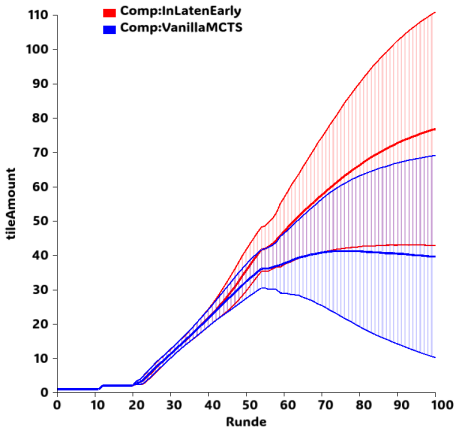
[ET]



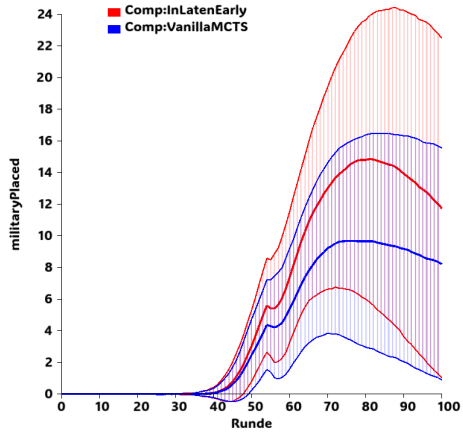
(a) Income Points



(b) Points



(c) Tile Amount



(d) Military Placed

Abbildung 7.12: In Late No Early Building Pruning

Bei dem T-Test ist der t-Wert 17,98 mit einem Freiheitsgrad von 1931,81 und damit deutlich über dem kritischen Wert von 1,96. Damit ist das Ergebnis eindeutig statistisch signifikant.

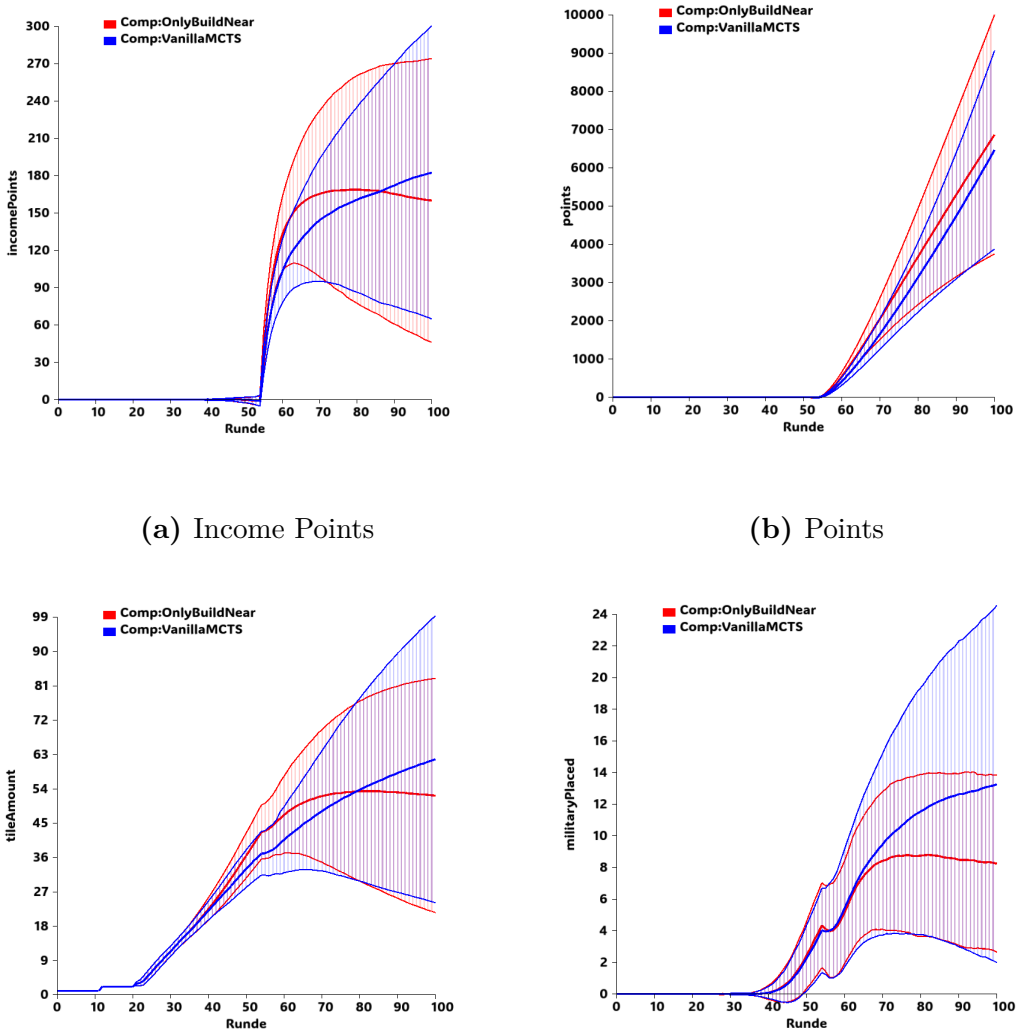
Das *In Late No Early Building Pruning* gibt einen sehr starken Vorteil in allen Metriken, vor allem je länger das Spiel andauert. So steigen die durchschnittlichen Income-Points am Ende des Spiels auf ca. das Doppelte des *VanillaMCTSBot*. Dadurch beendet der Pruning Bot das Spiel im Durchschnitt mit 40% mehr Punkten. Außerdem kann man beobachten, dass beide Bots, aber vor allem der geprunte, gegen Ende der Partie wieder weniger Militärgebäude besaßen als zum Maximum bei Runde 80 (siehe Abbildung 7.12).

Das *In Late No Early Building Pruning* liefert mit Abstand den größten Vorteil unter allen Pruningverfahren. Dass das Puning so effektiv ist, liegt daran, dass es ca. ab Runde 45 gut

ein Drittel aller Aktionen entfernen kann, ohne viel Performance zu kosten. Dabei sind die geprunten Aktionen auch fast immer schlecht, da die Startgebäude im späteren Spielverlauf zu wenig für den Platz bringen, den sie verbrauchen.

7.4.3 Only Build Near Last Action Pruning

[ET]



(a) Income Points

(b) Points

(c) Tile Amount

(d) Military Placed

Abbildung 7.13: Only Build Near Last Pruning

Bei dem T-Test ist der t-Wert 3,15 mit einem Freiheitsgrad von 1932,11 und damit deutlich über dem kritischen Wert von 1,96. Damit ist das Ergebnis eindeutig statistisch signifikant.

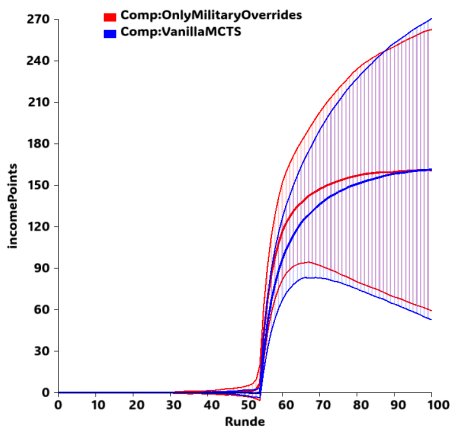
Grundsätzlich ist das *Only Build Near Last Action Pruning* besser als der Benchmark Bot, wie man an den Gesamtpunkten in Abbildung 7.13 erkennen kann. Aber *IncomePoints* des Pruning Bots steigen bis Runde 60 deutlich stärker und bleiben bis Runde 85 auch höher als beim *VanillaMCTSBot*, werden danach aber von ihm überholt. Der so erspielte Vorteil reicht dann, um insgesamt gegen den Benchmark Bot zu gewinnen. Aus den Graphen zu *TileAmount* und *MilitaryPlaced* geht auch hervor woran das liegt. Ab Runde 70 nimmt sowohl *MilitaryPlaced* als auch *TileAmount* nicht mehr zu.

Das *Only Build Near Last Action Pruning* verstärkt und verschlechtert somit den Bot gleichzeitig. Das liegt daran, dass der Bot den Action Space so einschränkt, dass immer

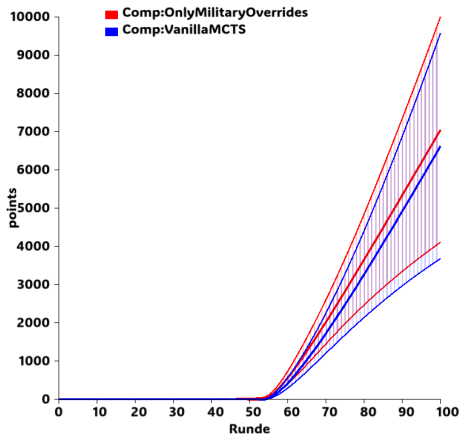
nur die Aktionen in der Nähe der letzten Aktion betrachtet werden. Damit können die betrachteten Regionen deutlich stärker optimiert werden, da hier eine deutlich größere Suchtiefe erreicht werden kann. Diese Herangehensweise hat aber 2 Nachteile. Erstens gibt es keine Garantie, dass der Bot alle Gebiete regelmäßig betrachtet und verbessert. Zweitens und noch deutlich wichtiger, kann sich mit diesem Pruning nur selten gegen gegnerische Angriffe gewehrt werden, nämlich nur dann, wenn die letzte Aktionen zufällig in der Nähe des Angriffs war. Diese Probleme können mit relativ einfachen Schritten behoben werden. So sollten alle Felder, die in der letzten Runde vom Gegner angegriffen wurden oder dort in der Nähe sind ebenfalls betrachtet werden. Außerdem sollte jedes Feld alle 5 bis 10 Runden trotzdem betrachtet werden, damit kein Gebiet völlig vernachlässigt werden kann.

7.4.4 Only Military Overwrites Pruning

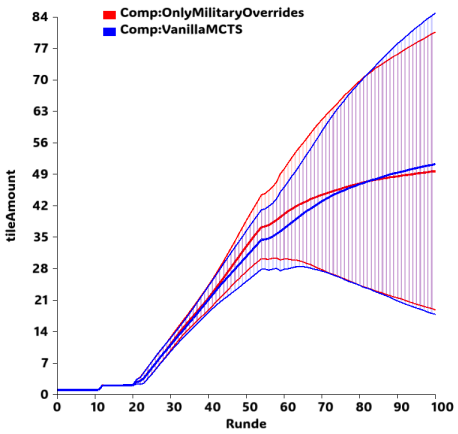
[ET]



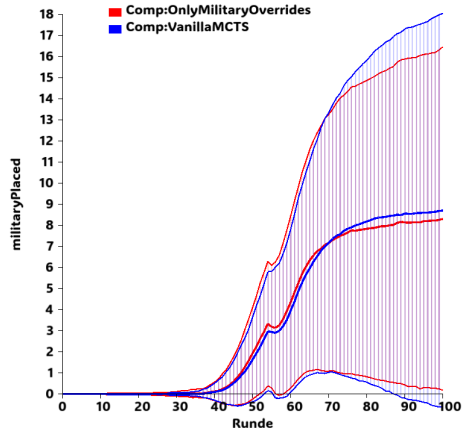
(a) Income Points



(b) Points



(c) Tile Amount

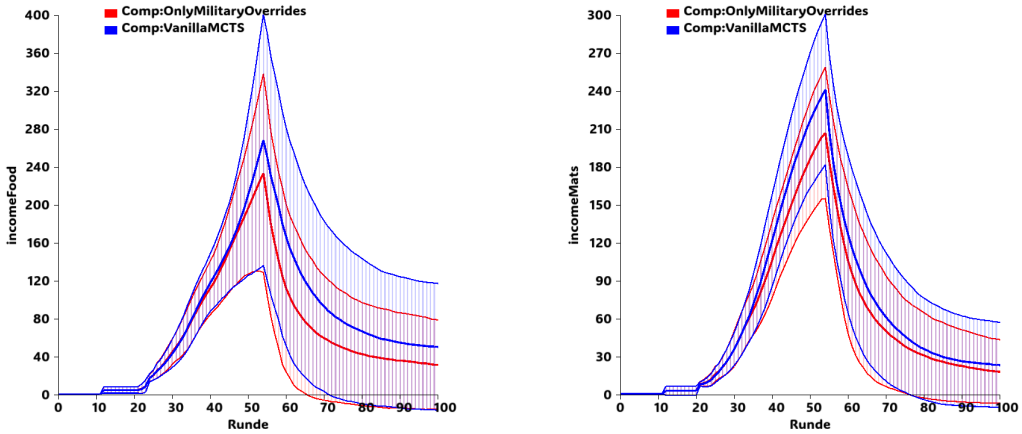


(d) Military Placed

Abbildung 7.14: Only Military Overwrides Pruning

Bei dem T-Test ist der t-Wert 3,24 mit einem Freiheitsgrad von 1998,00 und damit deutlich über dem kritischen Wert von 1,96. Damit ist das Ergebnis eindeutig statistisch signifikant.

Only Military Overwrites Pruning bringt einen Vorteil, aber dieser ist sehr klein. Der Effekt ist sehr ähnlich zu dem vom *Only Build Near Last Action Pruning*, nur deutlich weniger ausgeprägt. Von Runde 50 bis 85 ist das Pruning ein wenig stärker, danach wird der Vorteil tendenziell wieder kleiner (siehe Abbildung 7.14).



(a) Income Food **(b)** Income Material

Abbildung 7.15: Only Military Overwrites Pruning Food and Material

Interessant ist auch, dass der Bot in der Nahrungs- und Materialproduktion deutlich schlechter war, als der *VanillaMCTSBot*, wie man in Abbildung 7.15 sehen kann.

Das **Only Military Overwrites Pruning** bringt einen kleinen Vorteil im von Runde 50 bis 85. Das liegt daran, dass das Pruning nur das Überbauen von Militär- und Kulturgebäuden sowie das Überbauen mit ihnen erlaubt. Dadurch wird das Ausbauen von Nahrungs- und Materialgebäuden erschwert und der Bot hat hier schlechtere Werte als der *VanillaMCTSBot*. Deshalb fällt es dem Bot schwer, im späteren Spielverlauf mitzuhalten, da ihm schlichtweg die Ressourcen fehlen. Da das Pruning aber sowohl das militärische Spielen als auch die Punktegenerierung verstärkt schneidet der Bot am Ende trotzdem besser ab.

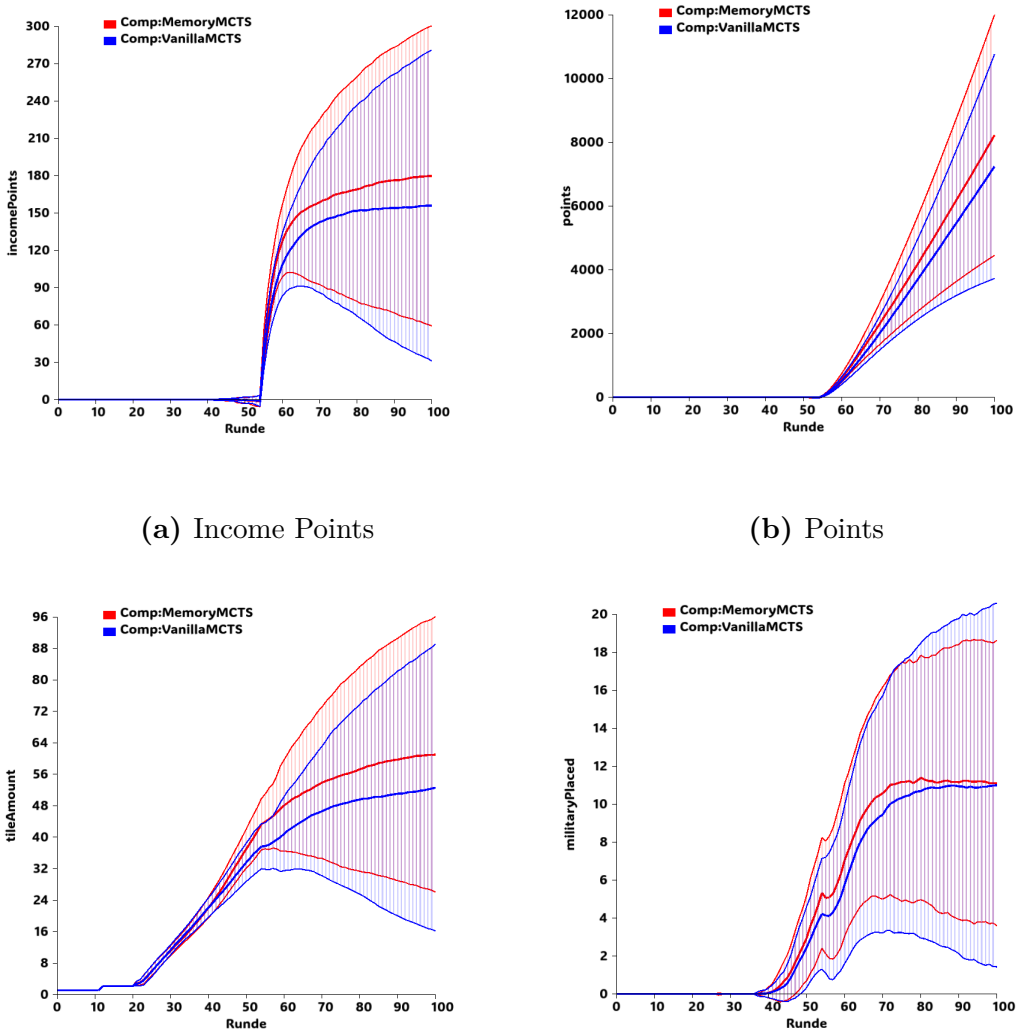
7.4.5 Auswertung

[ET] Die Forschungsfrage "Wie stark lässt sich die Performance des Algorithmus mit action space pruning verbessern?" kann wie folgt beantwortet werden. Die Performance des Algorithmus kann durch verschiedene Pruningverfahren erheblich verstärkt, bei einem Pruning sogar vedoppelt werden. Dabei ist anzumerken, dass die simplen Prunings mitunter besser funktioniert haben, da sie keine extra Rechenzeit benötigen.

Trotzdem zeigt der Algorithmus noch Schwächen, die sich nicht durch erhöhte Performance verhindern lassen. So kann der Algorithmus trotzdem nur wenige Züge in die Zukunft schauen und ist damit anfällig gegen Langzeitpläne. Strategien wie das Einkesseln des Gegners früh im Spiel, bereiten dem Bot weiterhin Probleme.

7.5 Memory MCTS

[ET]



(a) Income Points

(b) Points

(c) Tile Amount

(d) Military Placed

Abbildung 7.16: Memory MCTS

Bei dem T-Test ist der t-Wert 2,83 mit einem Freiheitsgrad von 439,86 und damit deutlich über dem kritischen Wert von 1,96. Damit ist das Ergebnis eindeutig statistisch signifikant. Die Freiheitsgrade sind hier deutlich kleiner als bei den anderen Pruningverfahren. Das liegt daran dass hier nur 111 Wiederholungen gemacht wurden. Da das Ergebnis trotzdem statistisch signifikant ist, ist das aber kein Problem.

Der Memory-MCTS hat, im Gegensatz zu den anderen Pruningverfahren, noch andere eigene Parameter, die angepasst werden können. Dieses Experiment untersuchte mit folgenden geschätzten Parametern, ob das Memory überhaupt einen Vorteil bringt.

- **Memory Exploration Parameter**: 0.5
- **Memory Tournament Selection**: 3

- **Memory Reset Strength**: 0.05
- **Memory Learning Rate**: 0.5
- **Memory Reset Point**: 1

Mit diesen Parametern stärkt das Memory den Bot durchgängig (siehe Abbildung 7.16). Der Vorteil flacht gegen Ende zwar wieder etwas ab, aber das Memory erreicht permanent mehr Income Points und mehr Tiles. Der Vorteil ist allerdings relativ klein. Interessant ist auch, dass das Memory deutlich mehr Tiles erreicht, obwohl es nicht wirklich viel mehr Militärgebäude baut. Dass das Memory gegen Ende wieder schlechter wird, könnte daran liegen, dass es sich Aktionen als schlecht merkt, die gegen Ende wieder sehr gut sind. Das der Vorteil generell nur so gering ist, liegt aber mit Sicherheit auch an der Wahl der Parameter. Diese wurden händisch ausgewählt und sind daher nur Schätzungen.

7.6 Der beste Memory-MCTS

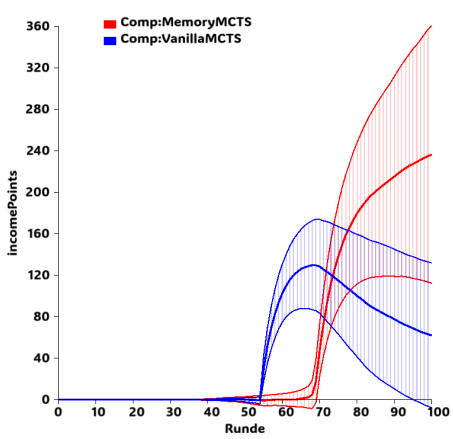
[ET] Um die optimalen Parameter und damit die größte Verbesserung des memory-MCTS zu bestimmen, wurde eine weitere HPO gestartet, bei der das Memory angeschaltet war. Zu der HPO ist zu sagen, dass sie erstens nicht besonders lange ging. Insgesamt wurde nur 16 Generationen lang optimiert. Zweitens wurden hier nicht nur die Parameter des Memories optimiert, sondern auch die generellen Parameter des Bots. Damit sind die Ergebnisse nicht mehr ganz vergleichbar zu den anderen Pruningverfahren, da diese ihre Parameter nicht explizit auf das Pruning anpassen konnten. Trotzdem sind Ergebnisse interessant um das Potenzial des Memories zu beurteilen. Im Folgenden wird der stärkste Bot aus der HPO weiter betrachtet, der folgende Prameter hat.

- **Simulationstiefe**: 6
- **Simulationsanzahl**: 1
- **Vanishing-Backpropagation**: 0.83
- **Winner-Bias**: 0.14
- **Tree-Policy**: *Greedy*
- **Simulationsstrategie**: *JustWaiting*
- **Material-Param**: 1,42
- **Food-Param**: 0.20
- **Point-Param**: 1.99
- **Tile-Param**: 5.71
- **Balance-Param**: 0.14
- **PointInc-Param**: 1.34
- **Memory Exploration Parameter**: 0.92
- **Memory Tournament Selection**: 5
- **Memory Reset Strength**: 0.32
- **Memory Learning Rate**: 1,52
- **Memory Reset Point**: 0.14

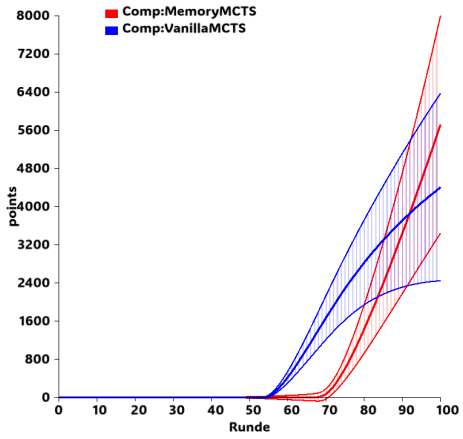
Dieser Bot wurde dann wie die anderen Pruningverfahren in 500 Spielen gegen den *VanillaMCTSBot* getestet.

Bei dem T-Test ist der t-Wert 13,78 mit einem Freiheitsgrad von 1955,29 und damit deutlich über dem kritischen Wert von 1,96. Damit ist das Ergebnis eindeutig statistisch signifikant.

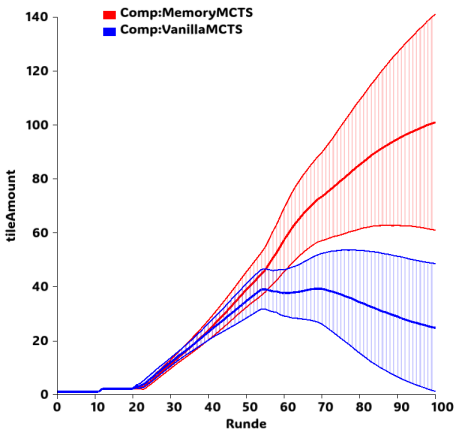
Bei den Ergebnissen in Abbildung 7.17 kann man die stärksten Unterschiede zwischen den konkurrierenden Bots erkennen. Der Memory-MCTS fokussiert sich erst später auf Punkte (ca. 14 Runden später) spielt aber vorher sehr aggressiv und nimmt so häufig



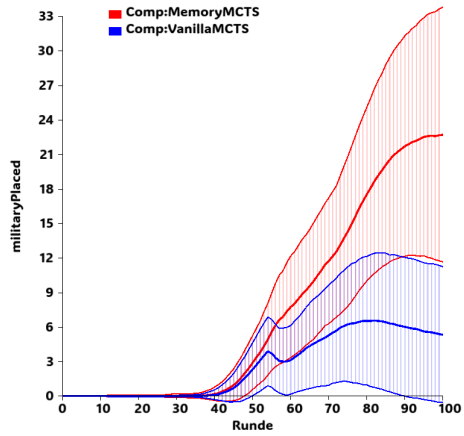
(a) Income Points



(b) Points



(c) Tile Amount



(d) Military Placed

Abbildung 7.17: Memory MCTS HPO optimized

große Teile des *VanillaMCTSBot*-Territoriums ein. Das führt dazu, dass der Memory-MCTS am Ende des Spiels im Durchschnitt 5600 Punkte und damit 1400 Punkte mehr als der *VanillaMCTSBot* mit 4400 Punkten hat.

Die HPO hatte somit einen deutlichen Einfluss auf den Memory-MCTS. Der Bot ist erwartungsgemäß nochmal deutlich besser und der Punktevorsprung am Ende deutlich größer als bei dem vorherigen Memory-MCTS. Interessant ist auch, dass sich der Bot so entwickelt hat, dass er länger offensiv spielt und erst ca. 14 Runden später als der *VanillaMCTSBot* mit dem Punkte sammeln beginnt. Außerdem haben sich die Parameter überraschend weiterentwickelt. So ist der *MemoryExplorationParameter* bei 0.92 und damit wird nur in 8 Prozent der Fälle ein Knoten weiter exploited. Die *LearningRate* ist mit 1,52 deutlich über 1. Das hat zur Folge, dass immer nur der neue Wert im Memory gespeichert wird und der alte Wert nur negativ mit einfließt (siehe Abschnitt 6.5). Ein Wert über 1 war hier nicht vorgesehen, scheint aber am besten zu funktionieren. Am überraschendsten ist allerdings die Entwicklung der Parameter *ResetStrength* und *ResetPoint*,

die eigentlich dafür sorgen sollen, dass alte Aktionen nach ein paar Runden reevaluiert werden. Diese sorgen jetzt für das genaue Gegenteil und setzen die Evaluationen aller Aktionen jede Runde nach unten. So hat eine Aktion nach ein paar Runden keine Chance mehr gewählt zu werden.

Insgesamt kann in Bezug auf die Forschungsfrage "Wie viel hilft die Einführung eines Gedächtnisses dabei, den Algorithmus schneller zu machen?" festgehalten werden, dass das Memory grundsätzlich sehr gut funktioniert und neue Strategien möglich macht. Es hat auch den Vorteil, dass es wenig Anpassungen braucht, um effektiv eingesetzt werden zu können. Das Spiel könnte sich stark verändern und das Memory wäre wahrscheinlich genauso einsetzbar. Allerdings müssen Systeme, wie das Zurücksetzen nach einer Runde, und Parameter für weitere Betrachtung nochmal angepasst werden.

8 Fazit

[ET] Das Einbauen einer Gegner-KI mit Hilfe des MCTS ist gut umsetzbar, wenn man die Möglichkeit hat, das ganze System über einen längeren Zeitraum hinweg zu optimieren.

8.1 Zusammenfassung

[ET] In dieser Arbeit wurde anhand eines selbstgebauten Strategiespiels (*Quick Civ*) getestet, wie gut sich der MCTS für die Gegner KI in komplexen 4X-Strategiespielen eignet. Da die Stärke des MCTS mit mehr möglichen Simulationen pro Sekunde steigt, wurde beim Entwickeln ein starker Fokus auf Leistungsoptimierung gesetzt. Strategiespiele sind oft sehr komplex und bieten viele mögliche Aktionen. In diesem Fall wurde der Action Space in Teilen des Spiels so groß, dass nicht mehr jede Aktion evaluiert werden kann. Um dieses Problem zu bekämpfen wurden in dieser Arbeit zwei Lösungsansätze verfolgt. Erstens den Aktionsraum mit Hilfe von Action Space Pruning zu verkleinern. Zweitens wurde ein Memory für den MCTS eingeführt, welches sich die Evaluierungen der Aktionen über mehrere Runden hinweg speichern kann. Diese Optimierung kann vorgenommen werden, da die Stärke der Aktionen in *Quick Civ* sich meist über mehrere Runden hinweg nicht drastisch verändert. Mit dem Memory gibt es einen dynamischen Weg schlechte Aktionen auszuschließen und somit auch den Action Space zu verkleinern. Um eine gute Vergleichsmetrik zu haben wurde der beste MCTS Bot ohne Verbesserungen (*VanillaMCTSBot*) bestimmt, indem ein evolutionärer Algorithmus eine HPO durchgeführt hat. Der resultierende Bot wurde dann zuerst gegen einen Random- und einen Greedy-Bot getestet welche er mit großem Abstand besiegt. Die Forschungsfrage "Wie anwendbar ist MCTS für 4X-Spiele und wie muss man die Parameter anpassen um eine optimale Performance zu erreichen?" ist schwierig zu beantworten, da die Einschätzung des Spielverhaltens relativ subjektiv ist. Man kann allerdings sagen, dass sich der Bot grundsätzlich kompetent spielt und mit der richtigen Ausgangslage auch gut gegen einen Menschen gewinnen kann. Trotzdem hat der Bot einige große Probleme. Erstens muss das Spiel stark optimiert sein, damit der Algorithmus überhaupt anwendbar ist. Dann wird er zweitens ohne Optimierung im späteren Spielverlauf durch die Menge an Aktionen überfordert. Generell ist die Suchtiefe des MCTS relativ gering und damit werden langfristige Entscheidungen nicht entsprechend gut genug durchgerechnet. Hier müssen bei einer Weiterentwicklung dann andere Systeme greifen, um größere Strategische Entscheidungen zu treffen.

Danach wurde der *VanillaMCTSBot* gegen sich selbst mit je einem von vier Pruning Verfahren getestet, indem die Bots 100 Spiele gegeneinander gespielt haben. Dabei brachte ein Pruning Verfahren einen Nachteil, 2 einen kleinen Vorteil und eins einen großen Vorteil. Die Forschungsfrage "Wie stark lässt sich die Performance des Algorithmus mit action space pruning verbessern?" lässt sich wie folgt beantworten. Die Performance lässt

sich deutlich durch die Benutzung von Action Space Pruning verbessern und verdoppelte die Stärke des Bots in einem Fall sogar. Dies ist wie schon beschrieben allerdings auch notwendig, da der Algorithmus ohne Optimierung im späteren Spiel durch die Menge an Aktionen überfordert ist.

In den letzten Experimenten wird der *VanillaMCTSBots* mit dem Memory-MCTS verglichen, indem beide wieder 100 Spiele gegeneinander spielen. Zuerst hat der Memory-MCTS händisch ausgesuchte Parameter und ist damit etwas besser als der *VanillaMCTSBot*. In einem zweiten Experiment sind die Parameter des Memory durch eine HPO optimiert. Hier spielt der Memory-MCTS eine deutlich andere Strategie als sonst. Die Forschungsfrage "Wie viel hilft die Einführung eines Gedächtnisses dabei, den Algorithmus schneller zu machen." kann somit wie folgt beantwortet werden. Der Algorithmus wird durch die Einführung eines Gedächtnisses deutlich stärker, vor allem mit optimierten Parametern.

In der Arbeit wurde gezeigt, dass sich MCTS als Algorithmus für Gegner-KI in 4X-Spielen eignet, wenn man das Spiel und den Algorithmus gut genug optimiert.

8.2 Ausblick

[ET] *Quick Civ* ist aktuell aktiv in Entwicklung und soll innerhalb der nächsten 1,5 Jahre als vollwertiges Spiel veröffentlicht werden. Dabei soll die beste Form des Bots natürlich auch im fertigen Spiel Anwendung finden. Da jetzt schon neue Mechaniken wie Wissenschaft und Kriegsnebel (unvollständige Spielinformation) existieren, werden weitere Anpassungen am Bot erforderlich sein. Außerdem soll es verschiedene Bots geben, die sich aufgrund verschiedener Parameter unterschiedlich verhalten und damit verschiedene große Anführer widerspiegeln können. Auch für den Algorithmus selbst gibt es noch Verbesserungsideen. Da das Einschränken des Action Spaces so extrem wichtig ist, hatten wir auch die Idee vor dem MCTS einen Planungsalgorithmus zu starten. Dieser würde einen optimierten Aufbau für die nächsten 10-15 Aktionen über einen EA erstellen, und der MCTS würde dann nur noch überlegen, was für diese Aktionen die beste Reihenfolge ist.

Anhang

A. Bot-Initialisierung

[EN] Bei der Initialisierung eines Bots für die Abschnitt 6.3 werden die Gene zufällig initialisiert, jedoch innerhalb vordefinierter Grenzen. Diese werden hier aufgelistet.

Code 1: Initialisierung der Greedy-Gene innerhalb vordefinierter Grenzen

```
turnsForCityCenter = random.randomInt(50) + 1;
materialParam = random.random() * 5;
foodParam = random.random() * 5;
pointParam = random.random() * 5;
tileParam = random.random() * 5;
balanceParam = random.random() * 5;
pointInc = random.random() * 5;
```

Code 2: Initialisierung der MCTS-Gene innerhalb vordefinierter Grenzen

```
explorationParameter = random.random() * 2 + 1
simulationDepth = random.randomInt(10) + 1
simulationAnzahl = random.randomInt(10) + 1
vanishingBackpropParam = random.random()
materialParam = random.random() * 2
foodParam = random.random() * 2
pointParam = random.random() * 2
balanceParam = random.random() * 2
tileParam = random.random() * 10
pointInc = random.random() * 2
winnerBiasParameter = random.random()
epsilonGreedyParameter = random.random() * 2

defaultPolicyParallelity = random.random()
treePolicyType = random.random()
defaultPolicyEnemy = random.random()
defaultPolicySelf = random.random()

if vanilla:
  useMemory = false
  onlyMilitaryOverwrites = false
  greedyNodesOnly = false
  inLateGameNoEarlyGameBuildings = false
  onlyBuildNearLastAction = false
else:
  useMemory = random.random() > 0.5
  onlyMilitaryOverwrites = random.random() > 0.5
  greedyNodesOnly = random.random() > 0.5
  inLateGameNoEarlyGameBuildings = random.random() > 0.5
  onlyBuildNearLastAction = random.random() > 0.5
  onlyBuildNearLastActionRange = rrandom.randomInt(4)
  memoryExplorationParameter = random.random()
  memoryTournamentSelectionSize = random.randomInt(10)
  memory_resetStrength = random.random() * 2
  memory_resetPoint = random.random()
  memory_learningRate = random.random() * 2
```

B. Gebäudeliste

[EN]

Gebäude	Beschreibung
Aquädukt	Nahrung für angrenzende Housing-Gebäude
Stallung	Nahrung für angrenzende unbebaute Felder
Baracke	Viel Militärischen Druck auf angrenzende Felder
Metzgerei	Punkte für angrenzende Nahrungs-Gebäude
Fort	Wenig Militärischen Druck auf angrenzende Felder
Stadtzentrum	Benötigt für Expansion
Farm	Nahrung
Schmiede	Punkte für angrenzende Militärgebäude und Produktion für angrenzende Minen
Festung	Viel Militärischen Druck auf eigenem Feld
Sammlerhütte	Nahrung für Wälder
Haus	Material für angrenzende Nicht-Housing-Gebäude
Ausguck	Wenig Militärischen Druck in Reichweite 2
Holzfäller	Material für angrenzende Wälder
Sägewerk	Material für angrenzende Holzfäller
Söldnerlager	Viel Militärischen Druck auf angrenzende Felder und negative Ressourcenproduktion für angrenzende eigene Felder
Mine	Material für angrenzende Berge
Kirche	Punkte für angrenzende unbebaute Felder und Berge auf eigenem Feld
Museum	Punkte für Housing-Gebäude in Reichweite 2
Plantage	Nahrung für angrenzende Plantagen
Steinbruch	Material für Berge auf eigenem Feld
Siedlung	Produktion für Nicht-Housing-Gebäude in Reichweite 2
Katapult	Mittlere Militärischen Druck in Reichweite 2
Steinkreis	Punkte für freie Naturelemente auf eigenem Feld
Suq	Ressourcenproduktion für angrenzende gegnerische Felder und Hohe Ressourcenproduktion für angrenzende gegnerische Handelsposten
Taverne	Punkte für angrenzende Housing-Gebäude
Handelsposten	Ressourcenproduktion für angrenzende gegnerische Felder
Windmühle	Nahrung für angrenzende Farmen
Werkstatt	Material für angrenzende Material-Gebäude

Tabelle 1: Überblick über alle Gebäude des Spiels

Literaturverzeichnis

[ABCS20] Sinan Ariyurek, Aysu Betin-Can, and Elif Surer. Enhancing the monte carlo tree search algorithm for video game testing. In *2020 IEEE Conference on Games (CoG)*, pages 25–32. IEEE, 2020.

[AHH10] Broderick Arneson, Ryan B. Hayward, and Philip Henderson. Monte carlo tree search in hex. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):251–258, 2010. doi:10.1109/TCIAIG.2010.2067212.

[alp16] Alphago versus lee sedol, 2016. URL: https://de.wikipedia.org/wiki/AlphaGo_gegen_Lee_Sedol.

[Bli10] Blizzard. Starcraft, 2010. URL: https://starcraft2.blizzard.com/de-de/.

[BPW+12] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[CF10] Paolo Ciancarini and Gian Piero Favini. Monte carlo tree search in kriegspiel. *Artificial Intelligence*, 174(11):670–684, 2010.

[CGDNV+21] Alba Cotarelo, Vicente García-Díaz, Edward Rolando Núñez-Valdez, Cristian González García, Alberto Gómez, and Jerry Chun-Wei Lin. Improving monte carlo tree search with artificial neural networks without heuristics. *Applied Sciences*, 11(5):2056, 2021.

[Che16] Jim X. Chen. The evolution of computing: Alphago. *Computing in Science Engineering*, 18(4):4–7, 2016. doi:10.1109/MCSE.2016.74.

[CIK+23] Harshit Choudhary, Aniket Inamdar, Swapnil Kharade, Anunay Maheshwari, and Preeti Kale. Comparative analysis of minimax algorithm with alpha–beta pruning optimization for chess engine. In Samarjeet Borah, Tapan K. Gandhi, and Vincenzo Piuri, editors, *Advanced Computational and Communication Paradigms*, pages 237–247, Singapore, 2023. Springer Nature Singapore.

[CIV] Civ vi ai forum. URL: https://forums.civfanatics.com/threads/are-behavior-trees-a-thing-of-the-past.631213/.

[Cou06] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, pages 72–83. Springer, 2006.

[DHGJ+21] Alexander Dockhorn, Jorge Hurtado-Grueso, Dominik Jeurissen, Linjie Xu, and Diego Perez-Liebana. Game state and action abstracting monte carlo tree search for general strategy game-playing. In *2021 IEEE Conference on Games (CoG)*, pages 1–8. IEEE, 2021.

[DV13] Barry De Ville. Decision trees. *Wiley Interdisciplinary Reviews: Computational Statistics*, 5(6):448–455, 2013.

[EC18] Elmer R. Escandon and Joseph Campion. Minimax checkers playing gui: A foundation for ai applications. In *2018 IEEE XXV International Conference on Electronics, Electrical Engineering and Computing (INTERCON)*, pages 1–4, 2018. doi:10.1109/INTERCON.2018.8526375.

[ESES15] Agoston E Eiben, James E Smith, AE Eiben, and JE Smith. What is an evolutionary algorithm? *Introduction to evolutionary computing*, pages 25–48, 2015.

[EUI] Euiv forum. URL: https://forum.paradoxplaza.com/forum/threads/how-does-the-eu4-ai-work.1237859/.

[Gam] Game star article. URL: https://www.gamestar.de/artikel/civilization-6-krone-mit-kratzer,3303691,fazit.html.

[Gam23] Firaxis Games. Civilization vi, 2023. URL: https://civilization.2k.com/de-DE/civ-vi/.

[GS11] Sylvain Gelly and David Silver. Monte-carlo tree search and rapid action value estimation in computer go. *Artificial Intelligence*, 175(11):1856–1875, 2011.

[GSLL16] Xiaoxiao Guo, Satinder Singh, Richard Lewis, and Honglak Lee. Deep learning for reward design to improve monte carlo tree search in atari games. *arXiv preprint arXiv:1604.07095*, 2016.

[JTTR14] Niels Justesen, Bálint Tillman, Julian Togelius, and Sebastian Risi. Script- and cluster-based uct for starcraft. In *2014 IEEE conference on computational intelligence and games*, pages 1–8. IEEE, 2014.

[JWF18] Tobias Joppen, Christian Wirth, and Johannes Fürnkranz. Preference-based monte carlo tree search. In *Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz)*, pages 327–340. Springer, 2018.

[KS06] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

[KS08] Carl Kingsford and Steven L Salzberg. What are decision trees? *Nature biotechnology*, 26(9):1011–1013, 2008.

[Lor15] Richard Lorentz. Early playout termination in mcts. In *Advances in Computer Games: 14th International Conference, ACG 2015, Leiden, The Netherlands, July 1-3, 2015, Revised Selected Papers 14*, pages 12–19. Springer, 2015.

[MDCB23] Jyoti Madake, Chinmay Deotale, Geetai Charde, and Shripad Bhatlawande. Chess ai: Machine learning and minimax based chess engine. In *2023 International Conference for Advancement in Technology (ICONAT)*, pages 1–6, 2023. doi:10.1109/ICONAT57137.2023.10080746.

[ML18] Rubens Moraes and Levi Lelis. Asymmetric action abstractions for multi-unit control in adversarial real-time games. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[MMLN18] Rubens Moraes, Julian Marino, Levi Lelis, and Mario Nascimento. Action abstractions for combinatorial multi-armed bandit tree search. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 14, pages 74–80, 2018.

[Mül] Tim-Hendrik Müller. Klassische ki in computerspielen.

[OSM20] Abdessamed Ouessai, Mohammed Salem, and Antonio M. Mora. Improving the performance of mcts-based µrts agents through move pruning. In *2020 IEEE Conference on Games (CoG)*, pages 708–715, 2020. doi:10.1109/CoG47356.2020.9231715.

[RFT14] Denis Robilliard, Cyril Fonlupt, and Fabien Teytaud. Monte-carlo tree search for the game of "7 wonders". In *Computer Games: Third Workshop on Computer Games, CGW 2014, Held in Conjunction with the 21st European Conference on Artificial Intelligence, ECAI 2014, Prague, Czech Republic, August 18, 2014, Revised Selected Papers 3*, pages 64–77. Springer, 2014.

[RM05] Lior Rokach and Oded Maimon. Decision trees. *Data mining and knowledge discovery handbook*, pages 165–192, 2005.

[Roe12] G Roelofs. Monte carlo tree search in a modern board game framework. *Research paper available at umimaas. nl*, 2012.

[Roy19] R Roy. Ml—monte carlo tree search (mcts). *Geeks for Geeks,(available at https://www. geeksforgeeks. org/ml-monte-carlo-tree-search-mcts/, visited Feb. 10, 2020)*, 2019.

[Sam18] Siim Sammul. Learning to play monopoly with monte carlo tree search. *Computer science bachelor thesis, School of Informatics, University of Edinburgh*, 2018.

[SCS09] István Szita, Guillaume Chaslot, and Pieter Spronck. Monte-carlo tree search in settlers of catan. In *Advances in computer games*, pages 21–32. Springer, 2009.

[ŚGSM23] Maciej Świechowski, Konrad Godlewski, Bartosz Sawicki, and Jacek Mańdziuk. Monte carlo tree search: A review of recent modifications and applications. *Artificial Intelligence Review*, 56(3):2497–2562, 2023.

[SHM+16] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[SHS+17] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

[SHS+18] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[sto22] Stockfish - an open source chess engine, 2022. URL: http://stockfishchess.org/.

[VNSB20] Viktor Voss, Liudmyla Nechepurenko, Rudi Schaefer, and Steffen Bauer. Playing a strategy game with knowledge-based reinforcement learning. *SN Computer Science*, 1(2):78, 2020.

[VSS17] Tom Vodopivec, Spyridon Samothrakis, and Branko Ster. On monte carlo tree search and reinforcement learning. *Journal of Artificial Intelligence Research*, 60:881–936, 2017.

[Wika] Wikipedia. Student's t-test. URL: https://en.wikipedia.org/wiki/Student's_t-test.

[Wikb] Wikipedia. Utility ai.

[Win17] Mark HM Winands. Monte-carlo tree search in board games. In *Handbook of Digital Games and Entertainment Technologies*, pages 47–76. Springer, 2017.

[XMM18] Chenjun Xiao, Jincheng Mei, and Martin Müller. Memory-augmented monte carlo tree search. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

[XNP+24] Chunyu Xuan, Yazhe Niu, Yuan Pu, Shuai Hu, Yu Liu, and Jing Yang. Rezero: Boosting mcts-based algorithms by backward-view and entire-buffer reanalyze, 2024. URL: https://arxiv.org/abs/2404.16364, arXiv:2404.16364.

[ZHR19] Alexander Zook, Brent Harrison, and Mark O Riedl. Monte-carlo tree search for simulation-based strategy analysis. *arXiv preprint arXiv:1908.01423*, 2019.

Appendix: STATEMENT OF AUTHORSHIP of the Student

Thesis: Monte Carlo Tree Search in 4X-Spielen

Name: Eric Surname: Neumann

Date of birth: 27.06.2003 Matriculation no.: 238171

I herewith assure that I have written the present thesis independently, that the thesis has not been partially or fully submitted as graded academic work and that I have used no other means than the ones indicated. I have indicated all parts of the work in which sources are used according to their wording or according to their meaning.

I am aware of the fact that violations of copyright can lead to injunctive relief and claims for damages of the author as well as a penalty by the law enforcement agency.

Furthermore, I confirm that I am aware that the use of content (including but not limited to text, figures, images and code) generated by artificial intelligence (AI) in the thesis must be disclosed. In those cases I have specified the AI system used, I have marked the specific sections of the thesis where AI-generated content was used, and I have provided a brief explanation of the level of detail at which the AI system was used to generate the content. I also stated the reason for using the tools.

I assure that even if a generative AI system has been used, the scientific contribution has been made entirely by myself.

Magdeburg, 18.06.2025

Signature

Appendix: STATEMENT OF AUTHORSHIP of the Student

Thesis: Monte Carlo Tree Seach in 4X-Spielen

Name: Emanuel Surname: Thon

Date of birth: 02.03.2003 Matriculation no.: 238201

I herewith assure that I have written the present thesis independently, that the thesis has not been partially or fully submitted as graded academic work and that I have used no other means than the ones indicated. I have indicated all parts of the work in which sources are used according to their wording or according to their meaning.

I am aware of the fact that violations of copyright can lead to injunctive relief and claims for damages of the author as well as a penalty by the law enforcement agency.

Furthermore, I confirm that I am aware that the use of content (including but not limited to text, figures, images and code) generated by artificial intelligence (AI) in the thesis must be disclosed. In those cases I have specified the AI system used, I have marked the specific sections of the thesis where AI-generated content was used, and I have provided a brief explanation of the level of detail at which the AI system was used to generate the content. I also stated the reason for using the tools.

I assure that even if a generative AI system has been used, the scientific contribution has been made entirely by myself.

Magdeburg, 18.06.2025

E. Thon

_____

Signature