Untersuchung baumbasierter Suchverfahren für intelligente Agenten und deren Anwendung für rundenbasierte Strategiespiele

Bernd Nötscher

1012974

Bachelorthesis

Zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)

31. Juli 2015

Frankfurt University of Applied Sciences

Fachbereich 2 – Informatik und Ingenieurwissenschaften
Studiengang Informatik (B.Sc.)

Referent: Prof. Dr. Thomas Gabel
Korreferent: Prof. Dr. Christian Baun

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Vorwort

Während meines Studiums wurden unter anderem verschiedene Suchverfahren für Computerspiele erläutert. Zu dieser Zeit reifte in mir die Vorstellung, für meine Bachelorarbeit ein eigenes Computerspiel mit einem möglichst intelligenten Computergegner zu entwickeln.

Schließlich habe ich mich für die Idee, ein rundenbasiertes Strategiespiel zu erstellen, entschieden. Als Vorlage für das Computerspiel wählte ich das bekannte Brettspiel „Risiko“ aus. Um das Spiel interessanter zu gestalten, habe ich zusätzlich die Regeln des Kartenspiels „Battle For Stalingrad“ verwendet und beide Spiele miteinander kombiniert.

Daraufhin habe ich viele eigene Einfälle zur Regelerweiterung ausprobiert und eingebaut, um nach Abschluss dieser Arbeit das Computerspiel als ansprechende App für Android und iOS zu veröffentlichen.

1. Einleitung

Im Bereich der künstlichen Intelligenz stellen moderne Brett- und Computerspiele eine große Herausforderung da. Anders als klassische Brettspiele wie z. B. Schach, können diese Spiele nicht-deterministisch sein, möglicherweise sogar Spielelemente mit unvollständiger Information enthalten (z. B. verdeckte Spielkarten) oder sie können mehr als zwei Spieler unterstützen. Klassische Suchverfahren wie z. B. das Minimax-Verfahren [1] sind sehr erfolgreich beim Schachspielen[1], führen aber bei modernen Brett- und Computerspielen zu schlechten Ergebnissen. Eine Übersicht über den aktuellen Fortschritt der KI im Bereich der Brettspiele ist in Abbildung 1.1 zusammengefasst.

Das in Asien beliebte Spiel Go[2] zählt zu den klassischen Brettspielen, hat aber dennoch eine so große Komplexität, dass klassische baumbasierte Suchverfahren keinen künstlichen Agenten mit überragenden Spielfähigkeiten ermöglichen. Forschungsanstrengungen führten schließlich im Jahr 2006 [2] zu einem ganz neuen Suchverfahren, der Monte-Carlo-Baumsuche (im Englischen „Monte-Carlo Tree Search“), was letztendlich den künstlichen Agenten für Go zu einem Durchbruch verholfen hat. Heute spielen Computergegner für Go, die diesen Algorithmus nutzen auf dem Niveau von menschlichen Weltklassespielern.

Dieser Durchbruch führte zu großer Aufmerksamkeit, da bis dahin die allgemeine Meinung vorherrschte, dass es noch Jahre dauern würde, bis ein gutes Suchverfahren für Go geben würde. Die Computerspieleindustrie griff diesen neuen Algorithmus bereits auf: Die Monte-Carlo-Baumsuche wird kommerziell eingesetzt, beispielsweise in dem bekannten Spiel „Total War“[3] (ein komplexes, rundenbasiertes Strategiespiele für Computer [3]).

In dieser Arbeit wird „Monte-Carlo Tree Search“(MCTS) als baumbasiertes Suchverfahren und deren Anwendung für rundenbasierte Strategiespiele untersucht. Zusätzlich wird ein komplexes Computerspiel, welches als Umgebung für die Untersuchung dient, erläutert.

Dieses extra für die Untersuchung erstellte Computerspiel, genannt „The American Civil War“, basiert auf einer Kombination der Spielkonzepte des Brettspiels „Risiko“ und dem Kartenspiel „Battle For Stalingrad“.

[1] Dem Computer Deep Blue gelang es 1996 als erstem Computer den damaligen Schachweltmeister Kasparov zu schlagen (`http://www.welt.de/print-welt/article652666/Computer-schlaegt-Kasparow.html`).

[2] `https://de.wikipedia.org/wiki/Go_(Spiel)`

[3] Total War: Rome II mit 1,1 Millionen verkauften Exemplare erschien im September 2013 (`https://de.wikipedia.org/wiki/Total_War#Total_War:_Rome_II`).

Die Komplexität des Spiels ist sehr groß, was die Entwicklung eines Computergegners anspruchsvoll macht, aber gerade auch deshalb ist dieses Spiel eine gute Testumgebung für die Untersuchung und Anwendung von MCTS.

Die Bachelorarbeit ist folgendermaßen strukturiert. Zuerst werden im zweiten Kapitel die Regeln des Spiels und der Spielaufbau erläutert. Außerdem wird dort die Komplexität des Spiels analysiert. Die Umgebung (also das Computerspiel), welche alle Regeln und die Spielwelt mit einer grafischen Oberfläche für die menschlichen Spieler beinhaltet, ist die Basis für die restliche Arbeit. Diese Implementierung wird im dritten Kapitel beschrieben und gehört mit zum praktischen Teil dieser Thesis. Bevor ein intelligenter Agent erstellt wurde, war es zuerst die Aufgabe eine allgemeine Schnittstelle zu definieren, die es ermöglicht mit der Spieleumgebung zu interagieren.

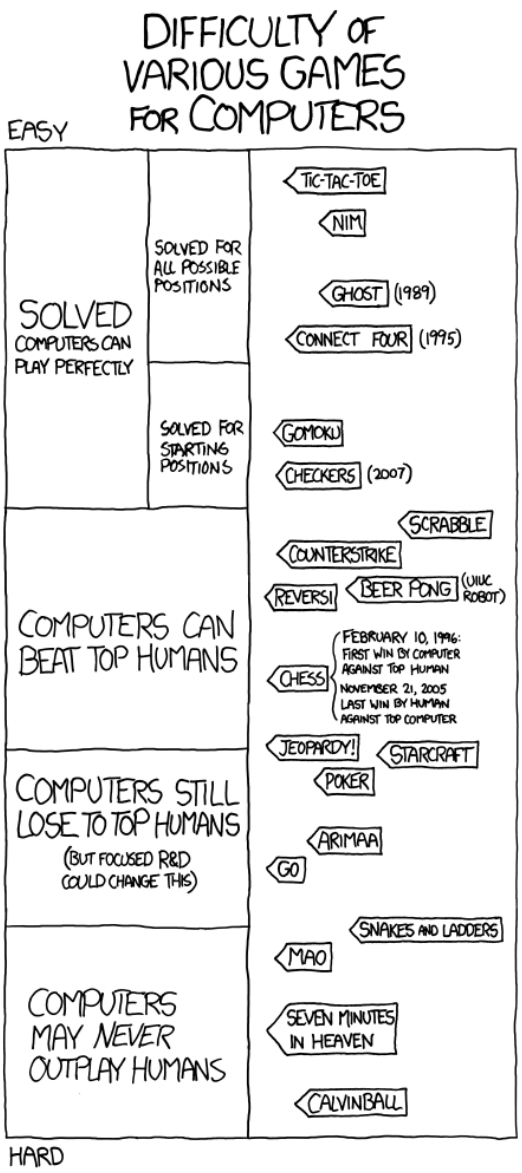

Abbildung 1.1.: Stand der Entwicklung[4]

Zur Unterstützung der Auswertung des intelligenten Agenten dienen zwei verschiedene zufallsbasierte Agenten und ein heuristischer Agent. Zuerst wurde ein zufallsbasierter Agent erstellt, der alle Entscheidungen per Zufall fällt (siehe drittes Kapitel). Es folgt eine kurze Analyse der Defizite, die dieser Agent aufzeigt und eine mögliche Verbesserung, die in einem neuen Agenten, dem erweiternden zufallsbasierten Agenten, resultiert. Zum besseren Vergleich wurde außerdem ein leistungsfähiger heuristischer Agent entwickelt. Dieser versucht, anhand handgeschriebener vom Autor dieser Arbeit ausgedachter einfacher Algorithmen, gute Spielzüge zu finden und auszuführen. Im vierten Kapitel werden baumbasierte Suchverfahren kurz vorgestellt und MCTS erläutert. Zum praktischen Teil dieser Thesis gehört auch die Implementierung der Monte-Carlo-Baumsuche um einen intelligenten Agent zu erschaffen. Dies wird im fünften Kapitel dargestellt. Um in der Lage zu sein, die Spielfähigkeit der Spieler genau zu beurteilen, wurden mehrere Experimente durchgeführt und die Leistung der verschiedene Agenten miteinander wurde verglichen. Die Methoden und Ergebnisse der Experimente werden im sechsten Kapitel vorgestellt. Am Ende bietet das siebte Kapitel ein Fazit zu dieser Arbeit.

[4]Quelle: `http://www.xkcd.com/1002/`

2. Ein rundenbasiertes Strategiespiel

In diesem Kapitel wird auf die rundenbasierten Strategiespiele eingegangen, die als Basis zur Untersuchung herangezogen wurden. Darauf folgt die Beschreibung der Spielaustattung und Spielregeln von The American Civil War in den Unterkapiteln 2.3.1 und 2.3.2. Die Wahrscheinlichkeiten und Komplexität dieses Spiels werden in 2.3.3 und 2.3.4 betrachtet.

2.1. Risiko

Risiko ist ein klassisches rundenbasiertes Strategiebrettspiel für zwei bis sechs Spieler veröffentlicht von Parker im Jahre 1959[1]. In den letzten Jahren sind einige Varianten

[1] Ursprünglich wurde es von Albert Lamorisse im Jahre 1957 als "La Conquete du Monde"konzipiert und kurze Zeit später von Parker angepasst (`https://de.wikipedia.org/wiki/Risiko_(Spiel)`).

Abbildung 2.1.: Brettspiel Risiko

mit Regelerweiterungen erschienen. Als Vorlage für diese Arbeit wurde das Spiel Star Wars - Die Original Triologie benutzt [4]. Beim klassischen Risiko stellt das Spielbrett die Welt dar, die in sechs Kontinente und 42 Territorien unterteilt ist. Es gibt Armeen, die auf diesen Territorien platziert werden können. Die Regeln des klassischen Risikos wurden im Laufe der Zeit erweitert, sodass es heute sogar spezielle Spielkarten gibt, die z. B. ganze angreifende Armeen abwehren oder sogar vom Spielbrett nehmen können. Ziel dieses Spiels ist es, alle gegnerischen Territorien zu erobern. Abbildung 2.1 zeigt die für die Implementierung verwendete Variante von Risiko.
Es gibt einige existierende Implementierungen als Computerspiele. Manche dieser Spiele sind kostenlos, andere sind Shareware oder sogar kommerzielle Titel [5].

Abbildung 2.2.: Kartenspiel Battle For Stalingrad

2.2. Battle For Stalingrad

Bei diesem Spiel handelt es sich um ein Kartenspiel von DVG [6] (siehe Abbildung 2.2). Zwei Spieler spielen gegeneinander die historische Schlacht von Stalingrad nach. Jeder Spieler erhält Einheiten und Aktionskarten und muss damit versuchen alle wichtigen Orte der Stadt Stalingrad besetzt zu halten um zu gewinnen. Das Spiel stellt die Stadt ähnlich einem Schachbrett in einem 5x5 Raster dar, in den Einheiten und Orte platziert sind. Der Spielverlauf ähnelt einem Kräfteringen mit Abnutzung des Gegners, denn schlussendlich gewinnt derjenige, der durch geschicktes Taktieren die vorhanden Ressourcen besser nutzt und bei Gefechten, zwischen den gegnerischen Einheiten, klug seine Aktionskarten einsetzt.
Bis zum Erscheinen dieser Arbeit gibt es von diesem Kartenspiel keine Computerspielumsetzung.

2.3. The American Civil War

The American Civil War ist ein Computerspiel, das neu für diese Abschlussarbeit erstellt wurde und basiert auf den Spielkonzepten von Risiko und Battle For Stalingrad. Zusätzlich wurden eigene von dem Autor der Arbeit erdachte Regeln eingebaut, um das Spiel interessanter zu gestalten. Trotz alledem gibt es bereits einige Computerspiele [7, 8, 9], die in ähnlicher weise die Idee umsetzen mittels Kombination von globaler Kartensicht und lokaler Schlachtansicht ein unterhaltsames Spiel anzubieten.
Grundsätzlich gilt: Bei diesem Spiel spielen zwei Kontrahenten gegeneinander. Es ist ein rundenbasiertes Spiel, bei dem ein Spieler die Rolle der Nordstaaten übernimmt, der andere die Südstaaten spielt. Nacheinander führen die Spielteilnehmer ihre Züge aus. Außerdem gibt es noch einen neutralen Spieler, der nur aufgrund spezieller Spielereignisse sich einer der beiden Seiten anschließt (Spieler Europa).

Das Spiel enthält drei Ansichten:

Landkarte

Die Landkarte, welche große Teile des Nordamerikanischen Kontinents mit seiner politische Unterteilung in einzelne Staaten und deren Territorien zeigt, ist die Hauptansicht des Spiels (Abbildungen 2,3 2.4 und 2.5). In dieser globalen Sicht ist der Spieler in der Lage, Einheiten seinen Territorien zuzuweisen und diesen Befehle zu geben wie beispielsweise bewege dich von Territorum A nach Territorum B oder greife aus Territorum A gegnerische Einheiten in Territorum B an.
Erfolgt ein Angriff, wird, bevor dieser ausgeführt wird, eine zweite Ansicht angezeigt.

Vor der Schlacht

In der Ansicht vor der Schlacht (Abbildung 2.6) können beide Spieler versuchen den Ausgang der Schlacht zu beeinflussen, indem bestimmte Spielkarten eingesetzt werden, die Auswirkungen auf die eigenen oder die gegnerischen Einheiten haben (z. B. Verstärkung anfordern oder mit dem Ausspielen der richtigen Spielkarte kann der Verteidiger den Abbruch der Schlacht erzwingen).

Schlacht

Nachdem keiner mehr Spielkarten ausspielen kann oder möchte kann der Angreifer den Befehl zum Starten der Schlacht geben, womit die dritte Ansicht, die Schlachtansicht gezeigt wird (Abbildung 2.7 und 2.8). Diese unterteilt den Bildschirm schachbrettartig in Felder. Auf der linken Seite befinden sich die Felder des Angreifers, auf der rechten Seite die des Verteidigers. Keine der beiden Seiten kann auf die gegnerischen Felder gelangen. Jedes eigene Feld kann beliebige eigene Einheiten aufnehmen. Um eine Schlacht zu gewinnen ist es notwendig rechtzeitig vor dem automatischen Schlachtabbruch nach 20 Runden die Felder in der Mitte des Schlachtfeldes zwischen der linken Seite des Angreifers und der rechten Seite des Verteidigers unter eigene Kontrolle zu bekommen. Dies geschieht durch Platzierung eigener Einheiten direkt auf den Feldern neben den Feldern der Mitte, was dazu führt das diese Felder der Mitte zu den eigenen zugerechnet werden, also unter eigener Kontrolle sind, es sei denn, es befinden sich gegnerische Einheiten auf der anderen Seite. Jetzt kann es zu einem Gefecht kommen, bei dem die Angriffswerte der Einheiten addiert und diese dann mit den gegnerischen Lebenspunkten verrechnet werden. Sind alle Felder der Mitte durch den Angreifer rechtzeitig vor dem automatischen Schlachtabbruch nach 20 Runden erobert, also kontrolliert, so hat dieser die Schlacht gewonnen und alle Einheiten des Verteidigers werden aus dem Spiel genommen. Das Territorium auf dem die Schlacht stattgefunden hat, kommt nun in den Besitz des Angreifers und die angreifenden Einheiten werden auf dem neu eroberten Territorium platziert.
Während einer Schlacht hat jeder Spieler folgende Spielelemente zu kontrollieren: Kommandopunkte, Einheiten die sofort platziert werden können, Einheiten die

als Verstärkung erst angefordert werden müssen und Aktionskarten, die Auswirkungen auf den Schlachtverlauf haben. Kommandopunkte werden verbraucht um Aktionen auszuführen. Kommandopunkte werden automatisch jede Runde wieder hinzugefügt. Mögliche Aktionen, die Kommandopunkte verbrauchen, sind: weitere Einheiten anfordern, neue Aktionskarten ziehen und Einheiten Lebenspunkte hinzufügen.
Wenn ein Feld der mittleren Spalte sich unter eigener Kontrolle befindet, wird pro Runde für dieses Feld der Kontrollzähler bis maximal fünf um eins inkrementiert. Solange dieser Kontrollzähler größer null ist, ändert sich die Zugehörigkeit nicht, selbst wenn der Gegner in der Zwischenzeit Einheiten in das entsprechende Nachbarfeld positioniert hat. Ist dieses Feld der mittleren Spalte links und rechts davon von Einheiten umgeben, wird der Kontrollzähler pro Runde um eins dekrementiert. Die untere Grenze des Kontrollzählers ist null. Der Kontrollzähler wird sofort auf null gesetzt, wenn die Einheiten, die bis jetzt das Feld der mittleren Spalte kontrolliert haben wegbewegt oder zerstört wurden und gegnerische Einheiten die Kontrolle übernehmen könnten, weil diese direkt daneben stehen. Einheiten in den beiden äußeren Spalten links und rechtes bekommen pro Runde einen Versorgungspunkt.

2.3.1. Ausstattung

Das Spielbrett ist in Staaten unterteilt, welche aus einem Territorium oder mehreren Territorien bestehen. Einheiten können in Territorien stationiert werden. Dabei können die Verteidigungschancen durch den Bau von Forts und Gräben verbessert werden. Jedes Territorium hat ein oder mehr Nachbarterritorien, was es ermöglicht Einheiten von Territorium zu Territorium zu bewegen. Liegt zwischen benachbarten Territorien ein Fluss, so kann dieser bei einem Angriff nur mit der Spielkarte Fluss überquert werden. Weiterhin können von Territorien mit Hafen Einheiten zu anderen Territorien mit Hafen verschifft werden, wenn die notwendige Spielkarte Hafen ausgespielt wird. Zusätzlich zu den Einheiten können auch noch Generäle in Territorien platziert werden, die mit den Einheiten in die Schlacht ziehen um den Schlachtausgang günstig zu beeinflussen. Neue Einheiten erhalten die Spieler jeweils am Rundenanfang. Die Anzahl neuer Einheiten wird unter anderem durch die Anzahl der unter eigener Kontrolle stehenden Territorien

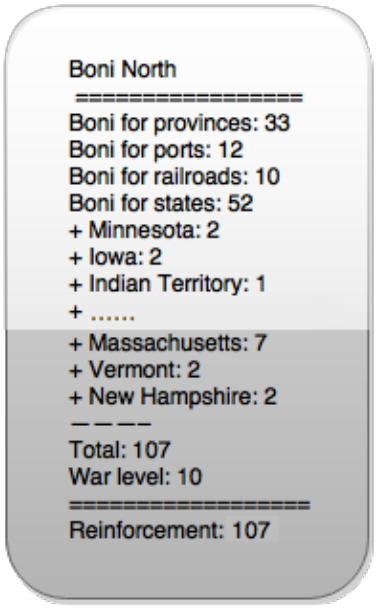

Abbildung 2.3.: Boniauswertung für Landkarte

des jeweiligen Spielers bestimmt. In der globalen Kartenansicht gibt es noch Spielkarten, die von jedem Spieler eingesetzt werden können (z. B. Ausspielen einer Karte um einen neuen General zu bekommen). Einige dieser Spielkarten sind nur in der globalen Kartenansicht spielbar, andere nur in der Ansicht direkt vor einer Schlacht.
Drei wichtige Spielparameter, die jeder Spieler im Auge behalten sollte sind: Moral, Marine und Kriegsanstrengungen. Beim Spielen sollte jeder Spieler möglichst diese globalen Parameter verbessern. Weiterhin ist es möglich im Laufe des Spiels Europa als dritte Partei zum Kriegseintritt zu bewegen, was einem Spiel eine überraschende Wendung geben kann.
Neue Spielkarten erhält man nach Eroberung eines Territoriums oder durch explizites Eintauschen gegen Einheitenverstärkungen. Die Höhe der Moral hat direkten Einfluss auf die Kampfkraft der Einheiten während der Schlacht. Die Marine bestimmt die Anzahl von Einheiten die auf einmal transportiert werden können, wenn die Spielkarten Fluss oder Hafen ausgespielt werden. Ein starke Marine wird auch zu einer Blockade von gegnerischen Häfen führen, was dessen Verstärkungen pro Runde reduzieren wird. Die Kriegsanstrengungen verändern prozentual die möglichen Verstärkungen pro Runde. Es gibt Spielkarten, die die Werte von Moral, Marine und Kriegsanstrengungen verändern (z. B. die Spielkarte Seeschlacht).

2.3.2. Regeln

Im diesem Abschnitt erfolgt eine kurze Zusammenfassung der Spielregeln.

Ziel des Spiels Um ein Spiel zu gewinnen muss ein Spieler alle gegnerischen Territorien erobern.

Spielvorbereitung Das Spiel startet im historischen Szenario des US-amerikanischen Bürgerkriegs im Frühjahr 1861 auf dem nordamerikanischem Kontinent. Per Zufall wird eine festgelegte Anzahl von Einheiten auf die jeweiligen Territorien platziert. Jeder Spieler erhält zu Anfangs fünf Spielkarten und außerdem eine festgelegte Anzahl Einheiten an Verstärkung. Die Spieler starten mit unterschiedlichen Parametern in Moral, Kriegsanstrengungen und Marine unter Beachtung der historischen Vorgaben.

Landkarte In der Standardansicht, der Landkarte, wird die aktuelle Spielrunde mit Datum angezeigt. Hier gibt es die Möglichkeit den eigenen Zug zu beenden um den Gegenspieler spielen zu lassen. Die weiteren Handlungsoptionen des Spielers sind Spielkarte zurückgeben, Spielkarte ziehen, Spielkarte ausspielen, Verstärkung platzieren, automatische Wegfindung für Einheitenbewegungen über viele Territorien hinweg, der Bau von Forts und Gräben sowie Einheiten oder Generäle bewegen. Ist das Ziel einer Bewegung ein gegnerisches Territorium, wird ein Angriff ausgeführt. Angezeigt werden

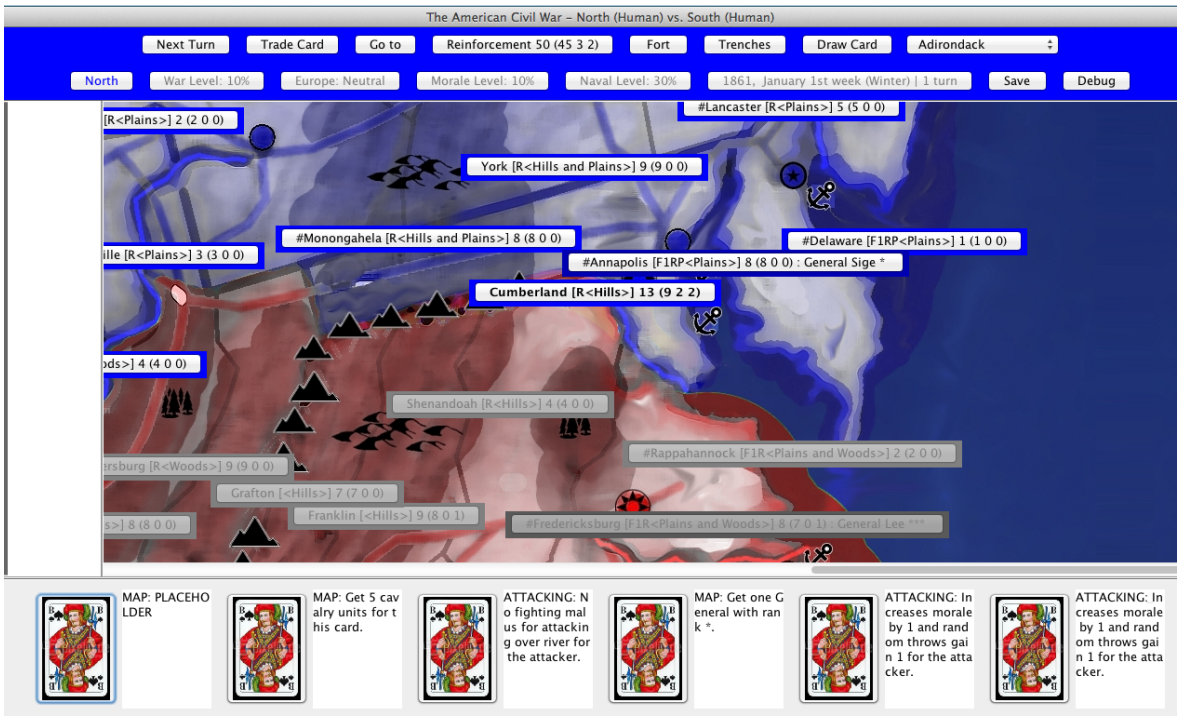

Abbildung 2.4.: Landkarte als Hauptansicht

in dieser Ansicht außerdem die globalen Parameter wie Kriegsanstrengen, Moral und Marine und zusätzlich die Beteiligung Europas am Konflikt. Das ziehen von Spielkarten erfolgt nach dem Zufallsprinzip. Der Nachziehstapel erhält für jede verfügbare Spielkarte eine bestimmte Anzahl davon. Spielkarten werden grundsätzlich abhängig vom aktuellen Spieljahr verfügbar. Die Spielkarten lassen sich in Ereigniskarten, historischen Karten und normalen Spielkarten unterteilen. Nur normale Spielkarten können explizit vom Spieler gezogen werden. Jeder Spielteilnehmer erhält eine Ereigniskarte jeweils am Monatsanfang. Zusätzlich werden historische Karten abhängig von einem historischen Datum gezogen.
Einheiten werden in Infantrie, Kavallerie und Artillerie unterteilt. Generäle werden in Ein-Sternegeneral, Zwei-Sternegeneral und Drei-Sternegeneral unterteilt. Jeder Spieler kann maximal acht Generäle gleichzeitig einsetzen. Territorien haben verschiedene Geländetypen, die die Kampfstärke der Einheiten bei einer Schlacht beeinflussen. Zusätzlich wird die Größe des Schlachtfeldes durch das Vorhandensein von Städten und deren Größe mitbestimmt.

Vor der Schlacht Der Ausgang einer Schlacht kann schon vor der Schlacht beeinflusst werden, indem Spielkarten eingesetzt werden. Es gibt Spielkarten, die nur bei Angriff oder Verteidigung gültig sind. Jeder Spieler kann immer nur eine Spielkarte ausspielen bevor der Gegner am Zug ist. Der Angreifer kann die Schlacht abbrechen oder im zweiten Zug sofort starten.

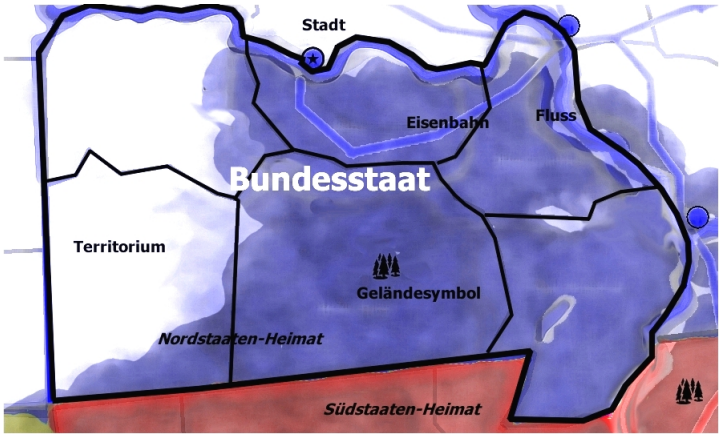

Abbildung 2.5.: Legende der Landkarte

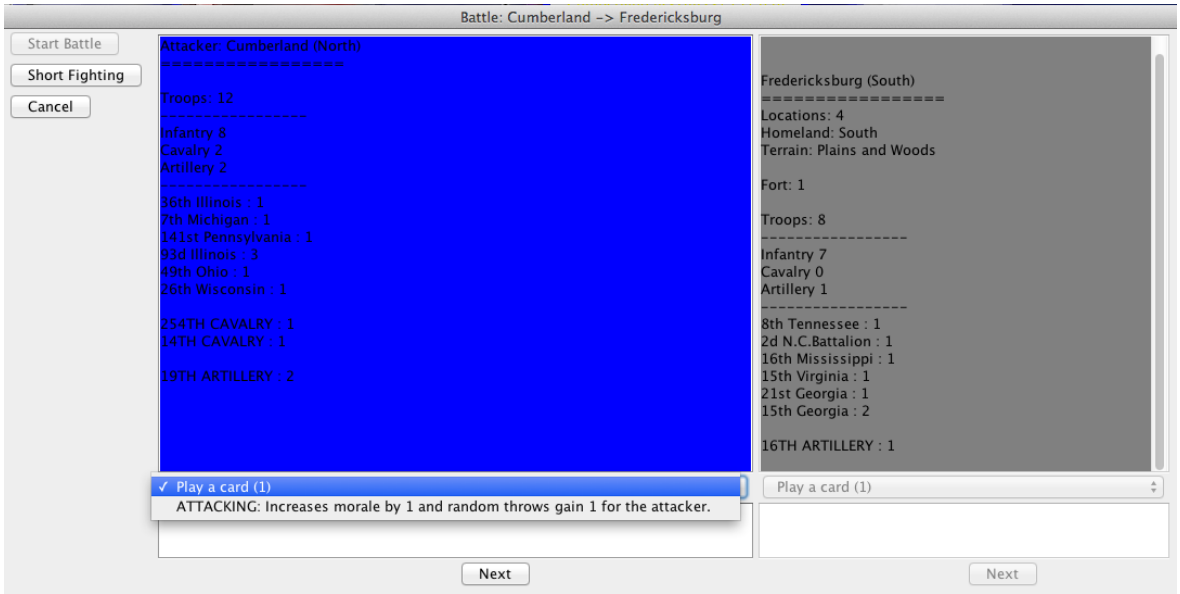

Abbildung 2.6.: Ansicht vor einer Schlacht

Schlacht In der Ansicht für die Schlacht wird u. a. die aktuelle Runde (automatischer Sieg für Verteidiger nach 20 Runden) und die gültigen Boni für die Schlacht angezeigt (z. B. Geländetyp, General, Flussüberquerung etc.). Der aktuelle Spieler hat die Möglichkeit den eigenen Zug zu beenden, um den Gegenspieler spielen zu lassen. Weitere Handlungsoptionen des Spielers sind: Aktionskarte zurückgeben, Aktionskarte ziehen, Aktionskarte ausspielen, Verstärkung platzieren, Lebenspunkte auf platzierte Einheiten verteilen und Einheiten bewegen oder mit diesen angreifen. Ist das Ziel einer Bewegung ein Feld der Mitte des Schlachtfeldes und der Gegner hat Einheiten direkt daneben platziert, so kommt es zu einem Angriff und somit zu einem Gefecht. Wird eine Einheit bewegt muss diese einen Lebenspunkt abgeben. Greift eine Einheit an, so muss diese auch einen Lebenspunkt abgeben. Jede Einheit hat bestimmte Attribute, die

Attacker (North):

Fort -2
Defending -1
Terrain Plains and Woods -1
General +1

Defender (South):

Defending Homeland +1
General +3

South attacking

Fire Fight Value: 2
Boni: 4
Random: 3
Values reduced by morale level (30%)

Total: 2

North defending

5th Vermont 3 # gets one hit
5th Vermont 3 retreated

Abbildung 2.7.: Bonus bei einer Schlacht und Gefechtsauswertung

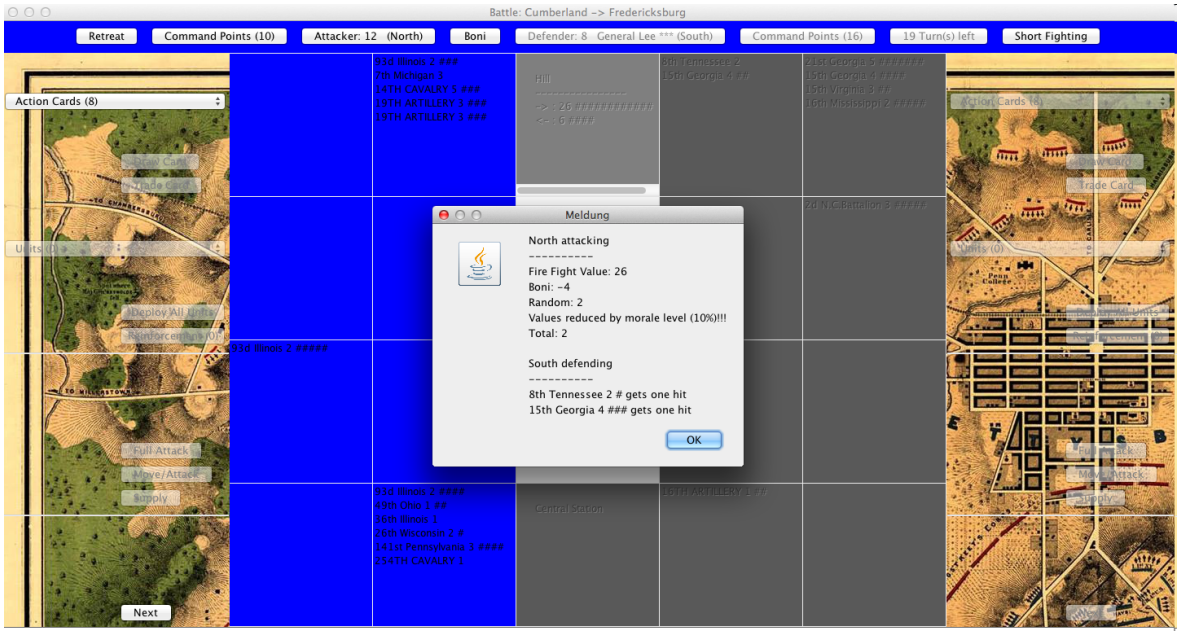

Abbildung 2.8.: Ansicht während einer Schlacht

man sich während des Spiels anzeigen lassen kann. Bei einem Gefecht wird anhand aller beteiligten Einheiten die Angriffsstärke ermittelt, die dann als gedachte Trefferpunkte bei den gegnerischen Einheiten die Lebenspunkte verringert. Hat eine Einheit nicht mehr genug Lebenspunkte um Trefferpunkte zu absorbieren, so wird diese aus dem Spiel genommen oder automatisch (falls möglich) ein Feld zurückgezogen (was zwei Trefferpunkte ungültig werden lässt). Bevor die Angriffsstärke aller beteiligten Einheiten ermittelt wird, können beide Spieler nacheinander Aktionskarten ausspielen um die eigene oder gegnerische Angriffsstärke zu verringern oder zu erhöhen.

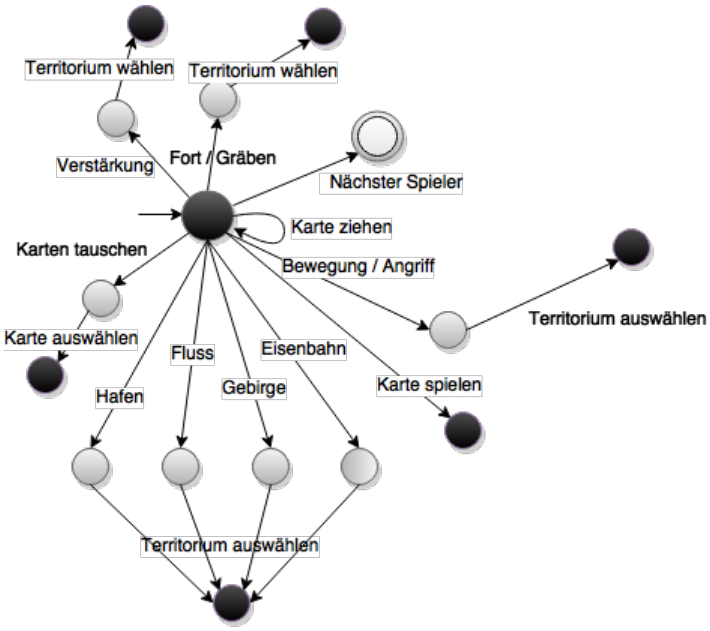

Abbildung 2.9.: Entscheidungsmöglichkeiten in der Ansicht Landkarte: Entscheidungen sind oft mehrmals möglich, wenn ein Spieler an der Reihe ist.

2.3.3. Wahrscheinlichkeiten

Beim Spielen von The American Civil War ist es wichtig zu wissen, mit welcher Wahrscheinlichkeit ein Angriff erfolgreich sein wird und welche Verluste der Angreifer und Verteidiger zu erwarten hat. Dies kann nicht unmittelbar für beide Seiten ausgerechnet werden. Als Grundregel gilt jedoch: Je mehr Einheiten der Angreifer im Verhältnis zum Verteidiger hat, desto größer sind die Chancen auf einen Sieg und damit einhergehend geringere Verluste. Die Kampfstärke ist deterministisch abhängig u. a. vom Geländetyp des angegriffenen Geländes und den eingesetzten Generälen. Durch Ausspielen von Spielkarten kann der Verlauf eines Angriffs schon vor der Schlacht nicht-deterministisch beeinflusst werden. Da einige Aktionen in der Hauptansicht (Bewegung und Angriff) zuerst zwischengespeichert werden und diese erst am Ende der Runde nach einer zufälligen Reihenfolge ausgeführt werden hat jeder Spieler nur unvollständige Informationen über den tatsächlichen Spielzustand. Außerdem wird bei jedem Gefecht zum errechneten Angriffswert ein zufallsbasierter Wert ermittelt, der den Angriffswert erhöht oder verringert. Zusätzlich sind die gegnerischen Spielkarten verdeckt und werden außerdem zufällig gezogen, was wiederum eine genaue Spielanalyse schwierig macht.

2.3.4. Komplexität

Es gibt das Markov-Entscheidungsproblem mit dem sich sequentielle Entscheidungsprobleme in einer voll beobachtbaren Umgebung modellieren lassen [10]. Dabei wird jedes Entscheidungsproblem mit Zuständen und Aktionen beschrieben. Die Aktionen verbinden die Zustände, mit denen man von einem Zustand zum nächsten kommen kann [11]. Diese Herangehensweise wird für die Definition der Komplexität verwendet und in späteren Kapiteln auch für die Suchbäume.

Die Komplexität eines Spieles[2] kann üblicherweise mittels Zustandsraum-Komplexität und Spielbaum-Komplexität gemessen werden. Die Zustandsraum-Komplexität wird von der Anzahl verschiedener Spielzustände, die in einem Spiel möglich sind, bestimmt. Die Spielbaum-Komplexität andererseits, ist die Gesamtzahl der möglichen Spielverläufe. Die Spielbaum-Komplexität ist erheblich größer als der Zustandsraum, da hier dieselben Stellungen in verschiedenen Spielen auftreten, indem Züge in unterschiedlicher Reihenfolge ausgeführt werden.
Jeder Knoten eines Spielbaums entspricht einem Spielzustand. Die Kindknoten eines Elternknotes entsprechen allen Spielständen, die von diesem Knoten aus im aktuellen Spielzug erreicht werden können. Die Komplexität der Entscheidungen kann mit den Entscheidungsgrad ausgedrückt werden. Bei dem Spiel The American Civil War besteht ein Spielzug aus vielen möglichen Entscheidungen (Abbildungen 2.9 und 2.10), wobei die Anzahl der möglichen getroffen Entscheidungen variabel ist und nicht vorherbestimmt werden kann. Jedes Spiel besteht genau aus 202 Territorien und 92 Städte.

Weil es schwierig ist den Zustandsraum in der Ansicht Landkarte zu bestimmen, erfolgt eine Schätzung der oberen Grenze. Dazu werden die globalen Parameter, die Territorien und die Einheiten betrachtet:

- Parameter: Moral × Marine × Kriegsanstrengungen × Spieleranzahl × Europäische Meinung × Kriegseintritt Europa

$$Parameter = 10^3 \times 2 \times 9 \times 3 = 54000$$

- Territorien: Nord/Süd/Neutral × Forts × Gräben

$$Territorien = 3^{202} \times 4^{202} \times 2^{202} = 6,3484 \times 10^{278} \approx 10^{279}$$

- Die maximale Anzahl Einheiten im Spiel wird durch die Anzahl der Städte mal 100 bestimmt.

$$Max = 92 \times 100 = 9200$$

- Einheiten berechnet nach [12]:

$$Einheiten = \sum_{A=202}^{9200} \frac{(A-1)!}{(202-1)! \times (A-202)!}$$

[2]Quelle: https://de.wikipedia.org/wiki/Spiel-Komplexität

$$Einheiten = 1,6379 \times 10^{420} \approx 10^{420}$$

Zur Vereinfachung der Berechnung wird folgendes nicht berücksichtigt:

1. Einheitentypen: Es gibt grundsätzlich 30 verschiedene Einheiten.
2. Generäle: Es gibt maximal acht Generäle pro Spieler, wobei jeder General ein Ein-, Zwei- oder Dreisternegeneral ist.
3. Verstärkung: Maximal 10000 pro Kategorie (Infantrie, Kavallerie oder Artillerie) für jeden Spieler sind erlaubt.
4. Spielkarten: Maximal 100 pro Spieler bei 67 im Spiel verfügbaren verschiedenen Spielkarten sind möglich. Diese 67 Spielkarten sind im Nachziehstapel unterschiedlich häufig verteilt. 264 Karten umfasst der Nachziehstapel.

Es ergibt sich folgende Formel zur Berechnung der Zustandsraum-Komplexität:

- Gesamt: Parameter $\times$ Territorien $\times$ Einheiten

$$Gesamt = 54000 \times 10^{279} \times 10^{240} = 5,4 \times 10^{523} \approx 10^{524}$$

Der Entscheidungsgrad der Ansicht Landkarte wird durch folgende Faktoren beeinflusst:

- Handlungsoption: Spielkarte zurückgeben
- Handlungsoption: Spielkarte ausspielen
- Anzahl der vorhandenen Spielkarten

- Handlungsoption: Bau von Forts und Gräben
- Handlungsoption: Verstärkung platzieren
- Anzahl der verfügbaren Verstärkungen

- Handlungsoption: Einheiten und Generäle bewegen
- Anzahl der verfügbaren Einheiten und Generäle
- Anzahl der eigenen Territorien
- Anzahl der gegnerischen Territorien

Bereits diese kurzen Überlegungen zeigen, dass das Spiel schon nur in der Ansicht Landkarte einen sehr großen Entscheidungsgrad und Komplexität hat. Eine weitere Betrachtung der Komplexität der beiden anderen Ansichten wird deshalb nicht durchgeführt.

Die Zustandsraum-Komplexität ist nicht unendlich, da es für alle Spielelemente Beschränkungen gibt:

Spiel	Zustandsraum	Spielbaum-Komplexität
Schach	10^{46}	10^{123}
Go (19 x 19)	10^{172}	10^{360}
Risiko (1000 Armeen) [12]	10^{47}	10^{2350}
The American Civil War (Landkarte)	10^{524}	

Tabelle 2.1.: Vergleich von Zustandsraum-Komplexität und Spielbaum-Komplexität bei verschiedenen Spielen und The American Civil War. Die Spielbaum-Komplexität für The American Civil War wurde aus Zeitgründen nicht berechnet, ist aber per Definition mindestens so groß wie die Zustandsraum-Komplexität.

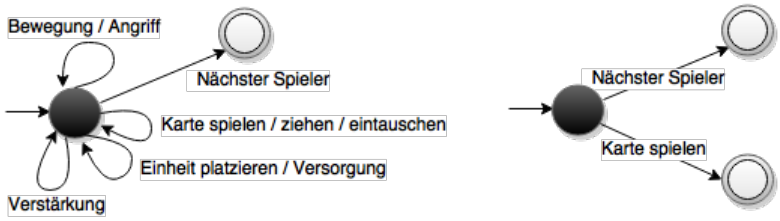

Abbildung 2.10.: Entscheidungsmöglichkeiten während einer Schlacht: Entscheidungen sind oft mehrmals möglich, wenn ein Spieler an der Reihe ist. Links werden die Entscheidungsmöglichkeiten während der normalen Phase gezeigt und rechts, wenn es zu einem Gefecht zwischen Einheiten kommt.

Beschränkungen gibt es in der Ansicht Landkarte bei den Einheiten, den Generälen (acht), den Territorien (202), den Spielkarten (100) und der Rundenanzahl (240). Die maximale Anzahl an Einheiten eines Spielers auf der Landkarte wird bestimmt durch die Anzahl der eigenen Städte mal 100.

Folgende Beschränkungen existieren in der Ansicht Schlacht. Beschränkt sind die Einheitenanzahl durch die Ansicht Landkarte, die Aktionskarten (100 pro Spieler aus 13 verschiedenen Karten) und die Rundenanzahl (20). Die maximale Anzahl der Felder einer Schlacht ist 30 (sechs Zeilen mal fünf Spalten). Die Namen der Felder in der Mitte des Schlachtfeldes werden aus acht verschiedenen Möglichkeiten per Zufall bestimmt.

Es ist offensichtlich anhand der oben genannten Überlegungen zu sehen, dass die in Tabelle 2.1 genannten anderen Spiele eine deutlich kleinere Spielkomplexität haben und dass der sehr große Verzweigungsgrad von The American Civil War es unmöglich macht einen intelligenten Agenten unter Anwendung des traditionellen Minimax-Verfahren zu verwirklichen.

3. Implementierung des Spiels

Als Grundlage für die Untersuchung baumbasierter Suchverfahren wird eine Umgebung gebraucht und dafür wurde das Spiel The American Civil War entwickelt, welches die Spielelemente und Regeln bereitstellt. Dieses Kapitel beschreibt die Vorgehensweise bei der Implementierung, die Architektur, das Design und die wichtigsten Klassen dieser Umgebung.

Als Programmiersprache wurde Java [13] gewählt, da dies die am meisten eingesetzte Programmiersprache für Android-Geräte [14] ist und Android wie eingangs im Vorwort erwähnt die eigentliche Zielplattform neben iOS [15] ist. Eine direkte Entwicklung mittels dem von Google zur Verfügung gestellten Entwicklungstool Android Studio [16] kam nach einem ersten Test nicht in Frage: Eine simple Hallo-Welt-Anwendung brauchte 15 Sekunden auf einem Android Smartphone[1] für den Zyklus Kompilieren, Ausführen und Testen und ist damit um den Faktor 5 langsamer als auf einem Computer[2]. Somit fiel die Entscheidung zuerst nur auf dem PC in der Entwicklungsumgebung Netbeans [17] zu entwickeln um dann später eine Portierung nach Android vorzunehmen.

3.1. Architektur und Design

Durch strikte Trennung nach dem MVC-Muster [18] müssen nur reine grafische Elemente für die jeweilige Zielplattform ausgetauscht werden, nicht jedoch die Logik (siehe Abbildungen 3.1 und 3.2). Das Programm ist in mehrere Pakete strukturiert: Es gibt ein Paket, welches Steuerung und Modelle beinhaltet (`core`). Erstellt wurde ein weiteres für die grafische Repräsentation (`gui`) und für deren Steuerung (`gui_core`). Das Paket für die künstlichen Agenten (`ai`) und für das plattformübergreifende Programmieren (`type`) sind noch zu erwähnen. Insgesamt wurden 95 Klassen erstellt. Bei der Benennung von Variablen und Methoden wurde nach dem Clean-Code Prinzip vorgegangen, was besagt möglichst selbsterklärende Namen zu verwenden, das hat zur Folge, dass Kommentare in der Regel überflüssig sind. Die beim Kodieren verwendete Sprache ist Englisch.

[1] Samsung Galaxy Nexus i9250; auf dem Markt seit Ende 2011
[2] Intel Core i5 mit 1.7 GHz und 4 GB RAM

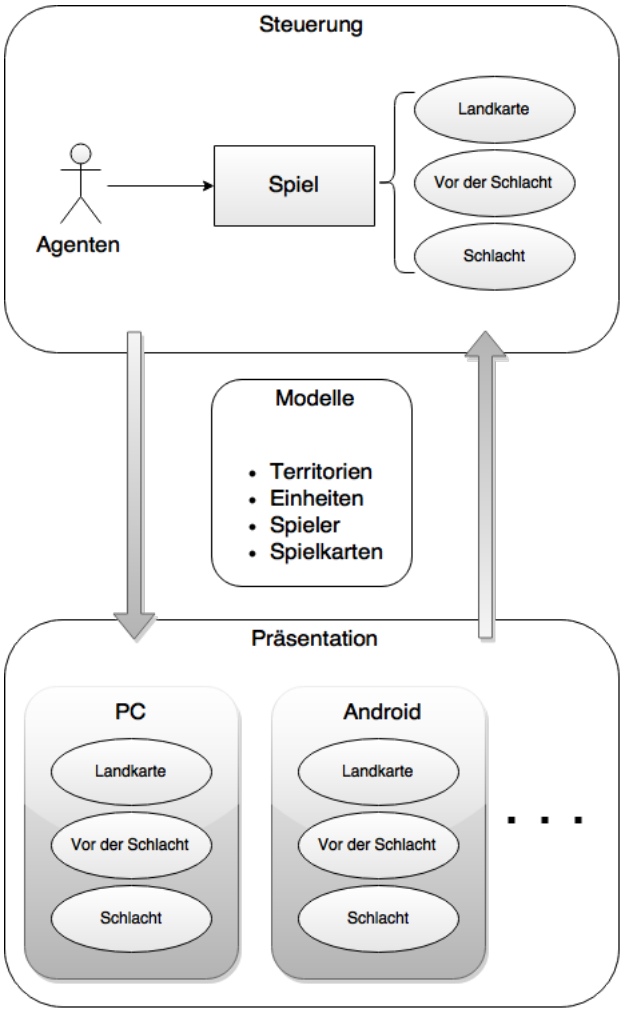

Abbildung 3.1.: Architektur des Spiels The American Civil War

3.2. Klassen

Die Klasse `Game` ist der zentrale Einstiegspunkt. Von dort wird der Programmfluss im Zusammenspiel mit den einzelnen Steuerungsklassen von `gui_core` gesteuert. Außerdem werden dort die Aktionen am Rundenende durchgeführt (Einheitenbewegung etc.) und überprüft ob eine Siegbedingung des Spiels eingetreten ist. Die Klasse `Battle` dient als Einstieg- und Austiegspunkt für die Abarbeitung einer Schlacht. Für die Steuerung werden auch viele Modelle verwendet, wobei grundsätzlich Klassen existieren die zur Verwaltung von Entitäten (teilweise auch deren Steuerung) in Listenform erstellt wurden (z. B. `MainMapCards` enthält alle verfügbaren Spielkarten, die in der Ansicht Landkarte gezogen werden können). Die Klasse `Player` enthält alle Informationen zu einem Spieler bzw. kann diese ermitteln (beispielsweise wie viele Generäle ein Spieler bereits im Spiel einsetzt). Die Territorien, implementiert in den Klassen `Province` und `Provinces`, kennen ihre direkten Nachbarn in Form einer Liste. Es gibt Klassen für die

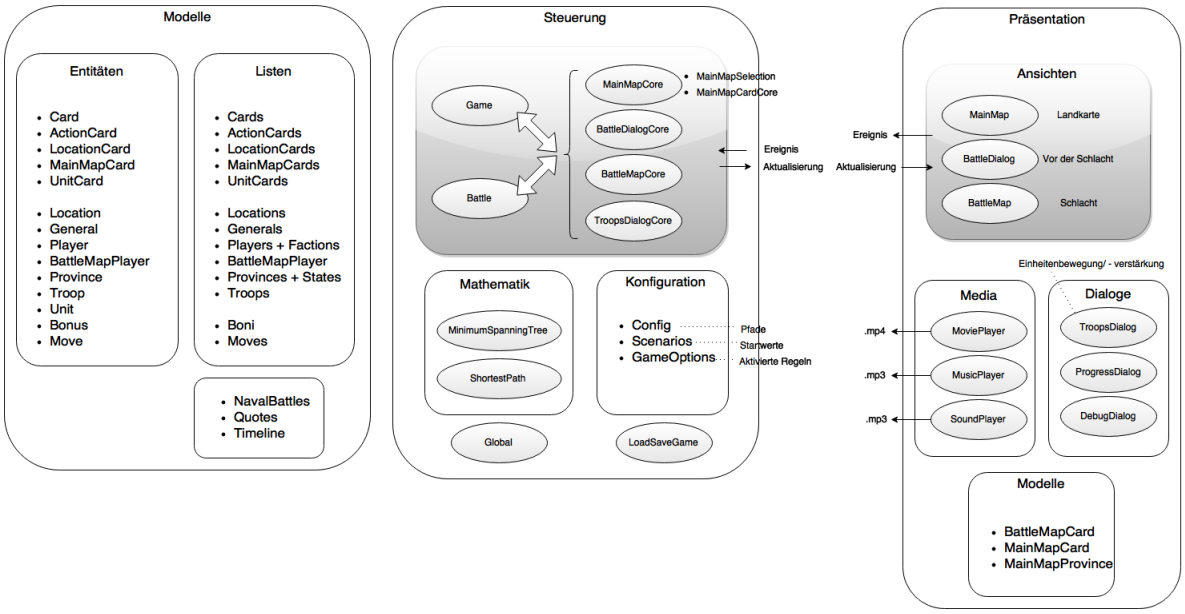

Abbildung 3.2.: Design des Spiels The American Civil War

Konfiguration, in denen unter anderem festgelegt wird, wie das Spiel gestartet werden soll. Getrennt von der restlichen Logik wurden zwei Klassen im Bereich Mathematik angelegt. Diese sind Minimalaufspannender Baum (`MiminumSpanningTree`) und kürzester Weg (`ShortestPath`).

Für die Repräsentation der Landkarte ist die Klasse `MainMap` verantwortlich, die in den jeweiligen Farben die Einheiten und Territorien anzeigt, zusätzlich auf Eingaben des Benutzers reagiert und diese an die Steuerungsklassen von `gui_core` weiterleitet. Die meisten Regeln für die Landkarte befinden sich in der Klasse `MainMapCore`. Die beiden Klassen `BattleDialog` und `BattleMap` sind die grafische Repräsentation jeweils vor und während einer Schlacht. Sie haben einen entsprechenden Gegenpart in `gui_core` für Steuerung und Logik. Für die Repräsentation werden auch einige Modelle verwendet. Für die Agenten wurde das Paket `ai` angelegt, welches dem Spiel über eine Schnittstelle Befehle erteilen kann.

Die Klassen `RandomAgent`, `HeuristicAgent` und `IntelligentAgent` basieren auf der Klasse `Agent`, die vor jedem Spielzug die erlaubten Befehle ermittelt. Zur Analyse eines guten Spielzuges können die Klassen `BattleMapFeatures` und `MainMapFeatures` verwendet werden, die Informationen über den aktuellen Spielzustand aufbereiten und dem heuristischen Agenten helfen, sinnvolle Entscheidungen zu treffen: Beispiele sind Gesamtanzahl der eigenen Einheiten im Verhältnis zum Gegner, Anzahl der eigenen Territorien (oder Staaten) oder wie viele Häfen sich im Besitz eines Spielers befinden.

3.3. Menschlicher Spieler

Der menschliche Spieler kann mittels einer grafischen Oberfläche das Spiel spielen. Der einzige Regelunterschied besteht darin, dass der menschliche Spieler neue Einheiten nur in Heimatterritorien platzieren darf, in denen eine Stadt sich befindet.

3.4. Zufallsbasierter Spieler

Die Klasse `RandomAgent` implementiert einen minimalistischen künstlichen Agenten. Ohne jegliche systematische Entscheidungslogik werden per Zufall die Spielzüge bestimmt. Der Algorithmus ist geradlinig und einfach. Für jede Runde gibt es eine Liste gültiger Spielzüge aus denen per Zufall gewählt wird.

3.5. Erweiterter zufallsbasierter Spieler

Der Nachteil des zufallsbasierten Spielers ist, dass die Wahrscheinlichkeiten für die Aktionen gleichmäßig verteilt sind. Somit gibt es keine Bevorzugung von Bewegung oder Angriff bzw. der Befehl Rückzug und Abbrechen einer Schlacht haben eine zu hohe Wahrscheinlichkeit, sodass sinnvolle Aktionen erst gar nicht ausgeführt werden. Damit ist die Wahrscheinlichkeit in ausreichender Zeit die Siegfelder in der Mitte einer Schlacht rechtzeitig zu erobern deutlich zu niedrig. Dennoch sind sinnvolle Aktionsketten theoretisch möglich.

Darauf hin wurde der zufallsbasierte Agent modifiziert, indem eine Anpassung der Wahrscheinlichkeiten für sinnvolle Befehle und kontextabhängige unsinnige Befehle (wie das Abbrechen einer Schlacht) vorgenommen wurde, was dazu führte das die Leistungsfähigkeit sich verbesserte.

Die Klasse `EnhancedRandomAgent` basiert auf der Klasse `RandomAgent` und erweitert diesen um eine bessere Gewichtung der Wahrscheinlichkeiten.

3.6. Heuristischer Spieler

Außerdem wurde ein heuristischer Agent entwickelt (Klasse `HeuristicAgent`), der Entscheidungen aufgrund von Bewertungen trifft und mittels vorgegebener festgelegter Wenn-Dann-Regeln Aktionen ausführt.

Dieser heuristische Agent ist beiden zufallsbasiertem Agenten überlegen, was im sechsten Kapitel in den Experimenten genauer erläutert wird.

3.7. Intelligenter Spieler

Der intelligente Agent (Klasse `IntelligentAgent`) basiert auf MCTS. Dies wird im fünften Kapitel näher beschrieben. Grundsätzlich deckt dieser Agent nur die Bereiche Bewegung, Angriff und Verstärkung auf der Landkarte und während der Schlacht ab. Anderen Aktionen wie Spielkarte spielen, werden vom heuristischen Agenten unterstützend übernommen (u. a. um den Rechenaufwand niedriger zu halten).

Der intelligente Agent nutzt vorhandene Ressourcen optimaler als der heuristische Agent aus. Einheiten aus Territorien ohne angrenzende feindliche Territorien werden an den Grenzen gruppiert, wo mittels Überlegenheit durch die Masse der eigenen Einheiten der gegnerische heuristische Agent das Nachsehen hat. Eine genaue Analyse zu diesem Aspekt wird in Kapitel 6 durchgeführt.

4. Baumbasierte Suchverfahren

In diesem Kapitel wird erläutert, was „Monte-Carlo Tree Search“genau bedeutet und wie es aufgebaut ist. Die spezifische Implementierung des Verfahrens Upper Confidence Bounds angewandt für Bäume (im Englischen „Upper Confidence Bounds Applied to Trees“, abgekürzt UCT) wird in Abschnitt 4.2.4 beschrieben.

Baumbasierte Suchverfahren basieren, wie der Name schon ausdrückt, auf Bäumen, mit deren Hilfe die Suche durchgeführt wird. Dabei werden die Knoten eines Baumes durchsucht.

Ausgehend von einem Knoten, werden dessen Kindknoten jeweils untersucht und abgespeichert. Neben den Knoten, wird auch noch eine Datenstruktur verwendet, die bestimmt in welcher Reihenfolge der Baum durchsucht wird. Zwei gängige Verfahren sind Breitensuche und Tiefensuche. Bei der erstgenannten wird mit Hilfe der Datenstruktur Warteschlange Ebene für Ebene durchlaufen, indem die Kindknoten in dieser Warteschlange gespeichert werden. Nach den Regeln der eingesetzten Datenstruktur erfolgt dann das Auslesen und Weiterverarbeiten. Analog wird für die Tiefensuche die Datenstruktur Stapelspeicher verwendet, was dazu führt das jeweils bis zu einem Blatt gesucht wird und dann erst beim nächsten Kindknoten weitergesucht wird [19].

4.1. Klassische Suchverfahren

Neben den sehr einfachen Baumsuchverfahren wie Breitensuche und Tiefensuche gibt es noch andere.

Das Minimax-Verfahren wird, wie in der Einleitung bereits erwähnt, sehr erfolgreich beim Spielen von Schach eingesetzt und üblicherweise als optimierte Variante mit dem Namen Alpha-Beta-Suche [1] verwendet. Hier werden Teile des Baums, die zur Lösungsfindung nicht notwendig sind, nicht verarbeitet, was Rechenzeit einspart. Grundsätzlich ist die Voraussetzungen zur sinnvollen Anwendung des Minimax-Verfahrens eine gute heuristische Funktion mit niedrigem Verzweigungsgrad [20].

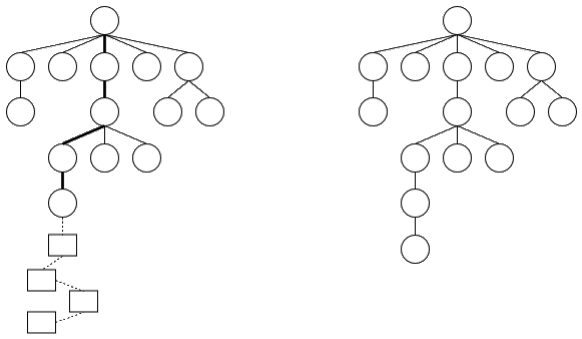

Abbildung 4.1.: Grundlegender Prozess der Monte-Carlo-Baumsuche

4.2. Monte-Carlo-Baumsuche

Die Monte-Carlo-Baumsuche ist eine Kombination aus zwei Verfahren (siehe Abbildung 4.1). Zum einen aus der oben kurz beschriebenen Baumsuche, zum andern aus der Monte-Carlo-Suche, die im folgenden Abschnitt erläutert wird.

4.2.1. Monte-Carlo-Suche

Bei diesem Verfahren wird versucht, Problemstellungen auf numerische Art mit Hilfe der Wahrscheinlichkeitstheorie zu lösen. Dabei ist eine sehr große Zahl gleichartiger Zufallsexperimente (Simulationen) die Basis der Monte-Carlo-Suche. Nach den Gesetzen der großen Zahlen aus der Stochastik werden diese Zufallsexperimente dann bewertet [21].

4.2.2. Kombination zweier Verfahren

Die Monte-Carlo-Baumsuche ist ein baumbasiertes Suchverfahren, was folgendem Prinzip folgt: Erstelle einen Suchbaum mittels Approximation basierend auf den Ergebnissen von Simulationen, die zufällig ausgeführt werden. Es ist somit eine Kombination von Baumsuche und Monte-Carlo-Suche. Der Algorithmus baut nach und nach den Suchbaum anhand bisheriger Simulationswerte auf, wobei die Ergebnisse auch für eine Bestensuche (Wahl des vielversprechendsten Knoten) innerhalb des Baums angewendet werden. Der Baum wird mit jeder Simulation genauer.

4.2.3. Algorithmus

MCTS liefert als Ergebnis, die beste Aktion, die vom aktuellen Zustand ausgeführt werden sollte (siehe Algorithmus 1 am Ende dieses Kapitels).

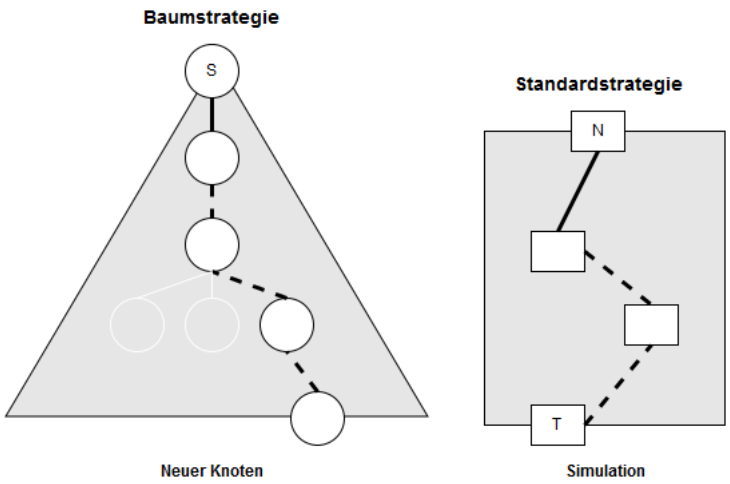

Abbildung 4.2.: Einsatzbereich von Baumstrategie und Standardstrategie. Erstere bestimmt die Wegfindung und Aktionsauswahl vom Ausgangszustand zum Endzustand innerhalb des Baums. Die Standardstrategie legt das Verhalten für die Simulation fest.

Die Funktion UctSearch des Pseudokodes implementiert den Algorithmus wie im folgenden beschrieben:

Der Kern des Verfahrens lässt sich in vier Schritte unterteilen, die möglichst oft wiederholt werden. Diese sind Selektion, Expansion, Simulation und Backpropagation [22]. Dabei wird mittels einer Bedingung vorgegeben, wie lange bzw. wie oft hintereinander diese vier Schritte ausgeführt werden sollen (z. B. durch Angabe der Anzahl der Iterationen oder der maximale Rechenzeit). Wenn die Bedingung zur Wiederholung nicht mehr erfüllt ist, wird die Suche abgebrochen und die beste Aktion der Wurzel des Baums wird zurückgegeben.

Zu Beginn eines neuen Suchdurchlaufs besteht der Suchbaum nur aus der Wurzel. Während jeder Iteration der vier Schritte wird ein neuer Knoten hinzugefügt und eine Simulation ausgeführt. Am Ende aller Iteration hat der Baum somit für jede Iteration mindestens einen Knoten hinzubekommen. Abbildung 4.3 zeigt die Schritte einer Iteration.

Es folgt die Beschreibung der oben genannten vier Schritte:

1) **Selektion**

   Die Aufgabe dieses Schrittes ist es, den optimalen Einstiegspunkt im Baum für die Neuanlage eines Knoten zu finden von dessen Zustand die Simulation dann auch startet:

   Starte mit der Wurzel (Startzustand). Wähle den vielversprechendsten Kindkno-

ten aus. Wende die Aktion an, welche beide Knoten verbindet, um zu diesen Kindknoten zu kommen (neuer Zustand). Wiederhole rekursiv die Vorgehensweise Kindknotenauswahl und Aktionsanwendung jeweils vom neuen aktuellen Knoten aus bis ein Terminalzustand erreicht wurde oder ein Knoten noch nicht alle möglichen Kindknoten erhalten hat, weil noch nicht alle Aktionen berücksichtigt wurden.

Grundsätzlich bestimmt die Baumstrategie (Tree Policy im Englischen) wie Knoten ausgewählt werden. Was der Ausdruck vielversprechendster Kindknoten bedeutet wird im Unterkapitel 4.2.4 beschrieben. Die Funktion TreePolicy des Pseudokodes implementiert diesen Schritt.

2) **Expansion**

In diesem Schritt erfolgt die Neuanlage eines Knoten als Kindknoten, in Bezug zu dem Knoten, der von der Selektion ausgewählt wurde. Die Kante zwischen diesen beiden Knoten ist eine Aktion, die der ausgewählte Knoten noch nicht als Verbindung zu anderen Knoten verwendet (was durch die Selektion erkannt wurde). Wie genau die Knoten expandiert werden, wird durch die Baumstrategie bestimmt, dargestellt in der Abbildung 4.2 (beispielsweise können ein oder mehrere Kindknoten expandiert werden). Implementiert wird dieser Teil des Algorithmus in der Funktion Expand des Pseudokode.

3) **Simulation**

Der neu angelegte Knoten wird als Ausgangspunkt für eine Simulation genommen. Es wird vom aktuellen Ausgangszustand bis zum Ende des Spiels gespielt, wobei für beide Spieler gespielt wird bzw. deren Spielzüge simuliert werden. Mit Hilfe der sogenannten Standardstrategie (Default Policy im Englischen) wird nun eine Aktion ausgehend vom Ausgangspunkt der Simulation ausgewählt und ausgeführt um in einen neuen Zustand zu kommen. Diese Standardstrategie wird bis zum Erreichen eines Terminalzustand zum Auswählen und Anwenden von Aktionen wiederholt benutzt um von Zustand zu Zustand zu kommen (siehe Abbildung 4.2). Beim Erreichen eines Terminalzustand wird das Endergebnis der Simulation ermittelt. Diese ist typischerweise bei einem Spiel für zwei Spieler:

- Sieg = 1
- Unentschieden = 0
- Niederlage = -1

Dieser Teil des Algorithmus ist im Pseudokode in der Funktion DefaultPolicy zu finden.

4) **Backpropagation**

Das Simulationsergebnis wird nun verwendet um den Baum anzupassen. Jeder Knoten speichert die Information, wie oft er besucht wurde und welchen Wert er

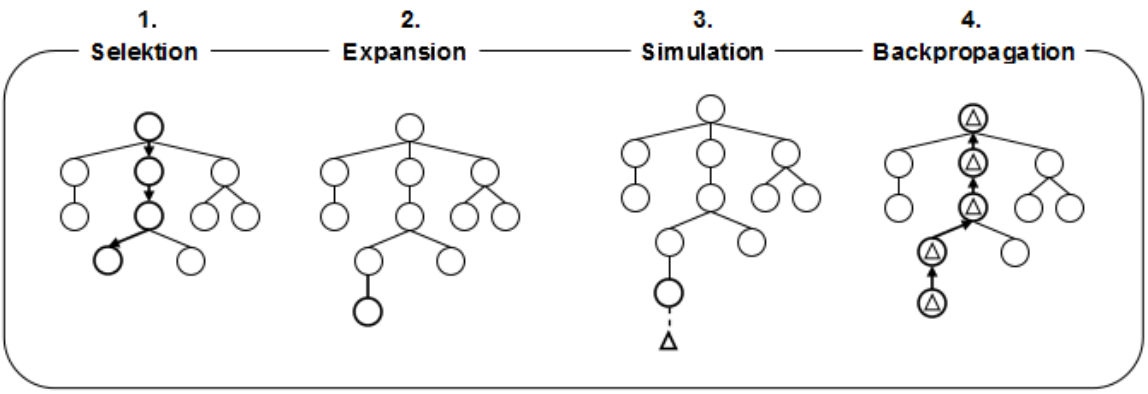

Abbildung 4.3.: Iterationsschritt eines Durchlaufs von MCTS: Jeder Knoten des Suchbaums repräsentiert einen Zustand. Jeder Durchlauf fügt einen Knoten hinzu. Jede Kante ist eine Aktion, die zu einem nächsten Zustand führt.

darstellt. Diese beide Informationen müssen nun bei bestimmten Knoten angepasst werden. Dazu werden nur die Knoten geändert, die in diesem aktuellen Durchlauf besucht wurden. Ausgehend vom neu hinzugefügten Knoten wird der Baum, dem Weg der Selektion rückwärts bis zur Wurzel folgend, durchlaufen. Dabei wird die Anzahl der Besuche inkrementiert sowie das Endergebnis der Simulation zu dem bestehenden Wert des Knotens addiert bzw. subtrahiert.

Die Funktion BACKUP des Pseudokodes implementiert diesen Schritt.

4.2.4. Upper Confidence Bounds angewandt für Bäume

Es gibt verschiedene Unterarten von MCTS. In diesem Unterkapitel wird beschrieben wie der in dieser Arbeit verwendete Algorithmus genau funktioniert. Verwendet wird das Verfahren Upper Confidence Bounds angewandt für Bäume (UCT), welches das Selektionsverhalten der Baumstrategie bestimmt. Diese Variante ist die am häufigsten eingesetzte Implementierung von MCTS mit über 90 % [20]. Dabei ist MCTS die Bezeichnung für den allgemeinen Algorithmus. Die Bezeichnung UCT setzt sich zusammen aus Upper Confidence Bounds (UCB) und MCTS.

Upper Confidence Bounds Bei der Selektion von Knoten wird eine Bewertungsgrundlage benötigt, mit der entschieden werden kann, welcher Knoten der vielversprechendste Kindknoten ist und ausgewählt werden soll. Dabei muss grundsätzlich darauf geachtet werden, dass bereits ausgewertete Knoten nicht immer unbekannten Knoten vorzuziehen sind, denn es könnte eine noch bessere Aktion zu den bereits bekannten Aktionen geben. Dieses Spannungsverhältnis zwischen Verwertung und Neuentdeckung von Knoten wird durch UCB gelöst [23].

UCB1-Formel Für UCB gibt es eine Formel, die folgendermaßen aufgebaut ist:

$$UCB1 = X_j + C\sqrt{\frac{2 \ln n}{n_j}} \tag{4.1}$$

Der linker Summand betont die Belohnung und Suche. Der rechte Summand ermutigt die Entdeckung weniger ausprobierter Knoten und reduziert somit den Effekt von schlechten Ergebnissen bisheriger Simulationen. Dabei ist

- X_j ist die zu erwartende Belohnung der Wahl j,
- n ist die Anzahl wie oft der Elternknoten ausprobiert wurde und
- n_j ist die Anzahl wie oft j ausprobiert wurde.

Der Ausdruck C balanciert zwischen Verwertung und Neuentdeckung. Der Logarithmus wird verwendet um den zu erwartenden Verlust durch schlechte Entscheidungen auszugleichen.

Diese Formel wird in der Funktion BestChild des Pseudokodes angewendet.

Confidence Bounds Confidence Bounds bezieht sich auf das Vertrauen in die Genauigkeit der Belohnung. Generell besteht eine engere Bindung, je häufiger ein Knoten besucht wurde.

Vorteile von UCB1 Die Autoren von „A Survey of Monte Carlo Tree Search Methods“ [10] schreiben, dass UCB1 zwei vielversprechende Eigenschaften hat. So ist der Algorithmus sehr einfach und effizient. Außerdem wird garantiert innerhalb einem konstantem Faktor die bestmögliche Bindung zu finden um dem Dilemma der Auswertung und Neuentdeckung zu entgehen.

4.2.5. Vorteile und Nachteile

In der Literatur werden einige Vor- und Nachteile genannt [10].

Die Vorteile von MCTS sind:

- Kein spezifisches Wissen der Welt ist erforderlich. Es genügt einfach zu wissen, was gültige Züge sind und welcher Zustand terminal ist.
- Dieser Algorithmus ergibt intelligente Spielzüge ohne expliziten Einsatz einer Strategie oder Taktik. Außerdem ist MCTS auch nutzbar bei Szenarien mit verzögernder Belohnung (wie beispielsweise dem Spiel Go).

- Jederzeit und ohne Verzögerung kann dieses Verfahren abgebrochen werden und gibt dennoch ein Suchergebnis zurück.
- Asymmetrisches Wachstum des Suchbaums hilft tiefer in einen Spielbaum vorzudringen. Außerdem hilft es, dass das Wachstum sich mehr auf vielversprechende Bereiche des Suchbaums konzentriert.
- Es konvergiert zu der Lösung des Minimax-Verfahren. Mit unendlich viel Zeit könnte das optimale Ergebnis gefunden werden.
- Der Algorithmus ist einfach zu implementieren.

Folgende Nachteile von MCTS existieren:

- Für einfache Problemstellung funktioniert dieses Verfahren sehr gut (z. B. beim Spiel Tic Tac Toe). Bei komplexen Problemen müssen aber unbedingt Optimierungen vorgenommen werden. Denn ohne Domainwissen werden bei solchen Problemen keine guten Ergebnisse erzielt.
- Der Algorithmus ist ressourcenintensiv, da viel Speicher verbraucht wird, schließlich muss der gesamte Suchbaum im Speicher gehalten werden.
- Die Qualität der Ergebnisse wächst nicht proportional mit der Anzahl der Iterationen.

4.2.6. Verwandte Arbeiten

Es gibt bereits sehr viele Veröffentlichen über MCTS seit der ersten Veröffentlichung 2006. So wurden über 250 Forschungsarbeiten in dieser Zeit veröffentlicht [20] und über 80 verschiedene Variationen und Erweiterungen [10] wurden vorgestellt.

Unter anderem folgende Variationen sind entwickelt worden:

- MCTS für Einzelspieler
- Multi-Agenten für MCTS
- MCTS für Echzeit
- Nicht-deterministisches MCTS

Bekannte Ansatzpunkte für Erweiterungen von MCTS sind u. a.:

- Selektion und Formel für Bestensuche
- Simulation, Backpropagation und Heuristik
- Parallelisierung und maschinelles Lernen
- Spieltheorie

Algorithmus 1 Pseudokode für UCT

```
function UctSearch(s_0)
    create root node v_0 with state s_0
    while within computational budget do
        v_1 ← TreePolicy(v_0)
        Δ ← DefaultPolicy(s(v_1))
        Backup(v_1, Δ)
    return a(BestChild(v_0, 0))

function TreePolicy(v)
    while v is non-terminal do
        if v not fully expanded then
            Expand(v)
        else
            v ← BestChild(v, C_p)
    return v

function Expand(v)
    choose a ∈ untried actions from A(s(v))
    add a new child v' to v
        with s(v') = f(s(v), a)
        and a(v') = a
    return v'

function BestChild(v, c)
    return argmax_{v' ∈ children of v} Q(v')/N(v') + c sqrt(2 ln N(v) / N(v'))

function DefaultPolicy(s)
    while s is non-terminal do
        choose a ∈ A(s) uniformly at random
        s ← f(s, a)
    return reward for state s

function Backup(v, Δ)
    while v is not null do
        N(v) ← N(v) + 1
        Q(v) ← Q(v) + Δ
        Δ ← −Δ
        v ← parent of v
```

5. Implementierung des Suchverfahrens

Untersucht wird in diesem Kapitel wie UCT für diese Arbeit angewendet wurde. Für das Computerspiel The American Civil War wurden für die beiden Ansichten Landkarte und Schlacht jeweils zwei Implementierungen vorgenommen. Um die Leistung zu optimieren wurden verschiedene Optimierungstechniken umgesetzt, die im folgenden erläutert werden.

5.1. Machbarkeitsstudie

Es wurde ein Prototyp vor der eigentlichen Implementierung des UCT-Algorithmus für das Spiel The American Civil War erstellt, mit dessen Hilfe die Umsetzungsmöglichkeit evaluiert wurde. Als Grundlage der Untersuchung wurde das Spiel Tic Tac Toe verwendet (Abbildung 5.1), da es ein sehr einfaches Spiel mit niedriger Komplexität ist. Die obere Grenze des Zustandsraums ist 3^9.

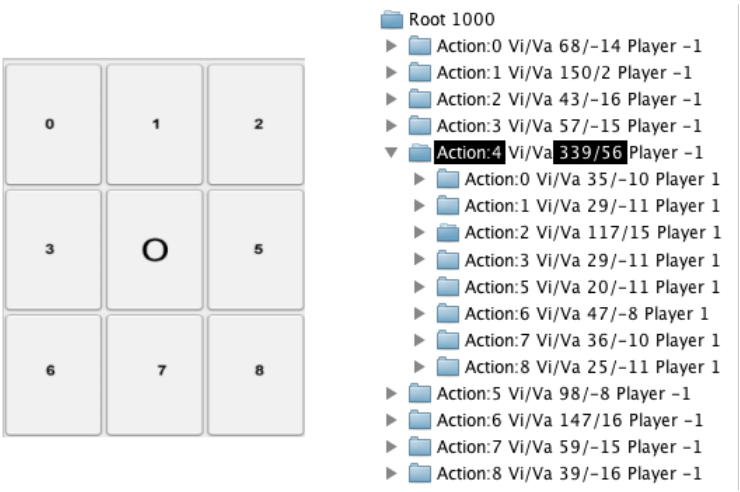

Abbildung 5.1.: Tic Tac Toe als Prototyp: Spieler -1 ist der Computer. Spieler 1 ist der Mensch. UCT wurde angewendet um den besten Spielzug für den Computer zu ermitteln. Auf der rechten Seite ist der Suchbaum abgebildet. Hervorgehoben ist die durch den Computer gespielte Aktion. Man sieht deutlich das asymmetrische Wachsen des Suchbaums. Der ausgewählte Knoten wurde 339 mal besucht und hat den höchsten Wert (56). Für die Suche wurden 1000 Iterationen des UCT-Algorithmus angewendet.

5.2. Geschwindigkeitsoptimierung

Für das Spiel Tic Tac Toe ist die Berechnungszeit des Prototyps bei 1000 Iterationen unter 100 Millisekunden (Intel Core i5 1,7 GHz). Im Vergleich dazu ist es bei komplexen Spielen schwierig die Berechnungszeit niedrig zu halten. Grundsätzlich gilt: je schneller der Algorithmus läuft, desto mehr Simulation sind pro Zeitabschnitt möglich und desto besser wird das Ergebnis sein. Der Algorithmus 3 zeigt die angepasste Standard-UCT-Implementierung.

Folgende Ansatzpunkte wurden realisiert um die Ausführungsgeschwindigkeit zu steigern:

1) **Kopie des Spielstands in Arrays**

   Aus Geschwindigkeitsgründen arbeitet die UTC-Implementierung mit einer eigenen Kopie des Spielstands basierend auf Arrays, da viele tausende mal Kopieroperation mit verschiedenen Spielständen während des Baumaufbaus als auch während den Simulationen vorgenommen werden müssen [3]. Im Anhang A befinden sich Quelltextausschnitte, die dokumentieren wie die Arrays aufgebaut sind um den Spielstand abzuspeichern.

2) **Kopie von Programmteilen zur Optimierung**

Für die Berechnung der Ergebnisse der Simulationen wurden die Routinen und die Spielmechanik der betroffen Spielelemente, die zur Anwendung der Regeln gebraucht werden, kopiert und teilweise vereinfacht (z. B. die Angriffsauswertung mit Boni-Analyse) [3]. Der Pseudokode in Algorithmus 2 zeigt den Aufbau der Simulation mit Angriff, Bewegung und Verstärkung als vereinfachte Kopie.

Algorithmus 2 Simulation in der Ansicht Landkarte

```
 1: function SIMULATE(s, a)
 2:     for each action ∈ actions(a) do
 3:         apply to s
 4:             ← ATTACK(s, action)
 5:             ← MOVEMENT(s, action)
 6:             ← REINFORCEMENT(s, action)
 7:
 8: function ATTACK(s, action)
 9:     apply to s
10:         ← action.Units − HITS(action.To) × action.Units.Count
11:         ← action.To.Units − HITS(action.From) × action.To.Units.Count
12:     if attacker wins then
13:         change owner of action.To to attacker
14:         move remaining troops to action.To
15:
16: function HITS(territory)
17:     boni ← gather boni information for territory // river, mountain crossing etc.
18:     return boni
19:
20: function MOVEMENT(s, action)
21:     apply to s
22:         ← action.From.Units − action.Units
23:         ← action.To.Units + action.Units
24:
25: function REINFORCEMENT(s, action)
26:     apply to s ← action.At.Units + action.Units
```

3) Gruppierung der Aktionen

Um die Komplexität bei der Aktionsauswahl zu reduzieren, werden die möglichen Spielzüge gruppiert. Jeder Knoten im Suchbaum wird durch eine Aktion vom Elternknoten aus erreicht. Hier wurde jeder Aktion des Suchbaums eine Gruppe von Spielzügen zugeordnet (siehe Abbildung 5.2). Das bedeutet die Anzahl der Aktionsgruppen bestimmt die Anzahl der möglichen Spielzüge im Suchbaum (Parameter `A` der UCT-Konfiguration).

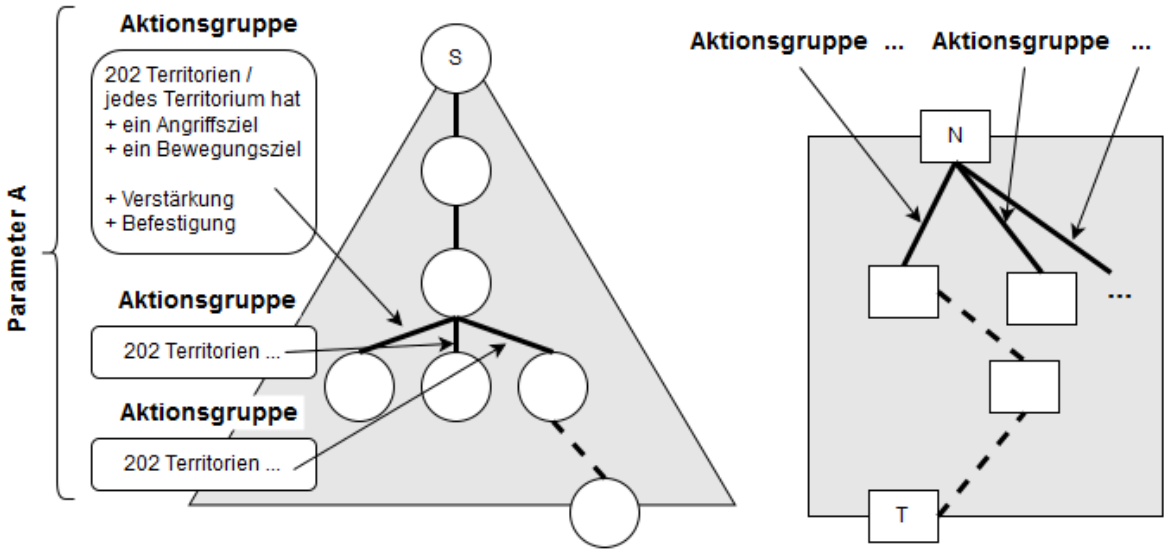

Abbildung 5.2.: Gruppierte Spielzüge für die Landkarte: Wird eine Aktion ausgewählt, werden deren zugehörige Spielzüge ausgeführt.

4) **Beschränkung der Simulationstiefe**

Die Simulation wird nicht beliebig lange durchgespielt bis ein Endzustand des Spiels erreicht wurde (ein Spieler gewonnen hat oder es unentschieden ist). Stattdessen wird nach einer bestimmten Anzahl Runden die Simulation gezielt (abhängig von dem Konfigurationsparameter Simulationstiefe `T`) abgebrochen und somit die Simulationstiefe künstlich verringert um Rechenzeit zu sparen [24].

5) **Heuristik für die Selektion**

Mögliche Spielzüge werden per Heuristik vorgeschlagen, dabei werden ungültige Spielzüge weggelassen und sinnvolle Züge ermittelt. Im ersten Schritt des MCTS-Algorithmus, der Selektion, wird Heuristik angewandt indem Expertenwissen eingesetzt wird. Die Leistungsfähigkeit des MCTS kann durch eine schlechte Heuristik stark geschmälert werden. Beispielsweise könnten gute Spielzüge der Heuristik unbekannt sein und somit dann gar nicht erst ausgeführt werden.

6) **Heuristik für die Simulation**

Auch während der Simulation werden keine zufälligen Spielzüge ausgeführt, sondern wie bei der Selektion mittels Heuristik Spielzüge vorgeschlagen, die dann ausgeführt werden. In dieser Implementierung wird für Selektion und Simulation die gleiche Routine für die Heuristik verwendet. Hierbei muss darauf geachtet werden, dass die Berechnung der Ergebnisse der Heuristik nicht zu lange dauert und somit den Algorithmus nicht ausbremst [25]. Diese Heuristik arbeitet unabhängig vom heuristischen Agenten, da verschiedene Implementierungen eingesetzt werden.

Zeitlich wurde der heuristische Agent zuerst implementiert und arbeitet direkt mit der Spielumgebung ohne Einsatz von Arrays wie beim intelligenten Agenten.

5.3. Konfiguration des Algorithmus

1) **Globale Parameter**

   Es gibt global einstellbare Parameter der UTC-Implementierung:

   - Anzahl der Aktionsgruppen (`A` in Zeile 13 von Algorithmus 5)
   - Anzahl der Iterationen (`I` in Zeile 3 von Algorithmus 3)
   - Maximale Laufzeit in Sekunden (`S` wäre in Zeile 3 von Algorithmus 3)
   - Maximale Anzahl der Runden pro Simulation (`T` in Zeile 28 von Alg. 3)
   - Konstante C der UCB1-Formel (`C` in Zeile 14 von Algorithmus 3)

   Für beide Ansichten wurden unterschiedliche Konstanten gewählt bzw. wird die Anzahl der Aktionsgruppen für die Ansicht Schlacht abhängig von der Einheitenanzahl der akutellen Schlacht bestimmt. Welche Parameter welche Leistung ergeben haben wird in Kapitel 6 erläutert.

2) **Ermittlung der besten Aktion**

   Am Ende des Algorithmus soll die beste Aktion aus dem Suchbaum zurückgegeben werden. Für diese Auswahl gibt es verschiedene Varianten [26]:

   a) Wähle den Kindknoten der Wurzel mit dem höchsten Wert.

   b) Wähle den Kindknoten der Wurzel, der am meisten besucht wurde.

   c) Wähle den Kindknoten der Wurzel, mit der höchsten Besuchszahl und dem höchsten Wert. Wenn keiner existiert, führe die Suche fort bis eine akzeptable Anzahl der Besuche erreicht wurde.

   d) Wähle den Kindknoten der Wurzel, der eine niedrigere Confidence Bound maximiert.

   Für diese Implementierung wurde das Verfahren gewählt, bei dem der Kindknoten der Wurzel mit dem höchsten Wert gewählt wird. Bei Testläufen hat sich gezeigt, dass diese Einstellung am besten zur Implementierung passt (siehe Zeile 7 im Pseudokode des Algorithmus 3).

3) **Ein Kindknoten bei der Expansion**

   Im zweiten Schritt des MCTS-Algorithmus kommt es zur Expansion. In dieser Implementierung wird immer nur ein neuer Kindknoten pro Expansion angelegt.

Algorithmus 3 Angepasste Standard-UCT-Implementierung

```
function UctSearch(s_0, p)
    create root node v_0 with state s_0
    while within iteration count limit (I) do
        v_1 ← TreePolicy(v_0, p)
        Δ ← DefaultPolicy(s(v_1), p)
        Backup(v_1, Δ)
    return a( arg max_{v' ∈ children of v_0} )

function TreePolicy(v, p)
    while v is non-terminal do
        if v not fully expanded then
            Expand(v, p)
        else
            v ← BestChild(v, C_p (C))
    return v

function Expand(v, p)
    choose a ∈ untried actions from ActionGroups(s(v), p)
    add a new child v' to v
        with s(v') = Simulate(s(v), a)
        and a(v') = a
    return v'

function BestChild(v, c)
    return arg max_{v' ∈ children of v} Q(v')/N(v') + c sqrt(2 ln N(v) / N(v'))

function DefaultPolicy(s, p)
    while s is non-terminal and within simulation depth limit (T) do
        choose a ∈ ActionGroups(s, p) uniformly at random
        s ← Simulate(s, a)
    return Reward(s, p)

function Backup(v, Δ)
    while v is not null do
        N(v) ← N(v) + 1
        Q(v) ← Q(v) + Δ
        v ← parent of v
```

5.4. Ansicht Landkarte

Für die Landkarte musste die Struktur der Aktionsgruppe, die Heuristik und die Bewertungsfunktion erstellt werden. Außerdem wird eine Optimierung durch das Einbinden einer Spieleröffnungsstrategie angewendet. Im folgenden werden die oben genannte Punkte erläutert.

5.4.1. Aufbau der Aktionsgruppe

Eine Aktionsgruppe ist ein Array basierend auf der Klasse `MainMapAction`. Es gibt 202 Territorien somit hat dieses Array 202 Einträge. Für jedes Territorium gibt es die Möglichkeit mehrere verschiedene Befehle nebeneinander abzuspeichern. Dabei wird ein Angriffsziel, ein Bewegungsziel, eine Verstärkung und eine Befestigung gleichzeitig pro Territorium erlaubt.

`MainMapAction`: Gespeichert werden nur Integer-Werte, die dann etwa als Index (AttackTo und MoveTo), normaler Zahlenwert oder Boolean (0/1) interpretiert werden. Ein Quelltextausschnitt zu dieser Klasse befindet sich im Anhang im Listing A.1.

5.4.2. Kopie des Spielstands

Für den zu kopierenden Spielstand wurde außerdem folgende Klasse erstellt um die Territorien abzubilden.

`MainMapProvince`: Ein mehrdimensionales Array wird angelegt. Dort werden die Eigenschaften abgelegt z. B. zu welcher Fraktion ein Territorium gehört (Nord/Süd/Neutral). Ein anderes Beispiel ist die Anzahl der stationierten Einheiten. Die maximale Anzahl der Nachbarterritorien eines Territoriums wird auf zehn festgelegt, was für die Abbildung der Territorienverbindungen ausreicht. Für jedes Territorium müssen außerdem die Nachbarn gespeichert werden und ob diese nur über einen Fluss oder Gebirge erreicht werden können. Gespeichert werden nur Integer-Werte, die dann etwa als Index, normaler Zahlenwert oder Boolean (0/1) interpretiert werden. Ein Quelltextausschnitt zu dieser Klasse befindet sich im Anhang im Listing A.2.

5.4.3. Eröffnungsstrategie

Als weitere Optimierung in der Ansicht Landkarte verwendet der intelligente Agent eine Eröffnungsstrategie, die Informationen liefert, welche Territorien erobert und verteidigt werden sollen. Dieses Expertenwissen wird benutzt um mögliche Angriffsziele festzulegen und bestimmt die Priorität bei der Platzierung der Verstärkung zur Verteidigung. Es leitet sich direkt von den Spielregeln ab. Dabei werden sinnvolle Staatenkombinationen

ausgewählt um möglichst viel Boni durch komplett kontrollierte Staaten zu erhalten oder um das Gelände besser auszunutzen, aber auch um kürzere Grenzen zu forcieren.

Der Agent bekommt aus einer vorgegebenen fest einprogrammierten Sammlung ein Array mit den Indizes der zu verteidigenden und anzugreifenden Territorien beim Start des Spiels.

Es gibt mehrere verschiedene Vorschläge als Listen in dieser Sammlung. Hier zwei Beispiele:

- Für den Norden als Angriffsziele sind definiert: Virginia, North Carolina, South Carolina, Georgia und Alabama.
- Für den Süden als Verteidigungsziele sind definiert: Arkansas, Texas, Louisiana, Mississippi und Tennessee.

Diese Ziele bekommen dann eine höhere Priorität bei der Aktionsauswahl für die Angriffsdurchführung. Diese Eröffnungsstrategie wird nur vom intelligenten Agent verwendet. Der heuristische Agent, wählt seine Ziel nur per Zufall aus. Im Pseudokode (Algorithmus 4) werden die beiden dafür eingesetzten Funktionen AttackTerritories und DefendTerritories beschrieben.

Algorithmus 4 Eröffnungsstrategie

```
1:  function AttackTerritories(p)
2:      if p = North then
3:          T = {(Virginia, North Carolina, ...), (Arkansas, Texas, ...), ...}
4:      if p = South then
5:          T = {(Kansas, Missouri, ...), ...}
6:      return choose randomly ∈ T
7:
8:  function DefendTerritories(p)
9:      if p = North then
10:         T = {(Kansas, Missouri, ...), ...}
11:     if p = South then
12:         T = {(Virginia, North Carolina, ...), ...}
13:     return choose randomly ∈ T
```

5.4.4. Heuristik

Die Funktion ActionGroups des Pseudokode in Algorithmus 5 gibt die gültigen Aktionen für Expansion und Simulation zurück und arbeitet nach folgender Vorgehensweise:

1. Die Verstärkung wird aufgeteilt für Angriff, Verteidigung und Befestigung.

Algorithmus 5 Heuristik für der Ansicht Landkarte

```
1:  function ActionGroups(s, p)
2:      distribute reinforcement for
3:          attack ← random value
4:          defense ← random value
5:          fortification ← random value
6:
7:      B ← get border territories controlled by p
8:      T ← get territories controlled by p
9:      A ← AttackTerritories(p)
10:     D ← DefendTerritories(p)
11:
12:     i = 0
13:     for i < count of actiongroups (A) do
14:
15:         HeuristicAttackReinforcement(s, i, B, T, A, attack)
16:         HeuristicAttack(s, i, B, A)
17:
18:         defense = defense + attack
19:         HeuristicDefenseReinforcement(s, i, B, D, defense)
20:
21:         fortification = fortification + defense
22:         HeuristicFortification(s, i, B, fortification)
23:
24:         HeuristicMovement(s, T)
25:     return s
```

2. Ermittle Grenzterritorien und pro Territorium ein feindliches Territorium (wenn vorhanden).

3. Platziere die Verstärkungen, die für den Angriff gedacht sind auf den Grenzterritorien. Wiederhole für alle Grenzterritorien folgendes: Per Zufall wird entschieden, ob es überhaupt Verstärkungen für dieses Territorium geben soll. Ist das der Fall, dann erhält das erste Territorium die Hälfte der möglichen Verstärkungen. Die verfügbaren Verstärkungen für den Angriff werden dann halbiert. Sollte das Territorium einen General haben, werden alle Verstärkungen hier platziert. Wiederhole diesen Vorgang (siehe Zeile 2 im Pseudokode in Algorithmus 6).

4. Platziere die übrig gebliebenen Verstärkungen, die für den Angriff gedacht sind auf den Territorien, die einen Hafen haben, aber nur dann wenn diese Territorien zu wenig Einheiten zur Verteidigung haben (also weniger als die eingestellte Verteidigungsanzahl für Grenzen) und genügend Verstärkungen vorhanden sind. Reduziere den Wert der verfügbaren Verstärkungen entsprechend (siehe Zeile 10

Algorithmus 6 Reinforcement for attack

```
1: function HeuristicAttackReinforcement(s, i, B, T, A, attack)
2:     for each b ∈ B do
3:         if b ∈ A and randomly allowed then
4:             if b has general and randomly allowed then
5:                 s[i] ← add new reinforcement complete attack to b.Units
6:             else
7:                 s[i] ← add new reinforcement attack/2 to b.Units
8:             reduce attack accordingly
9:
10:    for each t ∈ T do
11:        if t has port and t.Units.Count < as supposed then
12:            s[i] ← add new reinforcement attack to b.Units
13:            reduce attack accordingly
14:
```

von Algorithmus 6).

5. Ermittle mögliche Angriffe von den Grenzterritorien aus **anhand der Strategie zur Eröffnung.** Dabei wird berücksichtigt, ob die eigene und gegnerische Moral und Einheitenanzahl multipliziert mit einen bestimmten Faktor einen Angriff als aussichtsreich erscheinen lassen. Nur dann kommt es zu einem Angriff. Eine eventuell notwendige Fluss- oder Gebirgsüberquerung wird nur möglich, wenn die entsprechende Spielkarte vorliegt und diese es ist auch abhängig vom Zufall, ob diese erlaubt wird. Die Anzahl der angreifenden Einheiten wird um die eingestellten Verteidigungsanzahl für Grenzen reduziert (siehe Zeile 2 von Algorithmus 7).

Algorithmus 7 Attack

```
1: function HeuristicAttack(s, i, B, A)
2:     for each b ∈ B do
3:         if b ∈ A and attack possible then
4:             if strength of b > randomly choosen enemy territory's strength then
5:                 s[i] ← add new attack
6:
7:     for each b ∈ B do
8:         if attack possible then
9:             if strength of b > randomly choosen enemy territory's strength then
10:                s[i] ← add new attack
```

6. Ermittle mögliche Angriffe von den Grenzterritorien aus **unabhängig** der Strategie zur Eröffnung. Dabei wird berücksichtigt, ob die eigene und gegnerische Moral multipliziert mit der Einheitenanzahl um einen bestimmten Faktor einen Angriff

als aussichtsreich erscheinen lassen. Nur dann wird ein Angriff gestartet. Eine eventuell notwendige Fluss- oder Gebirgsüberquerung wird nur möglich, wenn die entsprechende Spielkarte vorliegt und diese ist auch abhängig vom Zufall. Entschieden wird auch per Zufall, ob die Anzahl der angreifenden Einheiten um die eingestellten Verteidigungsanzahl für Grenzen reduziert wird (siehe Zeile 7 von Algorithmus 7).

Algorithmus 8 Defense

```
1:  function HeuristicDefenseReinforcement(s, i, B, D, defense)
2:      distribute defense according to count of B
3:
4:      for each b ∈ B do
5:          if b ∉ D then
6:              if randomly allowed then
7:                  s ← add new reinforcement
8:              else
9:                  if b.Units.Count = 0 then
10:                     s[i] ← add new reinforcement
11:             reduce defense accordingly
12:
13:     for each b ∈ B do
14:         if b ∈ D then
15:             s[i] ← add new reinforcement
16:             reduce defense accordingly
```

7. Eventuell übrig gebliebene Verstärkung für den Angriff wird jetzt der Verteidigung zugeordnet. Es wird im folgenden versucht, diesen Wert gleichmäßig über die einzelnen Grenzterritorien zu verteilen.

Algorithmus 9 Fortification

```
1:  function HeuristicFortification(s, i, B, fortification)
2:      choose randomly what to build
3:
4:      for each b ∈ B do
5:          if allow randomly then
6:              s[i] ← add new fortification
7:              reduce fortification accordingly
```

8. Ermittle für die Grenzterritorien **unabhängig** der Eröffnungsstrategie die notwendige Verstärkungen zur Verteidigung. Entschieden wird für jedes Grenzterritorium per Zufall, ob Einheiten stationiert werden. Der Verstärkungswert wird entsprechend reduziert. Sollte per Zufall dagegen entschieden werden, wird geprüft, ob

überhaupt keine Einheiten in diesem Grenzterritorium stationiert sind. Ist das der Fall, werden Einheiten stationiert und der Verstärkungswert wird entsprechend reduziert (siehe Zeile 4 von Algorithmus 8).

9. Ermittle für die Grenzterritorien **anhand der Eröffnungsstrategie** die notwendige Verstärkungen zur Verteidigung. Wenn Einheiten verfügbar sind, werden diese stationiert. Der Verstärkungswert wird entsprechend reduziert(siehe Zeile 13 von Algorithmus 8).

Algorithmus 10 Movement

```
function HeuristicMovement(s, T)

    for each t ∈ T do
        if b.Units.Count ≥ as supposed then
            if t ∈ B then
                look for general in neighbourhood for target selection
            else
                choose target randomly
            if movement possible then
                s[i] ← add new movement
```

10. Eventuell übrig gebliebene Verstärkung für die Verteidigung wird jetzt der Befestigung zugeordnet.

11. Per Zufall wird entschieden, ob Forts oder Gräben gebaut werden sollen. Jetzt wird die Liste der Grenzterritorien durchgegangen und per Zufall bestimmt, ob gebaut wird oder nicht (solange genug Verstärkung vorhanden ist). Der Verstärkungswert wird jeweils entsprechend reduziert (siehe Algorithmus 9).

12. Jetzt wird die Liste der Territorien durchlaufen und überprüft, ob in jedem Territorium genug oder zu viel Einheiten stationiert sind. Bei Grenzterritorien wird überprüft, ob sich in einem unmittelbaren benachbarten Grenzterritorium ein General befindet. Ist das der Fall, werden Einheiten dorthin versetzt. Bei normalen Territorien wird per Zufall entschieden, welches Territorium als Ziel ausgewählt wird und ob Einheiten versetzt werden sollen (siehe Algorithmus 10).

5.4.5. Spielbewertung

Grundsätzlich ist darauf zu achten, dass die Berechnung der Spielbewertung möglichst schnell erfolgt.

Die Funktion Reward des Pseudokode in Algorithmus 11 gibt die Spielbewertung für einen Spielstand für einen Spieler zurück und wird herangezogen um am Ende einer

Algorithmus 11 Spielbewertung in der Ansicht Landkarte

```
1:  function REWARD(s, p)
2:      if Turn > 12 and Turn mod 5 = 0 then
3:          return count of territories controlled by p in s
4:      else
5:          T ← territories controlled by p in s
6:          for each t ∈ T do
7:              if t is part of border then
8:                  if t.Units.Count < as supposed then
9:                      a ← −10
10:                 b ← t.GeneralRank × t.Units.Count
11:                 c ← t.Units.Count
12:             else
13:                 d ← t.GeneralRank × (t.Units.Count − as supposed)
14:             if t has port then
15:                 if t.Units.Count < as supposed then
16:                     e ← −10
17:             else
18:                 if t.Units.Count − as supposed > 0 then
19:                     f ← t.Units.Count − as supposed
20:             r ← (r + 1000 × T.Count + a + b + c + d + e + f)
21:         return r
```

Simulation deren Endzustand zu bewerten. Die Spielbewertung ergibt die Anzahl der Punkte im Spiel. Diese Funktion ist zweigeteilt:

1. Bewertet wird die Anzahl der eigenen Provinzen. Damit werden Angriffe bzw. Aktionen forciert, die zur Eroberung neuer Territorien führen. Diese Bewertung tritt nur alle fünf Spielrunden auf und ist erst nach 12 Spielrunden aktiv, da vorher die Einheiten erst an den Grenzen konzentriert werden müssen und ein Angriff weniger aussichtsreich wäre.

2. Bewertet werden die Grenzterritorien. Abzüge geben Territorien, deren Einheitenanzahl unter der eingestellten Verteidigungsanzahl für Grenzen steht. Generäle und die Anzahl der Einheiten an der Grenze erhöhen den Wert. Generäle außerhalb Grenzterritorien verringern die Bewertung. Territorien die nicht zur Grenze dazu gezählt werden und keinen Hafen haben, verringern den Wert um den Anzahl ihrer stationierten Einheiten. Territorien mit Hafen, deren Einheitenanzahl unter der eingestellten Verteidigungsanzahl für Grenzen ist, verringern den Wert. Die Gesamtanzahl der eigenen Provinzen erhöht den Wert.

Diese Bewertungen haben im Zusammenspiel der Simulationen zur Folge, dass

- Generäle eher an der Grenze stationiert sind und sich an Angriffen beteiligen,
- Häfen nicht ungeschützt sind,
- Territorien ohne feindliche Nachbarfelder ihre Truppen an die Grenze schicken
- und es zu einer Truppenkonzentration an der Grenze kommt.

5.5. Ansicht Schlacht

Aktionsgruppen werden auch für die Ansicht der Schlacht eingesetzt. Der Aufbau der Aktionsgruppe unterscheidet sich aber zu der bereits vorgestellten. Zusätzlich wurde auch hier eine Funktion für die Heuristik und eine Bewertungsfunktion erstellt. Im folgenden werden diese Punkte erläutert.

5.5.1. Aufbau der Aktionsgruppe

Für die Ansicht Schlacht basiert die Aktionsgruppe auf der Klasse `BattleMapAction`. Jeder Eintrag speichert immer nur einen Befehl ab. Mögliche Befehle sind Bewegung, Angriff, Einheit platzieren, Verstärkung und Versorgung.

`BattleMapAction`: Gespeichert werden nur Integer-Werte, die dann etwa als Index (MoveFrom, AttackFrom, To), normaler Zahlenwert oder Hashcodes (Einheitenobjekte) interpretiert werden. Ein Quelltextausschnitt zu dieser Klasse befindet sich im Anhang im Listing A.3.

5.5.2. Kopie des Spielstands

Gleich zwei Klassen werden benötigt um eine Kopie des Spielstands zu speichern.

`BattleMapLocation`: Gespeichert werden nur Integer-Werte, die dann etwa als Index (MoveFrom, AttackFrom, To), normaler Zahlenwert oder Hashcodes (Einheitenobjekte) interpretiert werden. Neben der eigentlichen Lokation werden, die dazugehörigen Einheiten mit im Array abgespeichert. Ein Quelltextausschnitt zu dieser Klasse befindet sich im Anhang im Listing A.4.

`BattleMapPlayer`: Es werden nur Integer-Werte für die jeweiligen Werte wie z. B. Kommandopunkte oder Boni gespeichert.

5.5.3. Heuristik

Die Heuristikfunktion für die Schlacht gibt die gültigen Aktionen für Selektion und Simulation zurück.

Beschreibung der Heuristik aus Sicht des aktiven Spielers:

1. Die Kommandopunkte werden aufgeteilt für Verstärkung, Kartenziehen und Versorgung. Die Aufteilung erfolgt per Zufall, ist aber in den ersten zwei Runden vorbestimmt.
2. Solange noch der Wert für Verstärkung ausreichend ist, fordere Verstärkung an und reduziere entsprechend die Kommandopunkte.
3. Platziere alle verfügbaren Einheiten auf dem Schlachtfeld. Die Reihe wird zufällig bestimmt. Die Spalte ist beim Angreifer immer die erste, beim Verteidiger per Zufall.
4. Versorgungsphase: Gib allen Einheiten, die direkt neben der mittleren Spalte platziert sind einen Versorgungspunkt, wenn diese keinen mehr besitzen, unter der Bedingung, dass ausreichend Kommandopunkte vorhanden sind.
5. Bearbeite die Liste aller platzierten Einheiten und entscheide per Zufall, ob Versorgungspunkte vergeben werden sollen. Kommandopunkte werden entsprechend reduziert.
6. Bewegungsphase: Betrachte die Seite des Gegners. Ermittle das Feld mit der maximalen Anzahl an Einheiten direkt neben der mittleren Spalte. Wurde nichts gefunden, suche in der nächsten Spalte und speichere die Reihe ab (Brennpunkt). Betrachte nun alle Einheiten. Wenn eine Bewegung möglich ist, dann bewege die Einheiten in Richtung Brennpunkt oder ein beliebiges Feld (welche der beiden Optionen gewählt wird, wird zufällig getroffen). Aber lasse mindestens eine Einheit zurück, wenn das aktuelle Feld an die mittlere Spalte grenzt.
7. Angriffsphase: Betrachte nun alle Einheiten. Wenn ein Angriff möglich ist, dann führe ihn aus.

5.5.4. Spielbewertung

Wie bei der Ansicht Landkarte ist darauf zu achten, dass die Berechnung der Spielbewertung für die Schlacht möglichst schnell erfolgt.

Auch für die Schlacht gibt es eine Bewertungsfunktion, die herangezogen wird, um am Ende einer Simulation deren Endzustand zu bewerten. Sie gibt den Wert als Spielendstand zurück (Sieg, Niederlage oder Unentschieden).

Bewertet wird die Anzahl der kontrollierten Felder in der Mitte des Schlachtfeldes. Kontrolliert ein Spieler alle Felder wird Sieg oder Niederlage (abhängig vom aktuellen Spieler) zurückgeliefert, ansonsten ist das Ergebnis unentschieden.

5.6. Aufruf der UCT-Implementierung

Die Prozedur UctAgent des Pseudokode in Algorithmus 12 ruft den UCT-Algorithmus auf. Dabei wird vor dem Aufruf eine Kopie des aktuellen Spielstands in Arrays abgespeichert. Dann erfolgt der Aufruf der Funktion UctSearch. Dabei wird der Spielzustand als Array und der aktuelle Spieler übergeben. Der Rückgabewert der Funktion ist die auszuspielende Aktion und somit die bestmögliche Aktion, die durch UCT ermittelt wurde. Diese wird nun angewendet und die einzelnen Spielzüge die der Aktion zugeordnet sind, ausgeführt. Der Aufbau der Funktion Funktion UctSearch im Pseudokode in Algorithmus 3 deckt sich in großen Teilen mit der Beschreibung aus dem vierten Kapitel und dessen Pseudokode. In dieser Implementierung werden noch die Funktionen ActionGroups, Simulate und Reward aufgerufen, die bereits in diesem Kapitel beschrieben wurden.

Algorithmus 12 Aufruf der UCT-Implementierung

```
1: procedure UctAgent
2:     arrays ← get copy of game state
3:     action ← UctSearch(arrays, current player)
4:     execute actions of action on game
```

5.7. Leistungssteigerung durch MCTS

Die Vorschläge für die Spielzüge erfolgt durch die Heuristik und beeinflusst damit unmittelbar die gewählten Spielzüge. Die Leistungssteigerung des intelligenten Agenten besteht darin, dass durch die Simulationen möglichst gute Kombinationen der Ergebnisse der Heuristik gespielt werden. Der heuristische Agent verfügt nicht über die Eröffnungsstrategie, was damit dem intelligenten Agenten einen kleinen Vorteil verschafft. Weiterhin wurde aus zeitlichen Gründen keine Anpassung der Implementierung des heuristischen Agent vorgenommen bzw. wurde kein verbesserter heuristischer Agent erstellt. Wichtig wäre jedoch, eine möglichst gleiche Leistungsfähigkeit der Heuristik für die Agenten zu erzielen, um eine bessere Vergleichsmöglichkeit zu haben. Unabhängig davon ist zu erwarten, dass bei gleicher Heuristik für beide Agenten, der intelligente Agent immer noch deutlich besser im direkten Vergleich abschneiden wird.

Dem heuristischen Agenten werden pro Grenzterritorium nur ein Angriffsziel per Zufall für jede Runde angeboten. Damit ist nicht garantiert, dass immer optimal gewählt wird

und was damit einen Nachteil dieser Implementierung darstellt. Im Vergleich dazu sind die Chancen auf eine optimales Angriffsziel für den intelligenten Agenten allein schon durch die Simulation erheblich größer, da zwar auch hier per Zufall ausgewählt wird, aber diese Zielsuche jeweils mehrmals pro Spielrunde stattfindet.

6. Experimente

Die Auswertungen und Experimente, die durchgeführt wurden um die Leistungsfähigkeit der einzelnen Agenten einordnen zu können, werden in diesem Kapitel vorgestellt. Verglichen wird insbesondere der UCT-basierte intelligente Agent mit den drei bereits vorgestellten Agenten.

Beim ersten, dem zufallsbasiertem Agenten, werden alle Entscheidungen per Zufall getroffen. Handlungsoptionen sind die jeweilig gültigen Befehle für den aktuellen Zustand. Die Wahrscheinlichkeit, welche Aktion ausgeführt wird, ist gleichmäßig verteilt. Dieser Agent wird als Grundlage für alle weiteren Vergleiche herangezogen.

Der zweite Agent ist der erweiterte zufallsbasierte Agent. Dieser entscheidet per Zufall welche Aktion ausgeführt werden soll, wobei die Gewichtung der Aktionen angepasst wurde, sodass sinnvolle Aktionen häufiger ausgeführt werden.

Schließlich gibt es noch den heuristischen Agenten. Bei diesem wurden festgelegte Regeln benutzt, die dem Agenten helfen sinnvolle und gute Spielzüge zu spielen. Zusätzlich wird der aktuelle Spielstand analysiert und zur Entscheidungsfindung herangezogen.

6.1. Versuchsaufbau

Folgender Versuchsaufbau wurde für alle Experimente vorgenommen:

Ausgewertet wird die Leistung der Agenten anhand der gewonnen Spiele und Spielpunkte, die am Ende eines Spiels erreicht wurden. Es wurden für alle Spielpaarungen Spiele mehrfach ausgetragen.

Ein Spiel dauert 240 Runden von Anfang 1861 bis Anfang 1866. In dieser Zeit muss ein Spieler alle gegnerischen Territorien erobern um das Spiel zu gewinnen (neutrale Territorien zählen nicht als gegnerische Territorien). Danach kommt es zu einem Unentschieden und die Spielpunkte werden gewertet. Die Punktewertung wird mit der folgenden Formel aus der Sicht des jeweiligen Spielers berechnet:

$$Punkte = \text{Anzahl Territorien} + \frac{\text{Anzahl Einheiten}}{100} + \text{Staaten-Boni} \tag{6.1}$$

Nordstaaten	Südstaaten	Spiele	Runden	σ	Sieg Intelligent in %
Intelligent	Heuristisch	25	38	26,47	100
Heuristisch	Intelligent	25	46	6,7	100
Intelligent	Zufallsbasiert	15	31	6,24	100
Zufallsbasiert	Intelligent	15	32	4,45	100
Intelligent	Erweitert	15	56	16,44	100
Erweitert	Intelligent	15	46	7,23	100

Tabelle 6.1.: Der intelligente Agent spielt gegen die drei nicht intelligenten Agenten jeweils als Nord- und Südstaaten. Vergleich bei unterschiedlich vielen unabhängigen Spielen. Die Ergebnisse sind Durchschnittswerte.

Einige Regeln wurden für den Versuchsaufbau geändert um den Einfluss von Zufallselementen des Spiels zu reduzieren:

- Die globalen Spielparameter Moral, Marine und Kriegsanstrengungen wurden auf den höchsten Wert 10 für alle Spieler festgelegt.
- Alle historischen Spielkarten, die Spielkarten Emanzipation, Conscription und alle Europakarten wurden deaktiviert.
- Weiterhin wurden die Seeschlachten, die Blockade und alle Karten, die Verstärkungen vor einer Schlacht ermöglichen, herausgenommen.
- Alle anderen Spielkarten waren noch aktiviert.

6.2. Ergebnisse

In den Experimenten hat sich gezeigt, dass der erweiterte zufallsbasierte Spieler stärker spielt als der zufallsbasierte Agent. Im Vergleich mit dem heuristischen Agenten hat dieser aber schlechter gespielt. Der heuristische Spieler ist unter den nicht intelligenten Agenten der stärkste Spieler. Allen drei anderen Spielern überlegen ist der UCT-basierte intelligente Agent bei weitem. Aus Zeitgründen gab es nur sehr wenige Experimente für Variation der globalen Parameter von UCT, sodass eine genaue Aussage darüber noch nicht getroffen werden kann. Die Experimente haben außerdem gezeigt, dass die Ausgangssituation, ob als Nordstaaten oder als Südstaaten gespielt wird, für die Erfolgsaussichten beim Spiel irrelevant ist.

Tabelle 6.1 zeigt, dass der intelligente Agent gegen die anderen Agenten bei allen durchgeführten Spielen gewonnen hat. Bei einigen Paarungen wurden weniger Spiele ausgetragen um Rechenzeit einzusparen. Die nicht-heuristischen Agenten werden im

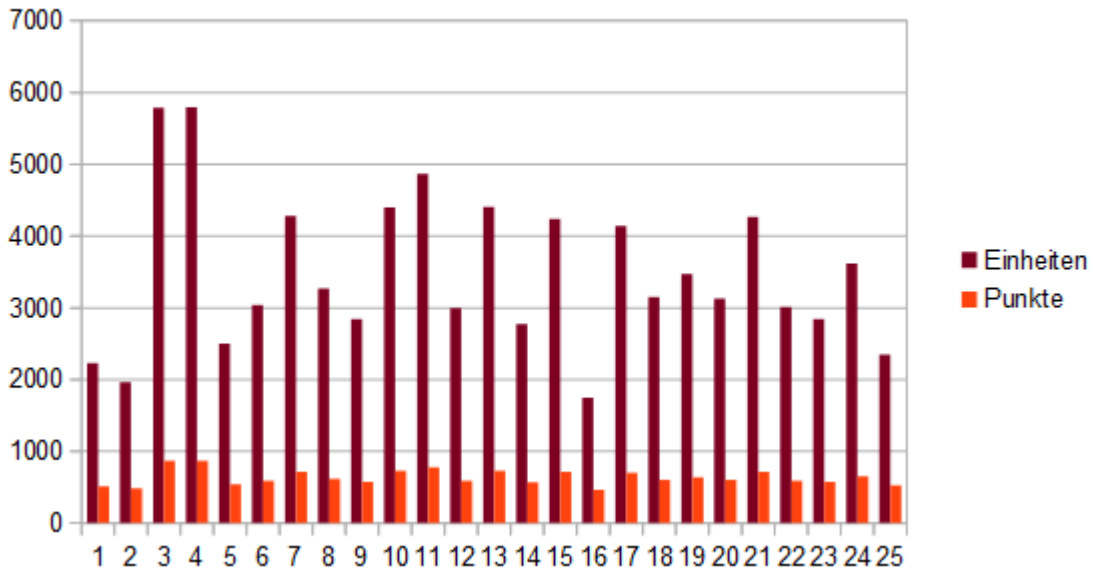

Abbildung 6.1.: Endergebnis der Spielpaarung Heuristisch (Nord) vs. Intelligent (Süd): Verteilung von Einheiten und Punkten des intelligenten Spielers bei 25 Spielen. Der Gegner hat in jedem Spiel null Punkte und null Einheiten.

Durchschnitt recht früh und spätestens innerhalb von 25 % der gesamten Spielzeit besiegt. Zu sehen ist eine große Standardabweichung bei der Paarung Intelligent gegen Heuristisch im Vergleich zu deren Gegenpart (Heuristisch gegen Intelligent), was darauf zurück schließen lässt, dass der Spielverlauf beim Gegenpart gleichmäßiger war. Bemerkenswert ist, dass der erweiterte zufallsbasierte Agent durchschnittlich länger die Niederlage hinauszögern konnte. Direkt abhängig von der Spieldauer sind die verfügbaren Einheiten am Ende des Spiels, was aus den Abbildungen 6.1 und 6.2 hervorgeht: Je länger ein Spiel dauert, um so mehr Einheiten hat ein Sieger am Ende. Den Spielverlauf eines Spiels mit einem intelligenten Agenten zeigen die Abbildungen 6.3 bis 6.5. Dort wird die kontinuierliche Verbesserung der Position des intelligenten Agenten innerhalb dieses Spiels sichtbar.

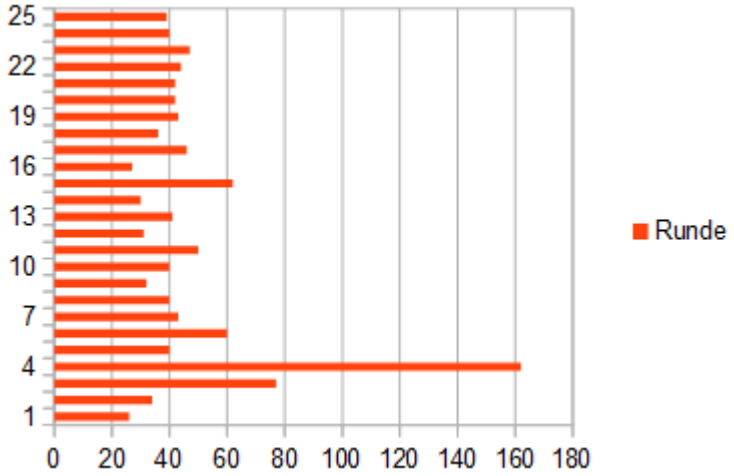

Abbildung 6.2.: Endergebnis der Spielpaarung Heuristisch (Nord) vs. Intelligent (Süd): Verteilung der erreichten Spielrunde bei 25 Spielen und zeigt in welcher Spielrunde der heuristische Agent besiegt wurde.

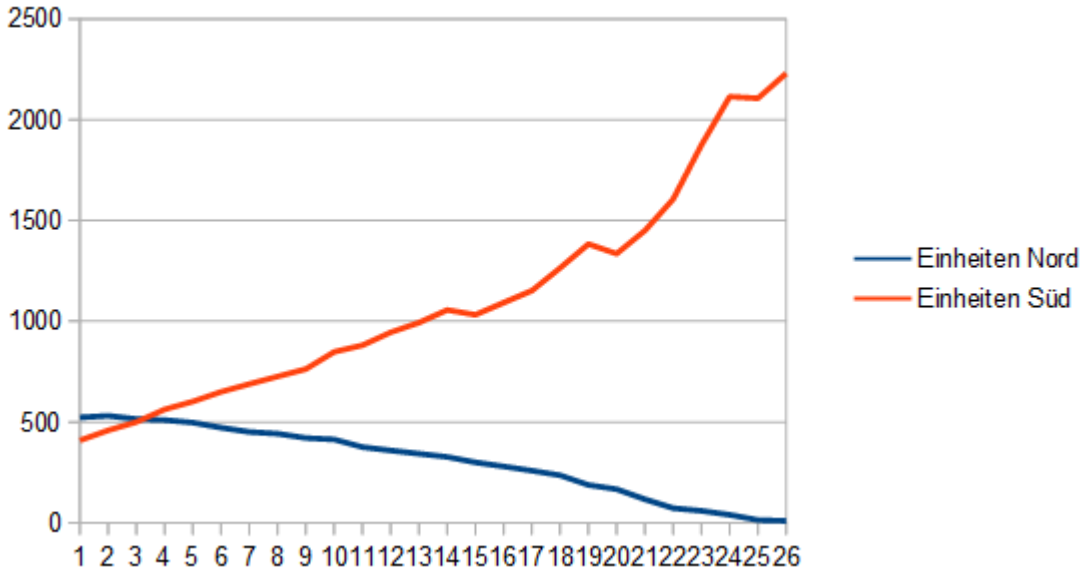

Abbildung 6.3.: Spielverlauf der Spielpaarung beim ersten Spiel mit 25 Runden: Heuristisch (Nord) vs. Intelligent (Süd).

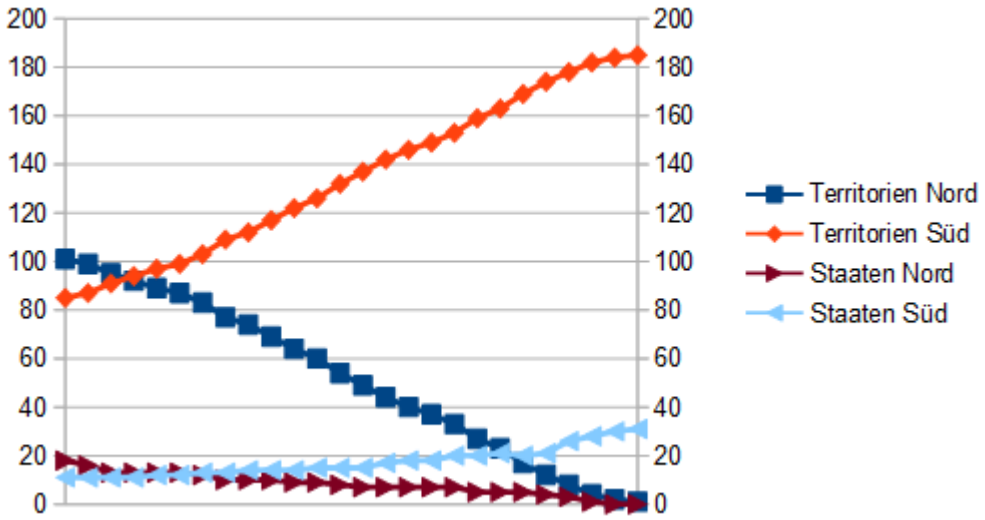

Abbildung 6.4.: Spielverlauf der Spielpaarung beim ersten Spiel mit 25 Runden: Heuristisch (Nord) vs. Intelligent (Süd).

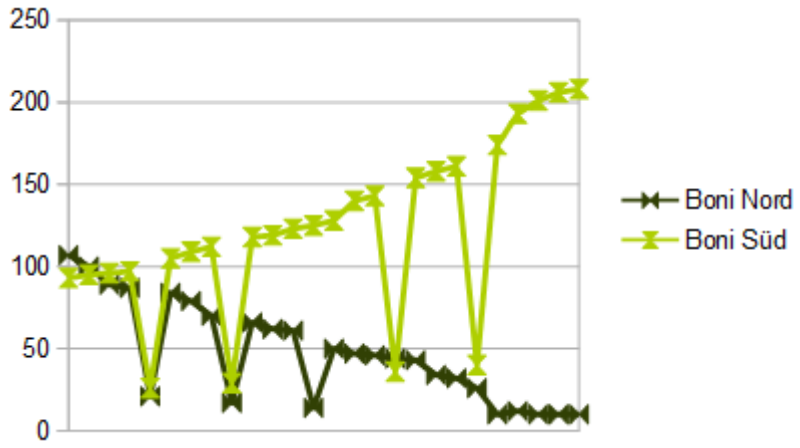

Abbildung 6.5.: Spielverlauf der Spielpaarung beim ersten Spiel mit 25 Runden: Heuristisch (Nord) vs. Intelligent (Süd).

Nordstaaten	Südstaaten	Sieg Nordstaaten in %
Zufallsbasiert	Erweitert	0
Erweitert	Zufallsbasiert	0

Tabelle 6.2.: Zufallsbasierter und erweiterter zufallsbasierter Agent spielen gegeneinander. Jeweils mit Seitenwechsel. Vergleich bei 100 unabhängigen Spielen. Die Ergebnisse sind Durchschnittswerte.

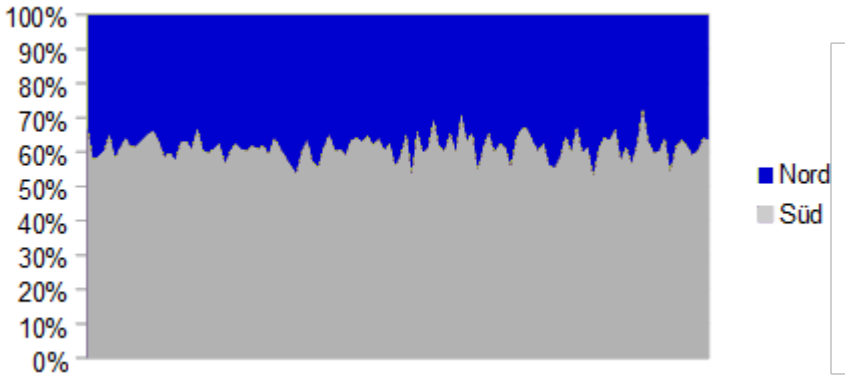

Abbildung 6.6.: Endergebnis der Spielpaarung Zufallsbasiert (Nord) vs. Erweiterter (Süd): Gezeigt wird die prozentuale Verteilung der Territorien bei 100 Spielen.

Um die Leistung der einzelnen Agenten besser einstufen zu können, wurden für jede mögliche Agentenpaarung Experimente durchgeführt.

Als der zufallsbasierte und der erweiterte zufallsbasierte Agent gegeneinander spielten, konnte keiner der beiden Agenten in 100 Spielen einen Sieg davon tragen (Tabelle 6.2). Dennoch hat der erweiterte zufallsbasierte Agent eine Dominanz durchgängig während der Spiele entwickelt hat, da er bei der prozentualen Verteilung von Einheiten und Territorien immer mehr als 50 % erzielt hat. Unterstützt wird diese Aussage von den Abbildungen 6.6, 6.7 und 6.8.

In Abbildung 6.9 wird die Verteilung der Forts, Gräben und Generäle angezeigt. Die Ergebnisse zeigen, dass die Anzahl von Forts und Gräben sehr stark variieren kann. Der Bau von Forts und Gräben hat beim intelligenten Agenten niedrigste Priorität und wird nur für Grenzterritorien durchgeführt.

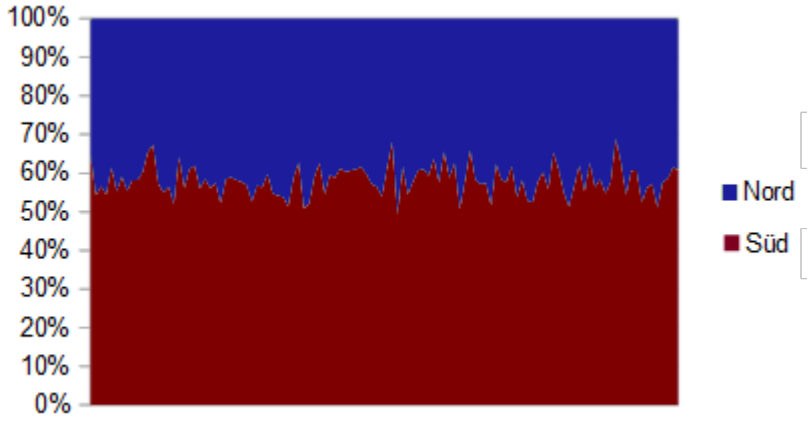

Abbildung 6.7.: Endergebnis der Spielpaarung Zufallsbasiert (Nord) vs. Erweiterter (Süd): Gezeigt wird die prozentuale Verteilung der Einheiten.

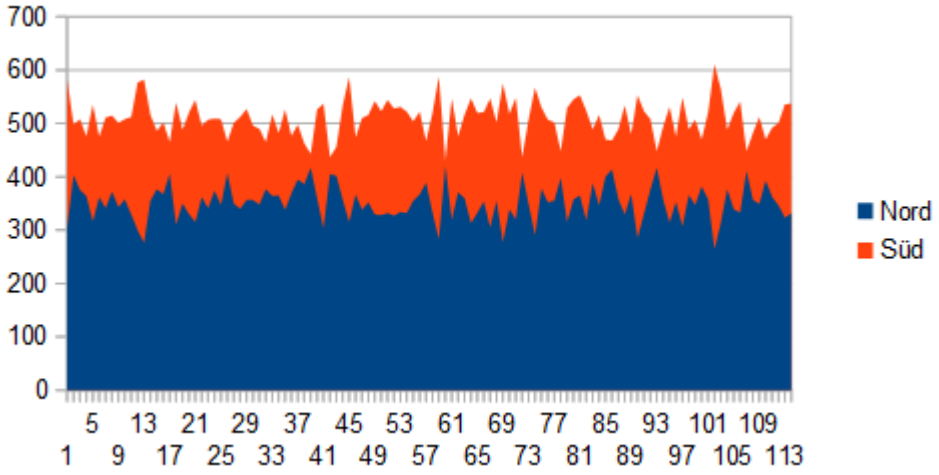

Abbildung 6.8.: Endergebnis der Spielpaarung Zufallsbasiert (Nord) vs. Erweiterter (Süd): Gezeigt wird die Verteilung der Spielpunkte bei 100 Spielen.

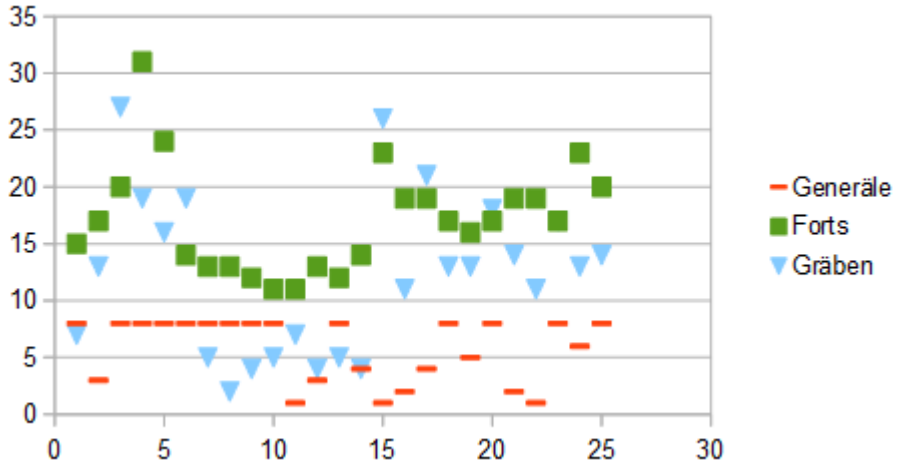

Abbildung 6.9.: Endergebnis der Spielpaarung Heuristisch (Nord) vs. Intelligent (Süd): Verteilung von Forts, Gräben und Generäle des intelligenten Spielers bei 25 Spielen bei vollständigem Sieg von Süd.

Nordstaaten	Südstaaten	Spiele	Punkte
Zufallsbasiert	Zufallsbasiert	100	451 zu 405
Erweitert	Erweitert	100	420 zu 453
Heuristischer	Heuristischer	1	862 zu 11
Intelligent	Intelligent	6	450 zu 437

Tabelle 6.3.: Agenten spielen jeweils gegen sich selbst. Vergleich mit unterschiedlich vielen unabhängigen Spielen. Die Ergebnisse sind Durchschnittswerte.

Nordstaaten	Südstaaten	Spiele	Sieg Heuristisch in %	Territorienanzahl *
Zufallsbasiert	Heuristisch	105	79,05	189
Heuristisch	Zufallsbasiert	101	14,85	188
Erweitert	Heuristisch	105	8,57	166
Heuristisch	Erweitert	129	3,88	158

Tabelle 6.4.: Der heuristische Agenten spielt gegen die beiden anderen nicht intelligenten Agenten. Jeweils mit Seitenwechsel. * Kleinster Wert der Territorienanzahl für den heuristischen Agenten in allen Spielen.

Beim Spielen der Agenten gegen sich selbst (siehe Tabelle 6.3) wird eine Balance festgestellt. Beim heuristischen Agenten wurde bei einem Spiel ein Sieg erreicht. Bei nur einem Spiel des heuristischen Agenten lassen sich daraus keine Rückschlüsse ziehen.

Zur Einstufung des heuristischen Agenten musste dieser gegen die beiden nicht intelligenten Agenten spielen. Dargestellt wird das Ergebnis dieses Experiments in Tabelle 6.4. Im Vergleich zum erweiterten zufallsbasierten Agenten hat der heuristische Agent schlecht abgeschnitten, denn die Anzahl der Siege ist unter 10 %, was auf eine schlechte Leistung dieses Agenten allgemein rückschließen lässt.

Obwohl der intelligente Agent schon sehr gute Ergebnisse gezeigt hat, wurden noch weitere Experimente durchgeführt (Tabelle 6.5 und 6.6) bei denen die globalen Parameter der UCT-Implementierung variiert wurden (`A`, `I`, `T` und `C`). Durch die geringe Anzahl der Experimente (aus Zeitmangel), und den schwach spielenden nicht intelligenten Agenten lassen sich keine direkten Erkenntnisse daraus ableiten, außer dass die Siegwahrscheinlichkeit des intelligenten Agenten in jeder Konfiguration recht hoch ist.

In der Tabelle 6.7 wird gezeigt, dass beim Sieg der heuristischen Agenten über den erweiterten zufallsbasierten Agenten, dieser kurz vor dem Spielende stattgefunden hat. Das bedeutet im Zusammenhang mit den bis jetzt über den heuristischen Agenten gesammelten Informationen, dass der heuristische tatsächlich schlecht spielt. Man muss

Nordstaaten	Südstaaten	Spiele	Runden	Parameter	Sieg in %	σ
Heuristisch	Intelligent	2	66	C = 0,5	100	0
Heuristisch	Intelligent	5	82	C = 2,5	80*	7,16
Heuristisch	Intelligent	2	47	A = 16	100	0
Heuristisch	Intelligent	2	64	I = 2000	100	0

Tabelle 6.5.: UCT-Anpassung für die Landkarte. Vergleiche mit unterschiedlich vielen unabhängigen Spielen. Die Ergebnisse sind Durchschnittswerte. * Bedeutet Unentschieden bei einem Spiel, aber der heuristische Spieler hatte nur noch zwei Territorien.

Nordstaaten	Südstaaten	Spiele	Runden	Parameter	Sieg in %	σ
Heuristisch	Intelligent	3	106	C = 0,5	66,67*	12,81
Heuristisch	Intelligent	2	39	A = 40	100	0
Heuristisch	Intelligent	2	56	I = 1000	100	0
Heuristisch	Intelligent	1	34	T = 40	100	0

Tabelle 6.6.: UCT-Anpassung für die Schlacht. Vergleiche mit unterschiedlich vielen unabhängigen Spielen. Die Ergebnisse sind Durchschnittswerte. * Bedeutet Unentschieden bei einem Spiel, aber der heuristische Spieler hatte nur noch ein Territorium.

aber in Betracht ziehen, dass die Abbildung 6.10 am Ende des Kapitels, die Aussage doch etwas relativiert, denn dort zeigt die prozentuale Verteilung der Territorien ganz deutlich, dass gegen Spielende der erweiterte zufallsbasierte Agent nur noch maximal bis zu 20 % der Territorien bei einem Spiel hat.

Anzahl	95	1	1	1	1	1
Runde	240	238	236	227	175	174

Tabelle 6.7.: Endergebnis der Spielpaarung Heuristisch (Nord) vs. Erweiterter (Süd): Gezeigt wird die Häufigkeit der erreichten Runde bei einer Spielanzahl von 100 bei Dominanz des heuristischen Agenten.

Folgende Standardeinstellungen des intelligenten Agenten bezüglich UCT für die Ansicht Landkarte wurden vorgenommen:

- Anzahl der Aktionsgruppen = 8 (`A`)
- Anzahl der Iterationen = 1000 (`I`)
- Maximale Laufzeit in Sekunden wurde deaktiviert (`S`)
- Maximale Anzahl der Runden pro Simulation = 5 (`T`)
- Konstante C der UCB1-Formel = 1,44 (`C`)

Folgende Standardeinstellungen des intelligenten Agenten bezüglich UCT für die Ansicht Schlacht wurden vorgenommen:

- Anzahl der Aktionsgruppen = 20 (`A`)
- Anzahl der Iterationen = 500 (`I`)
- Maximale Laufzeit in Sekunden wurde deaktiviert (`S`)
- Maximale Anzahl der Runden pro Simulation = 20 (`T`)
- Konstante C der UCB1-Formel = 1,44 (`C`)

Schließlich gab es auch ein Experiment, bei dem ein menschlicher Spieler (der Autor dieser Arbeit) gegen den intelligenten Agenten gespielt hat. Alle Spielregeln waren aktiviert. Der menschliche Spieler übernahm die Rolle der Nordstaaten. Nach etwa 60 Runden wurde das Spiel abgebrochen. Es konnte keine der beiden Spieler einen Vorteil erreichen. Jeweils auf beiden Seiten gab es Geländegewinne. Der intelligente Agent hat sich in diesem Spiel gegen einen menschlichen Spieler als spielstark erwiesen.

Abbildung 6.10.: Endergebnis der Spielpaarung Heuristisch (Nord) vs. Erweiterter (Süd): Gezeigt wird die prozentuale Verteilung der Territorien bei 100 Spielen.

7. Fazit

Untersucht wurde in dieser Arbeit das baumbasierte Suchverfahren für intelligente Agenten im Kontext rundenbasierter Strategiespiele. Im Detail wurde die Implementierung von UCT näher analysiert.

Dabei wurde festgestellt, dass dieser Algorithmus als Basis für den intelligenten Agenten den anderen in dieser Arbeit vorgestellten Agenten deutlich überlegen ist. Trotz Überlegenheit spielt der intelligente Agent dennoch nicht immer optimal. So wurde beobachtet, das während einer Schlacht günstige Spielzüge für den Ausgang der Schlacht nicht sofort gespielt wurden. Auch verbleiben oft Einheiten in einem bestimmten Gebiet auf der Landkarte für mehrere Runden, sodass diese nicht immer an die Grenze gezogen werden. Wobei diese Art von Spielzug (eine Reserve an Einheiten im Hinterland) schon wieder eine Strategie sein kann.

Grundsätzlich verfolgt der Agent eine schnellst mögliche Eroberung des Gegners durch Überlegenheit mittels Konzentration der Einheiten an der Grenze.

Für weitere Verbesserungen des intelligenten Agenten wäre es wichtig, einen zusätzlichen heuristischen Agenten einzuführen, der die gleiche Heuristik wie der intelligente Agent implementiert. Somit wäre eine bessere Vergleichsbasis vorhanden. Dazu müssten die Routinen des intelligenten Agenten, die auf Arrays basieren, kopiert und angepasst werden, sodass diese stattdessen auf die Originalobjekte der Spielimplementierung zugreifen.

Desweiteren wären zusätzliche Testspiele gegen menschliche Spieler ein Ansatz, um Schwächen in der UCT-Implementierung zu beobachten und dann auszugleichen. Eine vollständig getrennte und angepasste Heuristik für die Schritte Selektion und Simulation wäre ein weiter Ansatzpunkt für eine Verbesserung.

Die Berechnung der Spielzüge dauert auf einem Rechner mit aktueller Hardware (Intel Core i5 mit 1,7 GHz) für die Schlacht fünf Sekunden (abhängig von Einheitenanzahl) und für die Landkarte zwei Sekunden. Gemessen wurde auch die gesamte Spieldauer. Diese betrug auf der oben genannten Hardware für alle Spielpaarungen heuristischer Agent gegen intelligenten Agenten im Durchschnitt 80 Minuten. Um die Anzahl der Iterationen zu erhöhen, wäre es möglich die UCT-Implementierung in die Programmiersprache C++ zu übersetzen und über JNI [28] in das bestehende Java-Programm einzubinden. Denn es ist allgemein bekannt das C++ deutlich schneller als Java in der Ausführung ist [27]. Der Konvertierungsaufwand ist in Planung und eher gering, da die verwendeten Sprachkonstrukte in beiden Sprachen vorhanden sind. Der Zeitaufwand

beim Aufruf von UCT für den jeweiligen Datenaustausch zwischen C++ und Java ist vernachlässigbar.

Die Simulation verwendet eine unvollständige Kopie der Spielregeln. Eine Annäherung an alle Regeln ist sinnvoll und würde somit das Simulationsergebnis verbessern.

Die Erweiterung von UCT auf alle Regeln des Spiels insbesondere der verschiedenen Spielkarten, wäre ein anspruchvolle und herausfordernde Arbeit. Im bestehenden Programm müsste nicht-deterministische Elemente deterministisch abgebildet werden. Einen ersten Schritt findet sich in der Klasse `DeterministicDiceThrow`, die für die Ansicht Schlacht alle Würfelergebnisse im voraus berechnen würde.

Die globalen Parameter des UCT-Algorithmus wurden in Kapitel 7 variiert. Eine weitere Möglichkeit in diesem Zusammenhang wäre zu untersuchen, wie sich die Leistung von UCT ändert, wenn bei der Expansion mehr als ein Kindknoten angelegt würde.

Ein wichtiger Schritt wäre ein Verfahren zum Erkennen von Aktionengruppen, deren Spielzüge sich so stark ähneln, dass durch diese Redundanz Platz verschwendet wird und bessere Spielzüge erst gar nicht ermittelt werden.

Die Spielregeln für die Schlacht erlauben es, Einheiten erst zu bewegen und dann anzugreifen und dann weitere Bewegungen mit bereits verwendeten Einheiten durchzuführen. Diese Möglichkeit wird von der UCT-Implementierung noch nicht berücksichtigt.

Anhang

A. Java-Quelltextausschnitte

Listing A.1: Aufbau der Klasse MainMapAction

```

class MainMapAction {

    static public final int AttackTo = 0;
    static public final int AttackUnitCount = 1;
    static public final int MoveTo = 2;
    static public final int MoveUnitCount = 3;
    static public final int ReinforcementUnitCount = 4;
    static public final int FortifyFort = 5;
    static public final int FortifyTrenches = 6;

    static public final int SIZE = 7;

}
```

Listing A.2: Aufbau der Klasse MainMapProvince

```

class MainMapProvince {

        ...     // hier wurden einige Quelltextzeilen weggelassen

    static public final int HomelandId = 0;
    static public final int City = 1;
    static public final int Port = 2;

        ...     // hier wurden einige Quelltextzeilen weggelassen

    static public final int NeighboursCount = 9;

    static public final int NEIGHBOURS = 10;
    static public final int RIVER = 10 + MAX_NEIGHBOURS;

        ...     // hier wurden einige Quelltextzeilen weggelassen

}
```

Listing A.3: Aufbau der Klasse BattleMapAction

```
1
2 class BattleMapAction {
3
4     static public final int MoveFrom = 0;
5     static public final int AttackFrom = 1;
6     static public final int To = 2;
7
8     static public final int Unit = 3;
9     static public final int MovesAndAttacksWithoutPayingSupply = 4;
10
11    static public final int PlaceUnit = 5
12
13    static public final int Reinforcement = 6;
14    static public final int Supply = 7;
15
16    static public final int SIZE = 8;
17 }
```

Listing A.4: Aufbau der Klasse BattleMapLocation

```
1
2 class BattleMapLocation {
3
4         ...     // hier wurden einige Quelltextzeilen weggelassen
5
6     static public final int UnitCount = 0;
7     static public final int FactionId = 1;
8
9         ...     // hier wurden einige Quelltextzeilen weggelassen
10
11    static public final int UNITS = 6;
12
13    static public final int UNIT_HashCode = UNITS;
14    static public final int UNIT_Supply = UNITS + MAX_UNITS;
15
16        ...     // hier wurden einige Quelltextzeilen weggelassen
17
18 }
```

Literaturverzeichnis

[1] Enn Tyugu. *Algorithms and Architectures of Artificial Intelligence*. IOS Press, 2007.

[2] L. Kocsis and C. Szepesvári. Bandit based monte-carlo planning. Euro. Conf. Mach. Learn. Berlin, Germany, 2006. Springer.

[3] Vortrag auf der Game/AI Konferenz in Wien 2014. `http://aigamedev.com/open/coverage/mcts-rome-ii/`. [Online; Zugriff 16.Juli 2015].

[4] Spielanleitung Risiko: Star Wars. `http://winningmoves.de/fileadmin/spielanleitungen/RISIKO_Star_Wars.pdf`. [Online; Zugriff 16.Juli 2015].

[5] Risikospiel für iPad. `https://itunes.apple.com/us/app/risk-official-game-for-ipad/id407085219?mt=8`. [Online; Zugriff 16.Juli 2015].

[6] Spielanleitung Battle For Stalingrad. `http://images.dvg.com/www.dvg.com/stalingradrulebook.pdf`. [Online; Zugriff 16.Juli 2015].

[7] Computerspiel Heroes of Might & Magic III. `http://might-and-magic.ubi.com/universe/de-de/games/all-games/might-and-magic-heroes-3-hd/index.aspx`. [Online; Zugriff 16.Juli 2015].

[8] Computerspiel Defender of the Crown. `http://www.gamesbasis.com/defender-of-the-crown.html`. [Online; Zugriff 16.Juli 2015].

[9] Computerspiel North & South. `http://www.gamesbasis.com/north-and-south.html`. [Online; Zugriff 16.Juli 2015].

[10] Cameron Browne, Edward Powley, Daniel Whitehouse, Simon Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. volume 4. IEEE transactions on computational intelligence and AI in games, 2012.

[11] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2009.

[12] Michael Wolf. An intelligent artificial player for the game of risk. Master's thesis, Darmstadt University of Technology, 2005.

[13] Plattformunabhängige Programmiersprache Java. `http://java.com/de/`. [Online; Zugriff 16.Juli 2015].

[14] Android Betriebssystem für Smartphones und Tablets. https://www.android.com/. [Online; Zugriff 16.Juli 2015].

[15] iOS Betriebssystem für Smartphones und Tablets. https://www.apple.com/de/ios/. [Online; Zugriff 28.Juli 2015].

[16] Android Studio: offizielle Entwicklungsumgebung für Android-Anwendungen. http://developer.android.com/tools/studio/index.html. [Online; Zugriff 16.Juli 2015].

[17] NetBeans: Entwicklungsumgebung für Java-Anwendungen auf Computern. https://netbeans.org/. [Online; Zugriff 16.Juli 2015].

[18] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Entwurfsmuster als Elemente wiederverwendbarer objektorintierter Software*. mitp, 2015.

[19] Th. H. Cormen, Ch. E. Leiserson, R. Rivest, and C. Stein. *Algorithmen - Eine Einführung*. Oldenbourg, 2007.

[20] Cameron Browne. http://ccg.doc.gold.ac.uk/teaching/ludic_computing/ludic16.pdf. [Online; Zugriff 17.Juli 2015].

[21] Kurt Binder and Dieter W. Heermann. *Monte Carlo Simulation in Statistical Physics*. Springer Verlag, 2010.

[22] G. M. J.-B. Chaslot, S. Bakkes, I. Szita, and P. Spronck. Monte-carlo tree search: A new framework for game ai. page 216–217, 1600 Amphitheatre Parkway, Mountain View, CA, 94043 USA, 2008. Artif. Intell. Interact. Digital Entert. Conf., Stanford Univ.

[23] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Mach. Learn.*, 47:235–256, 2002.

[24] Frederik Christiaan Schadd. Monte-carlo search techniques in the modern board game thurn and taxis. Master's thesis, Maastricht University, 2009.

[25] S. Edelkamp, P. Kissmann, D. Sulewski, and H. Messerschmidt. Move ordering vs heavy playouts: Where should heuristics be applied in monte carlo go. page 35–42, 1600 Amphitheatre Parkway, Mountain View, CA, 94043 USA, 2007. Amer. Game-On Conf., Gainesville, Florida.

[26] G. M. J.-B. Chaslot, M. H. M. Winands, H. J. van den Herik, J. W. H. M. Uiterwijk, and B. Bouzy. *Progressive Strategies for Monte-Carlo Tree Search*, volume 4. New Math. Nat. Comput., 2008.

[27] Robert Hundt. *Loop Recognition in C++/Java/Go/Scala*. Google, 1600 Amphitheatre Parkway, Mountain View, CA, 94043 USA, 2007.

[28] Christian Ullenboom. *Java 7 - Mehr als eine Insel*. Rheinwerk Computing, 2012.

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht. Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen. Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Ort, Datum

Bernd Nötscher