Ein einheitliches Spiel Such Framework

A unified Game Search Framework
Bachelor-Thesis von Nils Mario Schröder aus Darmstadt
Tag der Einreichung:

1. Gutachten: Professor Johannes Fürnkranz
2. Gutachten: Christian Wirth

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Knowledge Engineering Group

Ein einheitliches Spiel Such Framework
A unified Game Search Framework

Vorgelegte Bachelor-Thesis von Nils Mario Schröder aus Darmstadt

1. Gutachten: Professor Johannes Fürnkranz
2. Gutachten: Christian Wirth

Tag der Einreichung:

Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 9. Mai 2017

(N. Schröder)

Zusammenfassung

Diese Arbeit befasst sich mit dem Vereinheitlichen verschiedener Algorithmen zu einem Framework im Bereich Spielbaumsuche.
Es werden folgende Algorithmen in diesem Framework implementiert: *Breitensuche*, *Tiefensuche*, *Best First Search (BFS)*, *Heuristic Search* und *Monte Carlo Tree Search (MCTS)*.
Um eine gute Zusammenarbeit zwischen den Algorithmen zu ermöglichen, wurden sie teilweise modifiziert. Das bedeutet, ihre Iterationsvorschrift wurde nicht verändert, nur die Reihenfolge einzelner Schritte vertauscht.
Es wurde erfolgreich ein Framework implementiert, indem verschiedene Spielbaum Suchalgorithmen implementiert und modularisiert wurden. Hierbei wurden die Algorithmen auf ein allgemeines Schema gebracht, welches aus den folgenden vier Schritten besteht: Init, Reward, Propagation und Update. Durch diese Änderung wurde es möglich gemacht dem Nutzer dieses Frameworks verschiedene grundsätzliche Algorithmen zur schnellen Verfügung und Kombination zugänglich zu machen. Mittels der Modularisierung ist es möglich Teilschritte zu vertauschen und Algorithmen zu Kombinieren. Diese Möglichkeit führt zu einem positiven Resultat.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Diese Arbeit befasst sich mit dem Vereinheitlichen verschiedener Algorithmen zu einem Framework im Bereich Spielbaumsuche.
Spielbaumsuche ist eine Unterkategorie der Baumsuche und verwendet ihre Verfahren.
Diese sind mittlerweile Teil des Standardwissens in der Künstlichen Intelligenz. Sie finden sich zu großen Teilen als Grundbaustein in verschiedenen Domänen, zum Beispiel kognitive Systeme, Spielprogramme, Theorembeweisen und Robotersteuerungen [9]. Probleme in diesen Anwendungen können sehr komplex sein und lassen sich nicht nur durch initiales Wissen alleine lösen. Bei derartigen Problemen wird das Ausgangsproblem zur Lösung sukzessiv in kleinere Teilprobleme zerlegt und alle verschiedene Lösungswege ausprobiert.
Eine momentan üblichste Herangehensweise in der Spielbaumsuche ist momentan für verschiedene Probleme einzelne, genau auf das Problem zurecht geschnittene Algorithmen zu benutzen. Diese können aber dann nicht für ein ähnliches Problem genutzt werden und müssen von Hand angepasst, wenn nicht teils redundant, neu implementiert werden.
Es werden folgende Algorithmen in diesem Framework implementiert: *Breitensuche*, *Tiefensuche*, *Best First Search (BFS)*, *Heuristic Search* und *Monte Carlo Tree Search (MCTS)* ([4] , [15]).
Um eine gute Zusammenarbeit zwischen den Algorithmen zu ermöglichen, wurden sie teilweise modifiziert. Das bedeutet, ihre Iterationsvorschrift wurde nicht verändert, nur die Reihenfolge einzelner Schritte vertauscht.
Dieses Framework soll der schnellen Implementation grundlegender Algorithmen dienen und bietet die Möglichkeit der Modularisierung dieser Algorithmen. Das soll heißen, man kann einzelne Teile dieser Algorithmen miteinander kombinieren.
Ein weiterer Punkt ist, dass man verschiedene Schritte der implementierten Algorithmen miteinander kombinieren kann, um so evtl. besser auf spezifische Probleme eingehen zu können.
Alle oben genannten Algorithmen werden dafür in vier grundlegende Schritte aufgeteilt: *Init*, *Reward*, *Propagation* und *Update*. Dieses Vorgehen ähnelt dem von *Monte Carlo Tree Search* und ist gut übertragbar auf die oben genannten Algorithmen.
Wenn nun ein Problem mehrere spezifische Algorithmen zur Lösung braucht, bzw. Teile verschiedener Algorithmen, wird es schnell unübersichtlich. Hierfür lohnt sich ein einheitliches Framework, welches grundlegende Algorithmen zur Verfügung stellt.
Weiterhin stellt das Framework Teile dieser Algorithmen zur Verfügung und ermöglicht es dem Nutzer, *BFS* mit zum Beispiel *MCTS* Rollouts laufen zu lassen, um so stochastische Effekte leichter zu erkennen. Diese Kombination gibt dem Nutzer die Möglichkeit, bei (verschiedenen) Problemen einfache Algorithmen mit effektiven Strategien anzuwenden und ihre Performanz damit zu verbessern.

1.1 Problemdefinition

Das momentan Üblichste im Bereich Spielbaumsuche ist es, jeden Algorithmus einzeln zu implementieren und diesen dann genau auf das Problem zuzuschneiden. Das bedeutet, dass man sich in der Wiederverwendbarkeit der Algorithmen einschränkt, d.h. man muss die Algorithmen neu adaptieren. Für manche Probleme lohnt es sich, verschiedene Algorithmen miteinander zu verknüpfen, oder schnell zwischen ihnen wechseln zu können. Dies wird zum Beispiel interessant, wenn man ein Spiel lösen möchte, welches am Anfang deterministisch ist und im Laufe des Spiels stochastische Elemente hinzukommen. Mit dem Framework kann man einfach zwischen deterministischen Suchalgorithmen wie *Breitensuche*, zu *Monte Carlo Tree Search* wechseln, um Teile der vorher gesammelten Daten zu erhalten. Weiterhin ist es möglich eine Breitensuche mit Rollout Logik von *MCTS* verbinden und so versuchen, stochastische Elemente leichter zu behandeln.

1.2 Leitfaden über Kapitel

Im folgenden Kapitel werden Grundlagen behandelt. Die im Framework implementierten Algorithmen und die Änderungen, die man an ihnen vornehmen musste, um sie vereinheitlichen zu können, werden beschrieben. In Kapitel 5 wird das Framework und dessen Datenspeicherung genau erläutert. Zudem wird die Nutzung im Detail erklärt.

2 Preliminaries

In diesem Kapitel werden grundlegenden Begriffe erläutert, welche zum Verstehen der Arbeit notwendig sind.

2.1 Markov Entscheidungsprozesse

Bei dem *Markov Entscheidungsprozess* oder auch *Markov decision process* (MDP) handelt es sich um Entschidungsprobleme, in welchen die nächste Aktion eines Agenten von mehreren Entscheidungen abhängig ist [17]. Um eine Lösung von sequentiellen Entscheidungsproblemen algorithmisch aufarbeiten zu können, muss eine Problemstellung formuliert werden. Dies geschieht im Folgenden.
Wenn man von Markov Entscheidungsprozessen (MDP) redet, ist ein Framework gemeint, welches zu Modellierung sequentieller Entscheidungsprobleme in stochastischen Umgebungen dient. Seine Eigenschaften sind wie folgt:

- S: Menge an Zuständen
- A: Menge aller möglichen Aktionen
- $A(s) \subseteq A$: Menge aller ausführbaren Aktionen in einem gegebenen Zustand $s \in S$
- $\delta(s'|a,s)$: Transitionsfunktion, welche die Wahrscheinlichkeit für den Übergang in Zustand s', gegeben des momentanen Zustands s und der Aktion a angibt.
- $R(s) \in \mathbb{R}$: Belohnungsfunktion, welche für jeden Zustandsübergang zum Zustand s eine Güte $\in \mathbb{R}$ zuordnet.
- s_0: Startzustand des Entscheidungsproblems.

Hierbei ist gemeint, dass wenn ein Agent in einem Zustand ist, er eine Aktion ausführen muss, um in einen anderen/neuen Zustand zu gelangen. Es gilt, dass der Agent sich zu jedem Zeitpunkt in einem Zustand $(s_t : t \in \mathbb{N})$ befindet. Nun kann der Agent durch Ausführen einer möglichen Aktion $A(s_t)$ in einen neuen Zustand gelangen. Die Wahrscheinlichkeit, in welchen Zustand er gelangt ist gegeben durch $\delta(s'|a,s_t)$. Nach Ausführen einer Aktion erhält man folgende zwei Informationen:

1. in welchen Zustand s' der Agent nach Ausführung von Aktion a gelangt.
2. wie hoch die Belohnung der Transition war, also $R(s')$.

Hierbei besteht das Ziel daraus, eine effiziente Strategie $\pi : S \times A \rightarrow [0,1]$ zu finden. Das bedeutet, die Strategie gibt zu jedem Zustands-Aktionspaar an, mit welcher Wahrscheinlichkeit $p \in [0,1]$ welche Aktion a in Zustand s ausgeführt wird, sodass eine Maximierung der Güte über die Zeit erreicht wird.
Bei gegebener Strategie kann man einem Zustands-Aktions-Paar eine Güte zuordnen, welche aus den bisherigen resultierenden Belohnungen bestimmt wird. Dazu rechnet man einen Discountfaktor von $\gamma \in [0,1]$ ein, welcher zeitlich frühere Belohnungen höher bewertet als spätere Belohnungen.

$$Q^{\pi}(s,a) = \sum_{t=0}^{inf} \gamma \sum_{s \in S} P^t(s) R(s)$$

Es ergibt sich die Wahrscheinlichkeit $P^t(s)$, dass der Agent zum gegebenen Zeitpunkt t im Zustand s ist. Dieser Wert wird iterativ berechnet.

$$P^t(s_0)$$

und

$$P(s_{t+1}) = \sum_{s \in S} P_t(s) \sum_{a \in A(s)} \delta(s_{t+1}|a, s_t)\pi(s_t, a)$$

Somit ist die optimale Lösung des Entscheidungsproblems die Strategie π^*, wobei

$$\pi^*(a|s) = argmax_\pi \sum_{a \in A(s)} Q^{\pi^*}(s, a)\pi(s, a)$$

Zu beachten ist, dass es Sonderfälle in deterministischen MDPs gibt. In dieser Arbeit werden sowohl deterministische als auch stochastische MDPs verwendet. Für deterministische MDPs gilt, es gibt nur einen Nachfolgezustand s' für einen Zustand s und eine Aktion a.

$$\forall s \in S, a \in A(s) \exists s' \in S : (\delta(s'|a, s) = 1) \bigwedge (\forall s^* \neq s' : \delta(s^*|a, s) = 0)$$

2.2 Allgemeine Begriffe

In diesem Abschnitt werden grundlegende Begriffe zum Verständnis des Frameworks geklärt.

- $\delta(s'|s, a)$ ist eine Transitionsfunktion, welche die Wahrscheinlichkeit für den Übergang in Zustand s', entsprechend des momentanen Zustands s und der Aktion a angibt.
- $\pi(a|s)$ ist eine Strategie, welche die Wahrscheinlichkeit zum Auswählen einer Aktion a im Zustand s dient.
- $r(s)$ steht für die Belohnung, welche man in einem Zustand bekommt, meistens $r(s) \in -1, 0, 1$ für Niederlage/Sieg oder 0 für nicht terminale Zustände.
- $\phi(s)$ repräsentiert die Merkmale, welche mit state s assoziiert werden. Dies kann eine Zeit $\phi_t(s)$ besitzen, welche die Spielzeit repräsentiert.
- $V^*(s)$ repräsentiert die voraussichtliche Langzeitbelohnung, welche in einem Zustand erreichbar ist, damit ist gemeint:

$$V^*(s) = \mathbb{E}_{\delta(s_{s+1}|s_t, \pi^*(s_t))} \sum_t r(s_t)\delta(s_{s+1}|s_t, \pi^*(s_t)) \tag{1}$$

  $V_i(s)$ ist das Kennzeichen der Approximation von $V^*(s)$ zur Zeit i.
- $v_i(s, a)$ definiert die Wichtigkeit eines Zustand/Aktionspaars für die Baumsuche zum Zeitpunkt i. Dieser skaliert normalerweise mit der erwarteten Langzeit Belohnung.
- $h(s) = f(\phi(s)) \approx V^*(s)$ ist eine (heuristische) Funktion, welche einen Zustand s evaluiert. Diese Bewertung basiert auf seinen Eigenschaften $\phi(s)$ und liefert eine Approximation für $V^*(s)$

2.3 Baumsuche

Mit dem Begriff Baumsuche wird das Durchsuchen eines Zustandsraumes und der dadurch iterativ entstehende Suchbaum beschrieben. Hierbei ist der Zustandsraum gegeben durch einen gerichteten Graph (K,E), wobei K (Anzahl an Knoten) die Zustände und E (Anzahl an Kanten) die möglichen Aktionen repräsentieren. Nun kann man, wenn zwischen zwei Knoten k_1 und k_2 eine Kante e_1 besteht, den Zustand k_2 erreichen, indem man im Zustand k_1 Aktion e_1 ausführt.
Beim Ausführen eines Baumsuchalgorithmus gilt, dass zu jedem Zeitpunkt t der aktuelle Suchbaum und eine Anzahl an Knoten, welche in der nächsten Iteration zum Baum hinzugefügt werden können, existiert. Für jeden dieser Knoten k' der Menge gilt, dass man durch eine Kante $e' = (k_n, k')$ von einem im Baum existierenden Knoten k_n verbunden ist, aber noch kein Teil dieses Suchbaums ist. Aus dieser Menge ergibt sich die Fringe (der Rand).
Es ist gegeben, dass in jeder Iteration mindestens ein neuer Knoten k' aus der Fringe, mit einer zugehörigen Kante e', in den momentanen Suchbaum übernommen wird. Welches oder wie viele, ist hierbei abhängig vom jeweiligen Algorithmus, der zum Suchbaum aufspannen verwendet wird.

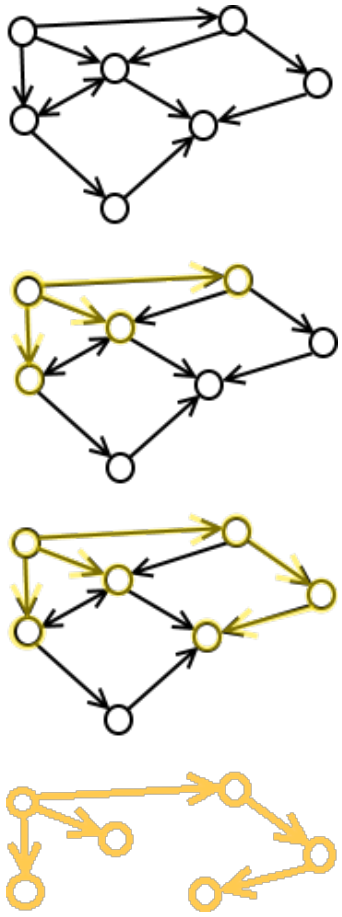

Abbildung 1: Das Aufspannen eines Suchbaums

2.3.1 Exploration versus Exploitation

In der Baum- und Spielbaumsuche stellt sich oft die Frage, welcher Knoten als nächstes, für weitere Bearbeitung, selektiert werden soll. Je nachdem welchen Algorithmus man verwendet, hat dieser strikte Richtlinien für die Selektion. Andere Algorithmen selektieren anhand von heuristischen Werten, diese Werte sind aber nur eine Schätzung. Der Erfolg ist abhängig von der verwendeten Heuristik [11].
Um vorzubeugen, dass ein Algorithmus sich ausschließlich Knoten mit den besten heuristischen Werten selektiert, sondern auch Knoten ausprobiert, die eventuell nicht so vielversprechend aussehen, gibt es *Exploration versus Exploitation*. Das wählen nicht optimaler Züge, kann in einigen Problemen zu besseren Resultaten führen.

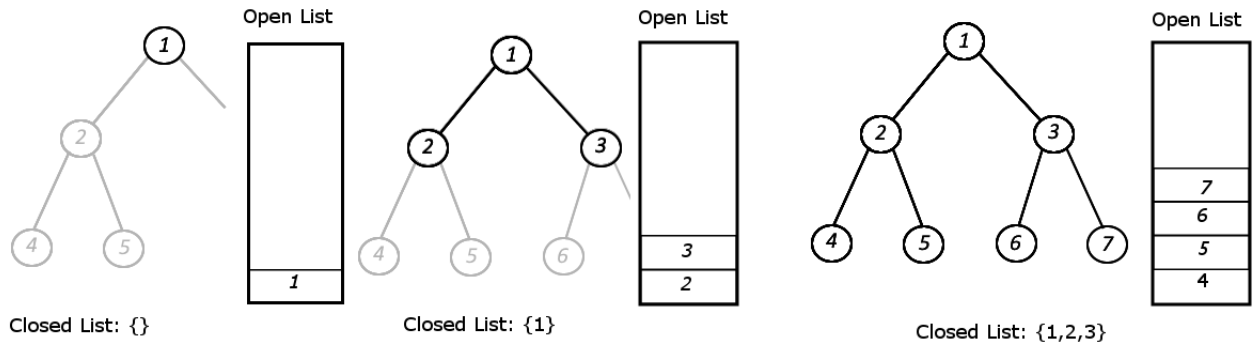

Abbildung 2: Standard Breitensuche

Breitensuche ist eine simple Strategie [15], in welcher der Wurzelknoten als erstes expandiert wird, dann alle Kindknoten und dann die Nachfolger der Kindknoten. Hierbei gilt, dass alle Knoten einer bestimmten Tiefe n ($d \in \mathbb{N}$) expandiert werden, bevor ein Knoten der Tiefe $d + 1$ expandiert wird.
In den meisten Spielbaum Szenarien entsteht der Suchbaum aus einem Graph. Man betrachtet jeden Knoten im Baum genau einmal. Um dies realisieren zu können, benötigt man eine Datenstruktur um zu überprüfen, ob ein Knoten bereits gesehen wurde. Diese Datenstruktur teilt sich auf in *open und closed list*. Die *closed list* beinhaltet alle schon bearbeiteten Knoten. Dieser Ansatz funktioniert aber nur in deterministischen Umgebungen.
Neu erzeugte Knoten, welche sich schon in der *closed list* befinden, können direkt verworfen werden.
Breitensuche wird im einfachsten Sinne mit einer *First in First out queue (im weiteren FIFO genannt)* realisiert, um sicher zu stellen, dass erst alle Knoten auf der Tiefe d abgearbeitet werden und nicht Knoten der Tiefe $d + 1$.
Hier ein Beispiel:

Schritt 1

Knoten *1* wird gesehen und in die *open list* zur weiteren Verarbeitung eingefügt (vergleiche Abbildung 2).

Schritt 2

Nun wird Knoten *1* aus der *open list* genommen und seine Kindknoten in diese geschrieben und dann Knoten *1* in die *closed list* hinzugefügt. Damit ist dieser Knoten bearbeitet, da sich so (nun) alle seine Kindknoten in der *open list* befinden.

Schritt 3

Da die *open list* als *FIFO* realisiert ist, wird nun Knoten *2* entnommen und seine Kindknoten *4 und 5* in die *open List* übernommen. Damit ist Knoten *2* fertig bearbeitet und wird in die *closed*

list geschrieben.
Als nächstes würde nun Knoten *3* zur Bearbeitung aus der *open list* entnommen werden.

2.5 Tiefensuche

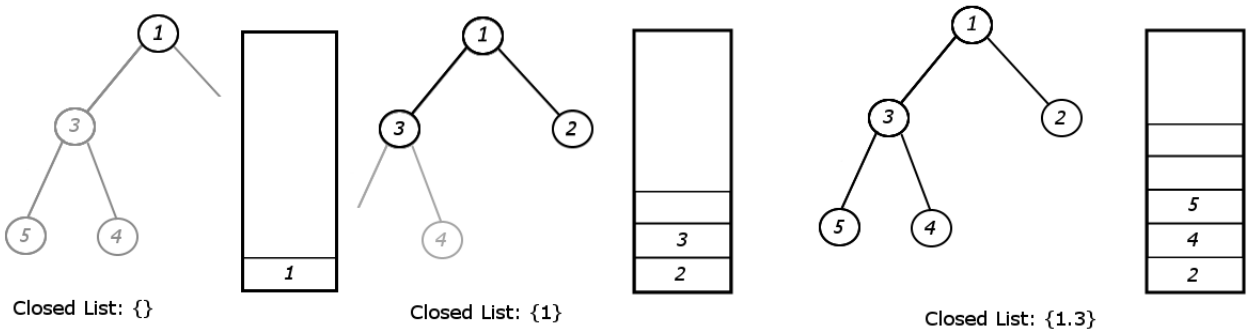

Abbildung 3: Standard Tiefensuche

Tiefensuche expandiert immer den tiefsten Knoten im Baum als erstes [15]. Somit ist gegeben, dass man direkt zur tiefsten Ebene des Baumes gelangt. Hierbei gilt, dass der tiefste Knoten keinen Nachfolger besitzt. Bei Erreichen eines solchen Knotens wird dieser verworfen und man geht auf die nächst höheren Knoten zurück, welche noch weitere Nachfolger besitzen.
Im Gegensatz zur FIFO wird bei Tiefensuche eine *Last in First out (im weiteren LIFO genannt)* verwendet. Unter LIFO versteht man eine Datenstruktur, aus der zuletzt erzeugte Knoten als erstes expandiert werden. Dies muss also der momentan tiefste Knoten sein, denn er ist eins tiefer als sein Elternknoten. Dies gilt, da der Elternknoten der zuletzt tiefste Knoten im Baum war als er ausgewählt wurde.
Hier ein Beispiel:

Schritt 1
Knoten *1* wird gesehen und in die *open list* zur weiteren Verarbeitung eingefügt (vergleiche Abbildung 3).

Schritt 2
Nun wird Knoten *1* aus der *open List* entnommen und seine Kindknoten in diese eingefügt. Damit ist Knoten *1* fertig bearbeitet und wird in die *closed list* übernommen.

Schritt 3
Da die *open list* als *LIFO* realisiert ist, wird nun Knoten *3* aus der Liste entnommen und seine Kindknoten in diese geschrieben.
Knoten *3* ist demnach fertig bearbeitet und wird in die *closed list* übernommen. Nun wird Knoten *5* als nächstes bearbeitet.

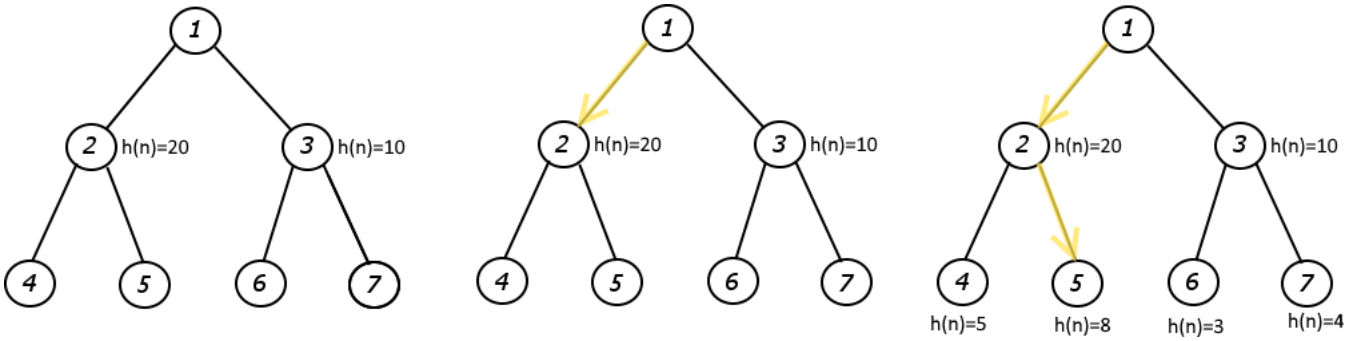

Abbildung 4: Standard Best First Search

Best First Search [15], oder auch *Greedy Best First Search* genannt, verfolgt eine Suchstrategie, welche versucht den Knoten zu expandieren, der nahe am Ziel liegt. Dieser ist nämlich der Knoten, der die höchste beziehungsweise die schnellste Lösung bietet. Um dies zu gewährleisten, werden die Knoten mit einer heuristischen Funktion evaluiert, diese Funktion lautet: $f(n) = h(n)$.
Hierbei steht und fällt die Effizienz von *Best First Search* mit der heuristischen Funktion. Je besser diese gewählt wird, desto schneller wird eine Lösung gefunden.
In dieser Arbeit wird *Best First Search* als Graph-Suchalgorithmus verwendet, welcher einen Spielbaum aufspannt. Damit ist *Best First Search* in diesem Szenario Komplett und die Zeit, die sie benötigt wird, ist im *Worst Case* $O(b^m)$, wobei m die maximale Tiefe im Suchraum ist. Im Fall dieser Arbeit ist dieser immer endlich.

Schritt 1

Knoten 1 wird in die *open list* hinzugefügt und seine Nachfolger werden initialisiert und mit ihrem Schritt bewertet.

Iteration 2

Knoten 1 wird aus der *open list* entnommen und in die *closed list* hinzugefügt. Dann wird der Nachfolger mit der besten heuristischen Bewertung der *open list* hinzugefügt, in diesem Fall: Knoten 2 mit einem Wert von 20.

Schritt 3

Knoten 2 wird aus der *open list* entnommen und in die *closed list* hinzugefügt. Dann wird der Nachfolger mit der besten heuristischen Bewertung der *open list* hinzugefügt, in diesem Fall: Knoten 5 mit einem Wert von 8.

Terminierung

Wenn ein Zielzustand gefunden wurde, wird abgebrochen und der Pfad zu diesem Zustand zurück geliefert.

2.7 Heuristic Search

Heuristic Search (kurz *HS*) traversiert den Spielbaum und selektiert den Knoten, welcher eine heuristische Funktion $h(s) : s \in S$ maximiert. Diese Funktion schätzt die Güte des Zustandes, in den man durch eine bestimmte Aktion gelangt. Nachdem dieser Wert berechnet wurde, wird eine optimale Strategie verfolgt. *HS* pflegt eine *open list* der noch nicht selektierten Knoten und erstellt alle Kindknoten für den Knoten mit dem höchsten, geschätzten, heuristischen Wert. Wenn alle Kindknoten für einen Knoten erstellt wurden, wird dieser aus der *open list* entfernt und nicht weiter in Betracht gezogen.
Sollte der Fall eintreten, dass ein terminaler Knoten gefunden wird, speichert das System den kürzesten Weg zu diesem und das Resultat (Punktestand).
Endet der Algorithmus, da keine Zeit mehr verfügbar oder die *open list* leer sein sollte so wird der kürzeste Weg zum terminalen Zustand gewählt. Abhängig von der Heuristik kann dies schnell zu guten Pfaden führen. *HS* is nicht anwendbar in stochastischen Spielen, da dieser stochastische Effekte nicht beachtet und das ausführen derselben Aktion in einem Zustand, in unterschiedlichen Kindknoten resultieren kann.

2.8 Monte Carlo Methode

Monte Carlo hat seine Wurzeln in der statistischen Physik, dort wurde sie benutzt, um eine Approximation schwieriger Integrale zu erhalten. Diese Approximation ist ebenso in der Spieltheorie nützlich. Hierbei ist die Idee, mit vielen gleich verteilten Zufallsvariablen eine approximierte Lösung für ein Problem zu erhalten. Das heißt, je mehr Zufallsvariablen verwendet werden, desto genauer wird das Ergebnis.
Hierbei geht MCTS nach folgendem Schema vor (siehe dazu Abbildung 5).
Dieses Schema ist wichtig, um das Framework und die Änderungen an den im Framework

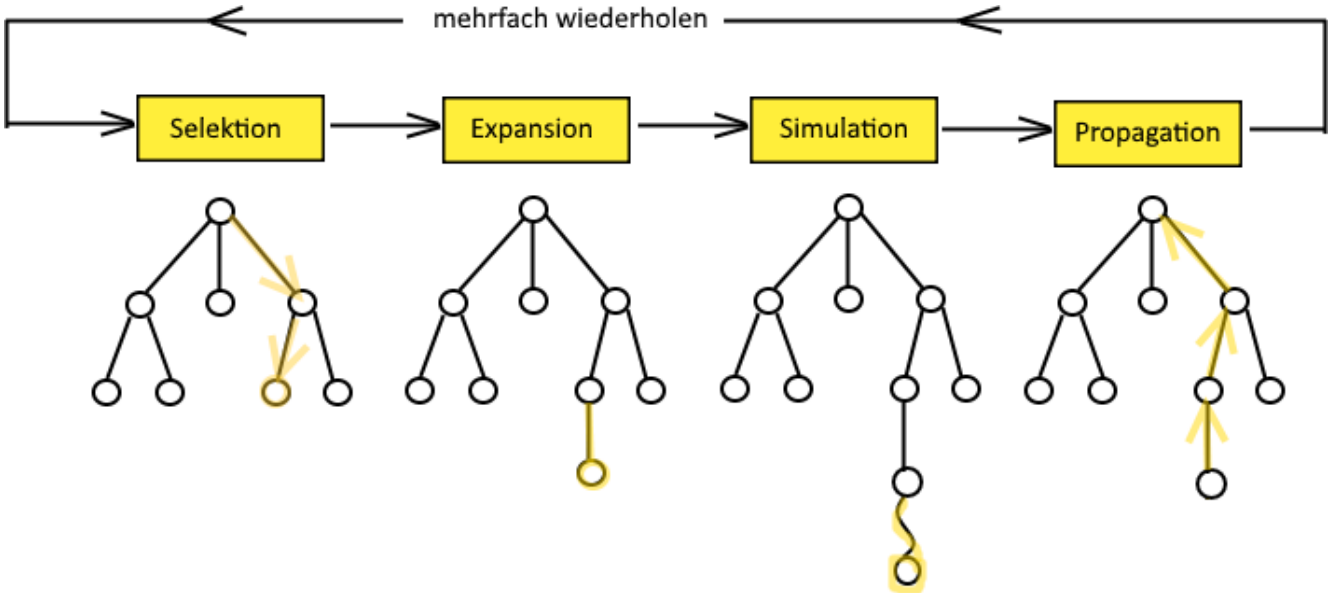

Abbildung 5: Vier Schritt Iteration: Selektion, Expansion, Simulation und Backpropagation. Ein Knoten pro Schritt

zur Verfügung gestellten Algorithmen zu verstehen. Sollten zu dieser Sektion Fragen entstehen, wird auf folgende Literatur verwiesen [4, 12, 14, 18, 11].

Während der Laufzeit des Algorithmus entsteht ein asynchroner, wachsender Spielbaum. Zum Zeitschritt $t = 0$ ist ein Knoten vorhanden, diesen nennt man Wurzel. Dieser Knoten repräsentiert den aktuellen Zustand des Agenten. In jeder weiteren Iteration $t > 0$ wird ein Kind knoten dem Baum hinzugefügt.
Jede Iteration kann also in vier Abschnitte unterteilt werden:

- Selektion
- Expansion
- Simulation
- Backpropagation

Diese vier Schritte werden wiederholt, bis keine Rechenzeit mehr zur Verfügung steht, oder eine andere selbst definierte Abbruchbedingung eintritt.

1. **Selektion**
   Beginnend zu jeder Iteration wird mittels einer *Tree Policy* ein neuer interessanter Knoten im momentanen Spielbaum gesucht, dieser soll dann untersucht werden. Interessant bedeutet in diesem Zusammenhand das folgende:
   a) Dieser Knoten soll eine möglichst hohe erwartete Punktzahl besitzen.
   b) Knoten die im ersten Augenblick nicht optimal erscheinen sollen auch untersucht werden, denn ihre Nachfolgeknoten, können auch vielversprechend sein.
   c) Der Ausgewählte Knoten soll noch mögliche Aktionen die ausgeführt werden können besitzen.

   In jeder Iteration, wird ausgehend vom Wurzelknoten, ein Kindknoten ausgewählt. Diese Auswahl wird unter Beachtung von Exploration und Exploitation getätigt (Vergleich Sektion 2.3.1). Sobald ein Knoten erreicht wird der noch nicht im aktuellen Suchbaum vorhanden ist, wird dieser dem Suchbaum hinzugefügt. Weiterhin ist damit Punkt c erfüllt, dieser besagt das sein Elternknoten noch nicht komplett expandiert wurde.
   Das bedeutet, dass die *Tree Policy* zuständig ist einen Knoten ausfindig zu machen der Expandiert werden soll.

2. **Expansion**
   Die Expansion ist dafür zuständig einen noch nicht voll expandierten Knoten zu expandieren und sein Kindknoten dem Suchbaum hinzu zufügen. Das bedeutet, wenn man einen Knoten $k \in K$ findet mit noch freien Aktionen $a \in A$, kann man mit dem Knoten v und der Aktion a einen neuen Knoten $k' \in K$ erzeugen und dem Baum hinzufügen.

3. **Simulation**
   Um einen Zustand $s \in S$ bewerten zu können benötigt man eine Güte des Zustandes, dies wird über eine Simulation des Zustandes bestimmt. Unter Simulation versteht man das zufällige Auswählen von Aktionen. Man wählt solange Aktionen aus, bis man einen terminalen Zustand erreicht hat, oder eine entsprechende Anzahl an Aktionen ausgewählt hat.

Die Zahl der ausgewählten Aktionen nennt man auch *Rollout Length*, sie ist ein Freiheitsgrad im Algorithmus.
Sollte die Simulation bei nichtterminalen Zuständen stoppen, muss mittels einer Bewertungsfunktion der Zustand bewertet werden. Diese Bewertungsfunktion muss jedem möglichen Zustand eine Güte $V_i(a)$ zuweisen. Wie diese Bewertungsfunktion gewählt wird ist vom Problem, beziehungsweise von der Umgebung abhängig, das heißt, sie muss domänenspezifisch angepasst werden.

4. **Backpropagation**
   Nachdem eine Bewertung $v_i(s,a)$ für den aus der Simulation erhaltenen Zustands $\hat{s}$ errechnet wurde, wird die Bewertung $v_i(s,a)$ ausgehend von s' an die Wurzel weitergegeben. Damit wird jeder Zustand, welcher ein Vorgänger von s' ist auf dem Weg zur Wurzel über den momentanen Spielverlauf in Kenntnis gesetzt.
   Dieses Aktualisieren der Knoten entspricht dem aktualisieren der *Tree Policy*. Dadurch wird die *Selektion* beeinflusst und die Möglichkeit, dass andere Knoten selektiert werden erhöht sich.

2.9 Upper Confidence for Trees - UCT

Eine Problemstellung in *MCTS* ist Selektieren eines des nächsten Knotens im Baum. Da man nicht immer den besten Knoten selektieren möchte sondern auch nicht optimale Knoten. Nicht optimale Knoten zu expandieren ist wichtig, da Nachfolgezustände dieser Knoten eine bessere Lösung enthalten können als der momentane optimale Knoten, steht man vor dem Problem der *Exploration und Exploitation*.
Eine beliebte Methode um eine Balance zwischen den beiden zu schaffen ist *Upper Confidence for Trees (kurz UCT)*.
Die UCT Formel leitet sich aus der *UCB* Formel ab, welche durchschnittlichen Gewinn als Explorationsterm nutzt [11].
Der Begriff *UCB* steht für *Upper Confidence Bound*, also obere Konfidenzschranke. Weiterhin ist *UCB* eines der ersten Verfahren, welches die Konfidenzintervalle zur Berechnung unendlicher Wiederholungen eines Zufallsexperiments mit einer gewissen Häufigkeit verwendet.
Hierbei wurde die *UCB* Formel sofern verändert, dass man die Belohnung in das Intervall $[0,1]$ eingeschränkt hat und eine Konstante zum Gewichten des Explorationsterms hinzufügte. Die *UCT* Formel lautet wie folgt:

$$UCT = V_i(s') + C \cdot \sqrt{\frac{\ln n}{n_i}}$$

wobei $n = \sum_{i=0}^{K} n_i$ die Gesamtanzahl aller gesehenen Spiele ist, welche während der Selektionsphase durch den Knoten s' liefen. Für die konstante C empfiehlt sich der Wert $\frac{1}{\sqrt{2}}$. Da für ihn Konvergenz bewiesen wurde. *UCT* hat den Vorteil das sie trotz der nicht gleich verteilten Aktionswahl eine gleichstarke Konvergenz hat.

3 Spielbaum Algorithmen im Framework

In diesem Kapitel werden die im Framework benutzten Algorithmen und die Änderungen die an ihnen durchgeführt wurden beschrieben.

3.1 Unified Single Player Game Search

In diesem Abschnitt wird beschrieben, wie die im Framework benutzten Algorithmen angepasst werden mussten um Modularität erzielen zu können und was man unter *Single Player Game Search* versteht.
Single Player Game Search befasst sich mit dem bestimmen einer *Tree Policy* $\pi^*(a|s_0)$ für den momentanen Spielzustand s_0. Dies wird erreicht indem man $\pi^*(a|s_0) = argmax_a \sum_{s'} \delta(s'|s,a)V^*(s')$ findet. Die Suche selber verwaltet hierbei die Werte $V_i(s)$ und $v_i(s,a)$, wobei $v_i(s,a)$ bestimmt wie wichtig das Verbessern der Schätzung $\sum_{s'} \delta(s'|s,a)V(s')$ ist. Hierbei wird zwischen deterministischen und stochastischen Umgebungen unterscheiden(siehe dazu 2.1).
Wenn man beachtet das:

$$\exists s' \delta(s'|s,a) = 1$$
$$\forall s'' \neq s' \delta(s''|s,a) = 0$$

Somit gilt für deterministische Spiele:

$$\sum_{s'} \delta(s'|s,a) f(s') = f(s') : s \in S$$

Durch diese Änderungen können nun alle Algorithmen auf vier Schritte reduziert, beziehungsweise vereinheitlicht werden:

1. Init: Bestimmt,durch das Setzen von initialen Werten $v_0(s,a)$, wie der nächste Knoten ausgewählt/besucht wird, ohne vorheriges Wissen über ihn zu haben.
2. Estimate: Bestimmt einen Schätzwert der Belohnung des Knotens $V_0(s)$, wenn er das erste Mal besucht wird.
3. Propagation: Stellt sicher, dass alle Elternknoten über den neuen Schätzwert benachrichtigt werden.
4. Update: Stellt sicher, dass alle Elternknoten über die Güte des neuen Knotens benachrichtigt werden.

Mittels dieser vier Schritte und das nicht Benutzen der *open* und *closed list*, sondern einer Matrix $v(s,a)$, arbeiten alle Algorithmen auf einem einheitlichen Datensatz.

3.2 Baumtraversierung

In diesem Abschnitt wird die Herangehensweise der Baumtraversierung beschrieben.
Den aktuellen Spielbaum zu traversieren und zu bestimmen, welcher Knoten als nächstes dem Baum hinzugefügt werden soll, ist ein Problem der Spielbaumsuche und wird in diesem Framework wie folgt gelöst.

$$s_{i+1} = argmax_a v_i(s_i, a)$$

Um zu erfahren welches Kind als nächstes untersucht werden soll, wird berechnet, welches dieser Kindknoten den höchsten Gütewert hat. Nachdem man es identifiziert hat, wird dieses ausgewählt und untersucht. Dies wird solange wiederholt, bis man einen Knoten gefunden hat der:

1. keine Kindknoten besitzt, oder
2. ein terminaler Zustand ist.

Im ersten Fall, werden die Kindknoten des Knotens erzeugt und mittels der *init* Methode initialisiert. Nachdem dies geschehen ist wird der Knoten zum weiteren untersuchen ausgewählt.
Im zweiten Fall, wird das Traversieren abgebrochen und der Knoten zur wiederholten Untersuchung zurückgeliefert.

3.2.1 Breitensuche im Framework

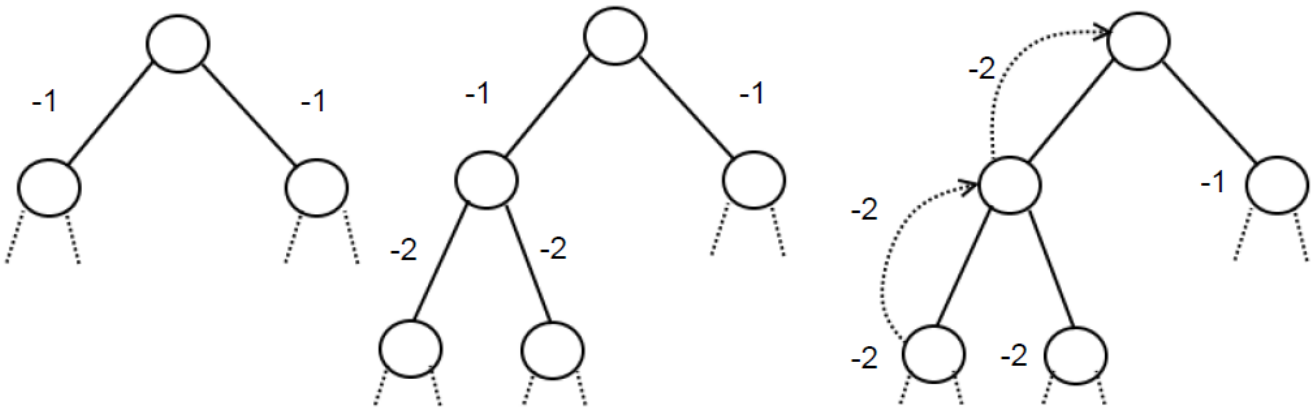

Abbildung 6: Breitensuche ohne closed-/open List

Standard *Breitensuche* benutzt wie Kapitel 2.4 erklärt eine *open* bzw. *closed list* zum verwalten der nächsten Züge.
Im Framework wird diese nicht mehr benutzt, statt dessen wird hier eine *State Action Matrix* $v(s,a)$ verwendet. Diese speichert zu jeder Aktion die Wichtigkeit des Zustandes den man mit der gegebenen Aktion erreichen kann. Im Fall von *Breitensuche* ist es die Tiefe des Knotens im Baum.
Neue Knoten werden mit der negativen Tiefe in der Matrix aktualisiert, dies wird für die Selektion benötigt.
Die Matrix wird am Ende einer Iteration aktualisiert. Im Fall von *Breitensuche* wird bei der Selektion nur der maximale Wert genommen und darüber ermittelt, welchen Knoten man als nächstes besuchen muss.

- Init: $\forall s \in S; v_0(s,a) = -\sum_{s'} \delta(s'|s,a)\phi_t(s')$ mit $\phi_t(s)$ als Zeitschritt von Zustand s.
  Hierbei ist gemeint, dass bei Erstbesuch eines Knotens im aktuellen Spielbaum S alle möglichen Aktionstransitionen mit ihrer negativen Tiefe im Baum Initialisiert werden.
- Reward: $V_0(s) = r(s)$
  Setzen der voraussichtlich zu erreichenden Belohnung im aktuellen Zustand. Dies kann in Form von Punkten oder anderen Features passieren.

- Propagation: $V_{i+1}(s) = r(s) + max_a \sum_{s'} \delta(s'|s,a)V_i(s')$
  Aktualisieren der zu erwartenden Langzeitbelohnung vom momentanen Zustand s_i bis zur Wurzel s_0.
- Update: $v_{i+1}(s,a) = \sum_{s'} \delta(s'|s,a)max_{a'} v_i(s',a')$
  Aktualisieren der Güte der Aktionstransitionen zum Bestimmen der nächsten Expansion, in diesem Fall wird das Maximum aller Kindknoten gewählt.

1. **Schritt**
   Beginnend mit dem Wurzelknoten k_0, werden seine Kindknoten erzeugt und mit ihrer negativen Tiefe im Baum initialisiert. Diese Initialwerte werden in der *State Action Matrixx* $v(s,a)$ gespeichert.

2. **Schritt**
   Nun wird der Knoten mit dem momentan maximalen Gütewert ausgewählt und weiter untersucht. In diesem Fall ist der Gütewert von k_1 und k_2 gleich -1, daher wird hier Knoten k_1 ausgewählt und seine Kindknoten erzeugt und initialisiert.

3. **Schritt**
   Nun wird der neue Gütewert von k_1 auf die maximal zu erreichende Tiefe aktualisiert und an die Elternknoten propagiert. Der neue Wert ist in diesem Fall -2. Damit wird in der nächsten Iteration Knoten k_2 zur Expansion ausgewählt, da er den momentan maximalen Gütewert besitzt, nämlich -1.

Der wesentliche Unterschied der hier implementierten Breitensuche zur vorgestellten Standard Breitensuche (vergleiche Kapitel 2.4) ist, dass auf eine *open-/closed list* verzichtet wird. Diese Entscheidung wurde getroffen, um *Breitensuche* auf die Struktur des Frameworks anzupassen und modularisierbar zu machen.

3.2.2 Tiefensuche im Framework

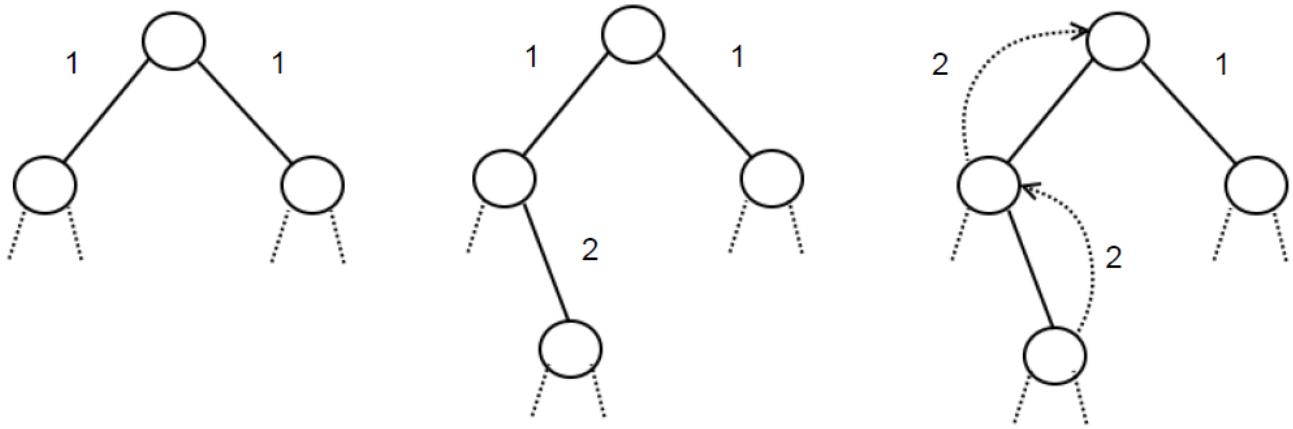

Abbildung 7: Tiefensuche ohne closed-/open List

Standard *Tiefensuche* benutzt wie Kapitel 2.5 erklärt eine *open* bzw. *closed list* zum verwalten der nächsten Züge.
Im Framework wird diese nicht mehr benutzt, anstatt dessen wird hier eine *State Action Matrix* $v(s,a)$ verwendet. Diese speichert zu jeder Aktion die Wichtigkeit des Zustandes, den man mit der gegebenen Aktion erreichen kann.
Im Fall von *Tiefensuche*, ist es die Tiefe des Knotens im Baum.
Neue Knoten werden mit der Tiefe in der Matrix aktualisiert, dies wird für die Selektion benötigt. Die Matrix wird am Ende einer Iteration aktualisiert. Im Fall *Tiefensuche*, wird bei der Selektion nur der maximale Wert genommen und darüber ermittelt, welchen Knoten man als nächstes besuchen muss.

- Init: $\forall s \in S; v_0(s,a) = \sum_{s'} \delta(s'|s,a)\phi_t(s')$ mit $\phi_t(s)$ als Zeitschritt von Zustand s.
  Alle möglichen Aktionstransitionen vom momentanen Zustand werden bei Erstbesuch des Knoten mit ihrer Höhe initialisiert. Dies unterscheidet sich nur geringfügig zu *Breitensuche*

- Reward: $V_0(s) = r(s)$
  Gleiches Vorgehen wie bei 3.2.1

- Propagation: $V_{i+1}(s) = r(s) + max_a \sum_{s'} \delta(s'|s,a)V_i(s')$ Gleiches Vorgehen wie bei 3.2.1

- Update: $v_{i+1}(s,a) = \sum_{s'} \delta(s'|s,a)max_{a'} v_i(s',a')$ Gleiches Vorgehen wie bei 3.2.1

Schritt 1

Beginnend mit dem Wurzelknoten k_0, werden seine Kindknoten erzeugt und mit ihrer Tiefe im Baum initialisiert. Diese Initialwerte werden in der *State Action Matrix* $v(s,a)$ gespeichert.

Schritt 2

Nun wird der Knoten mit dem momentan maximalen Gütewert ausgewählt und weiter untersucht. In diesem Fall ist der Gütewert von k_1 und k_2 gleich 1, daher wird hier Knoten k_1 ausgewählt und seine Kindknoten erzeugt und initialisiert.

Schritt 3

Nun wird der neue Gütewert von k_1 auf die maximal zu erreichende Tiefe aktualisiert und an die Elternknoten propagiert. Der neue Wert ist in diesem Fall 2. Damit wird in der nächsten Iteration zu k_3 traversiert und dieser zur Expansion ausgewählt, da er den momentan maximalen Gütewert besitzt, nämlich 2.

3.2.3 Best First Search im Framework

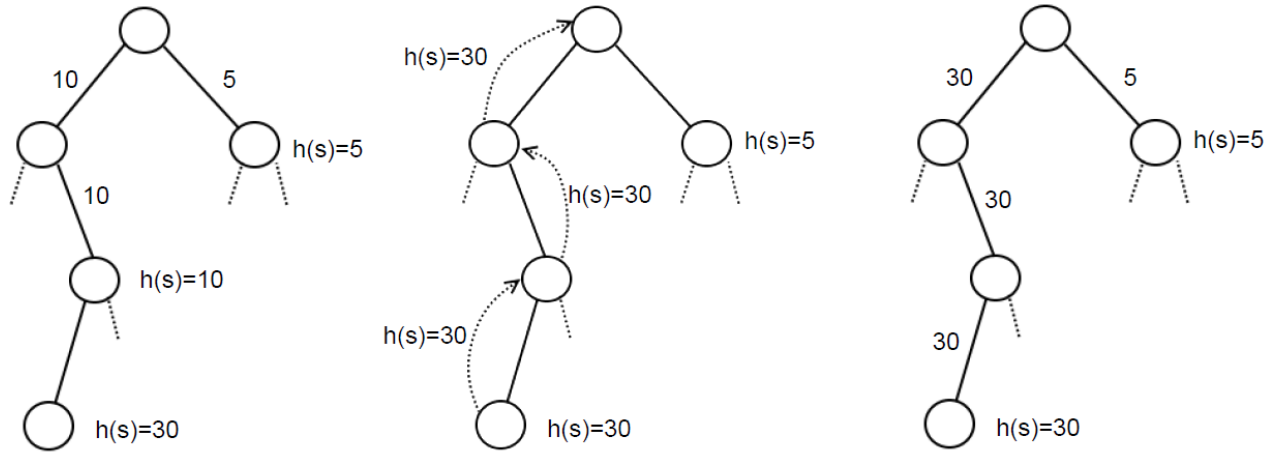

Abbildung 8: Best First Search Beispiel

Die traditionelle *Best First Search* arbeitet mit einer Prioritätsliste, welche die Zustände anhand ihrer Güte priorisiert.
Im Framework wird diese nicht mehr benutzt, anstatt dessen wird hier eine *State Action Matrix* $v(s,a)$ verwendet. Diese speichert zu jedem Aktion die Wichtigkeit des Zustandes, den man mit der gegebenen Aktion erreichen kann.
In diesem Fall wird die Prioritätsliste von der Matrix abgelöst. Hierbei wird für jeden Knoten die Summe der Güte der Kindknoten gespeichert und somit muss bei der Selektion nur noch die Aktion mit der maximalen Güte gewählt werden.

- Init: $\forall s \in S; v_0(s,a) = \sum_{s'} \delta(s'|s,a)h(s')$
  Initialisieren aller möglichen Aktionstransitionen des momentanen Zustands s, bei Erstbesuch des Knotens via heuristischem Wert. Diese Heuristik muss selber definiert werden und ist abhängig von der aktuellen Domäne.
- Reward: $V_0(s) = r(s)$
  Wie bei 3.2.1.
- Propagation: $V_{i+1}(s) = r(s) + max_a \sum_{s'} \delta(s'|s,a)V_i(s')$
  Wie bei 3.2.1.
- Update: $v_{i+1}(s,a) = \sum_{s'} \delta(s'|s,a)max_{a'} v_i(s',a')$
  Wie bei 3.2.1.

Schritt 1

Beginnend mit dem Wurzelknoten k_0, werden seine Kindknoten erzeugt und mit einer vorher definierten Heuristik initialisiert. Diese Initialwerte werden in der *State Action Matrix* $v(s,a)$ gespeichert.

Schritt 2

Nun wird der Knoten mit dem momentan maximalen Gütewert ausgewählt und weiter untersucht. In diesem Fall ist der Gütewert von k_1 größer als von Knoten k_2, daher wird hier Knoten k_1 ausgewählt und seine Kindknoten erzeugt und initialisiert.

Schritt 3

Nun wird der neue Gütewert von k_1 auf den maximal zu erreichenden Gütewert aktualisiert und an die Elternknoten propagiert. Der neue Wert ist in diesem Fall 10. Damit wird in der nächsten Iteration Knoten k_4 zur Expansion ausgewählt, da er den momentan maximalen Gütewert besitzt, nämlich 10.

3.2.4 Heuristic Search im Framework

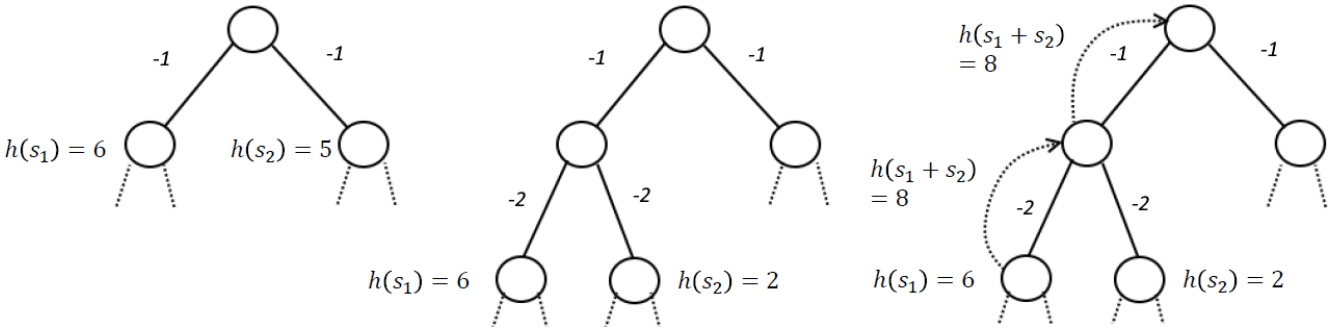

Abbildung 9: Heuristic search

- Init: $\forall s \in S; v_0(s,a) = -\sum_{s'} \delta(s'|s,a)\phi_t(s')$ mit $\phi_t(s)$ als Zeitschritt von Zustand s.
  wie bei 3.2.1.
- Reward: $V_0(s) = h(s)$
  Zu erreichende Belohnung im aktuellen Zustand, wird über eine Heuristik berechnet. Diese Heuristik muss selber definiert werden und ist abhängig von der Domäne des Spiels.
- Propagation: $V_{i+1}(s) = r(s) + max_a \sum_{s'} \delta(s'|s,a)V_i(s')$
  Wie bei 3.2.1.
- Update: $v_{i+1}(s,a) = \sum_{s'} \delta(s'|s,a)max_{a'}V_i(s)$
  Zum Aktualisieren der Güte eines Zustandes, werden die zu erreichenden Belohnungen der Kindknoten summiert. Damit ist die Güte eines Zustandes abhängig von der zu erreichenden Langzeit Belohnung seiner Kindknoten.

3.2.5 MCTS im Framework

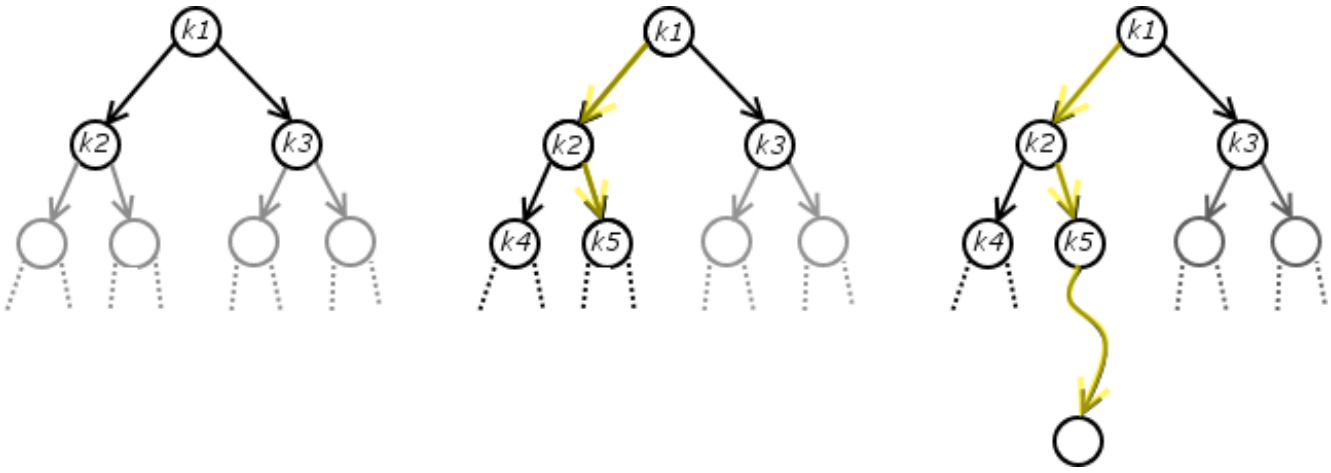

Abbildung 10: MCTS

Der für das Framework modifizierte *MCTS* funktioniert wie in Kapitel 2.8 erklärt. Hierbei wird als erstes ein Knoten aus dem aktuellen Suchbaum ,angefangen beim Wurzelknoten, über die *Tree Policy* ausgewält, bis man einen Knoten ohne Nachfolger findet. Danach wird ausgehend von diesem Knoten eine Simulation durchgeführt. Das Ergebnis dieser Simulation wird an die Elternknoten propagiert. Der einzige Unterschied zwischen dem normalen und dem modifizierten *MCTS* ist, dass die von der *UCT* erzeugten Werte im letzten Schritt für die *Tree Policy* aktualisiert werden.

- Init: $\forall s \in S; v_0(s,a) = M : M > max\, v(s,a)$
  Wenn ein Knoten das erste Mal im momentanen Spielbaum S besucht wird, werden seine möglichen Aktionstransitionen mit dem Wert unendlich (∞) initialisiert.
- Reward: $V_0(s) = \mathbb{E}_{\delta(s_{t+1}|s_t,\pi^r(s_t)} \sum_t^{t+T} r(s_t)\delta(s_{t+1}|s_t,\pi^r(s_t))$ mit rollout policy π^r und Horizont T.
  Die Belohnung wird über einen Rollout mit individueller *Rollout Policy* berechnet. In dieser Berechnung sind Heuristiken enthalten, welche individuell zur Domäne erstellt werden müssen.
- Propagation: $V_{i+1}(s) = r(s) + \sum_{s'} \delta(s'|s,a)V_i(s')$, wobei a die Aktion ist, welche in der letzten Iteration ausgewählt wurde ($max_a\, v_i(s,a)$)
  Die Langzeit Belohnung wird an die Elternknoten weiter gegeben und auf ihren Wert aufaddiert.
- Update: $v_{i+1}(s,a) = \sum_{s'} \delta(s'|s,a)V_i(s') + C \cdot \sqrt{\frac{n(s)}{n(s,a)}}$
  Aktualisieren der Güte des Knotens mittels *UCT* und propagieren dieses Wertes an die Elternknoten.

Die hier implementierte Version von *MCTS* (wie in Abbildung 10) unterscheidet sich kaum von der in Kapitel 2.8 vorgestellten Version: Statt die *UCT-Werte* der Knoten im *Selektionsschritt* zu berechnen, werden die Werte schon in der *Backpropagation* bzw. im *Update* berechnet und aktualisiert.

4 Framework

In diesem Kapitel wird behandelt, wie das Framework aufgebaut ist, welche Datenstrukturen es besitzt und wie man es benutzt.

4.1 State Action Matrix

In den meisten deterministischen Spielbaum Algorithmen (siehe 2.4, 2.5) wird eine *open* bzw. *closed list* verwendet um die Selektion des nächsten Knotens zu ermitteln.
Wie schon in Kapitel 2.4 und 2.5 erläutert, unterscheiden diese beiden Listen sich durch ihre Datenstruktur. Um ein einheitliches Framework zu schaffen, müssen also auch die Datenstrukturen angeglichen werden. Deshalb wird eine *State Action Matrix* verwendet, welche für jeden momentanen Zustand $s_i \in S$ allen möglichen Aktionen einen geschätzten Gütewert zuordnet. Durch richtiges Beschreiben der Werte kann nun im Selektionsschritt auf den maximalen Wert zugegriffen werden. Mit dieser Änderung kann ohne *open* bzw. *closed list* gearbeitet werden, was das Vereinheitlichen der Algorithmen möglich macht.
Die *State-Action Matrix* ist wie folgt definiert:

$$v(s,a) \rightarrow \mathbb{R} : s \in S, a \in A$$

Das bedeutet, wenn man der Matrix ein Tupel aus Zustand und Aktion übergibt, dass sie den geschätzten Gütewert für die Aktionstransaktion in den Zustand $s' \in S$ zurück liefert.
Dieser Wert wird ohne Vorwissen über den nächsten Zustand initialisiert. Die Initialisierung der Werte ist abhängig von der aktuell ausgewählten *init* Funktion. Weiterhin ist das Aktualisieren der Werte mittels der *update* Funktion ausschlaggebend, welcher Knoten in der nächsten Iteration selektiert wird. Hier ein Beispiel:

1. *init* Funktion von *BFS*
2. *reward* Funktion von *BFS*
3. *propagation* Funktion *BFS*
4. *update* Funktion von *MCTS*

Zu Beginn ist jede Transaktion des *Wurzelknotens* mit der negativen Tiefe im Baum initialisiert. Dann wird ein Knoten aus den Kindknoten gewählt. Dies passiert anhand des maximalen Wertes aus der Matrix $v(s,a)$, dieser ist momentan die negative Tiefe im Baum.
Wenn nun ein Knoten selektiert wird, werden seine Kindknoten erzeugt und initialisiert.
Als nächstes wird der Knoten anhand seines Zustandes bewertet. Dies kann auf viele Arten passieren und ist vom Nutzer selber zu definieren.
Nach dem Bewerten des Zustandes wird die Bewertung zur Wurzel propagiert und mit den Bewertungen der Elten summiert.
Die Güte des Knotens wird nun auch neu berechnet Dies passiert anhand der *UCT* Formel von *MCTS*.
Das hat zur Folge, dass *BFS* nicht mehr beim Selektieren anhand der geringsten Tiefe die Auswahl trifft, sondern anhand der *UCT*-Formel.

4.2 Abstract Class State

Damit das Framework in mehreren Domänen einsetzbar ist, wird eine abstrakte Klasse mit dem Namen *State* zur Verfügung gestellt. Diese Klasse repräsentiert einen allgemeinen Spielzustand und beinhaltet Methoden, welche in einer konkreten Zustandsklasse implementiert werden müssen. Hier eine Übersicht der zu implementierenden Methoden.

- *int getId()*: Diese Methode muss einen eindeutigen Identifier des Zustandes zurück liefern. Jeder Zustand besitzt eine eindeutige *ID*, welche zur genauen Identifikation dient, dies ist wichtig um seine Position in der *State-Action Matrix* zu finden.
- *State next(Object action)*: Diese Methode erzeugt anhand der übergebenen Aktion und dem aktuellen Zustand $s \in S$, einen neuen Zustand, in den man gelangt, wenn man *action* in Zustand *s* ausführt. Hierbei muss man beachten, dass das Framework auf Zuständen mit einem *Forward Model* arbeiten kann (siehe dazu Kapitel 2.1).
- *double reward()*: Diese Methode liefert die (heuristische) Belohnung des aktuellen Zustandes zurück.
- *Object[] getPossibleActions()*: Diese Methode liefert eine Liste der möglichen Aktionen im Zustand zurück.
- *int getIndexOfAction(Object action)*: Diese Methode liefert den zugehörigen Index der übergebenen Aktion, in der Liste aller Aktionen.
- *Object getActionofIndex(int index)*: Diese Methode liefert die Aktion an der Position *index* aus der Liste aller möglichen Aktionen. Diese Position sollte immer eindeutig sein.
- *public boolean isGameOver()*: Diese Methode liefert *true*, wenn das Spiel zu Ende ist. Ansonsten *false*.
- *public State copy()*: Diese Methode kopiert den momentanen Zustand.
- *public boolean isWinState()*: Diese Methode liefert *true* zurück, falls der Spieler gesiegt hat, ansonsten *false*.

Diese Methoden müssen in konkreten Zuständen implementiert werden, damit das Framework korrekt funktionieren kann.
Ein abstrakter Zustand ist wichtig, da jede Domäne eigene Zustände definieren muss. Ohne diese abstrakte Klasse müsste man das Framework neu anpassen, wenn die Domäne gewechselt wird.

5 Proof of Concept

In diesem Kapitel wird gezeigt, dass die Algorithmen die im Framework zur Verfügung gestellt werden, in der Anwendung funktionieren.

5.1 Proof Setup

Getestet wurden *Breitensuche, Tiefensuche, Monte Carlo Tree Search* und *Breitensuche* mit der *MCTS Reward Funktion*. Die Testdomäne ist das Spiel *Bait* des *General Video Game Artificial Intelligence (kurz GVGAI)* Frameworks. Um das Spiel *Bait* zu lösen, muss man an den Schlüssel

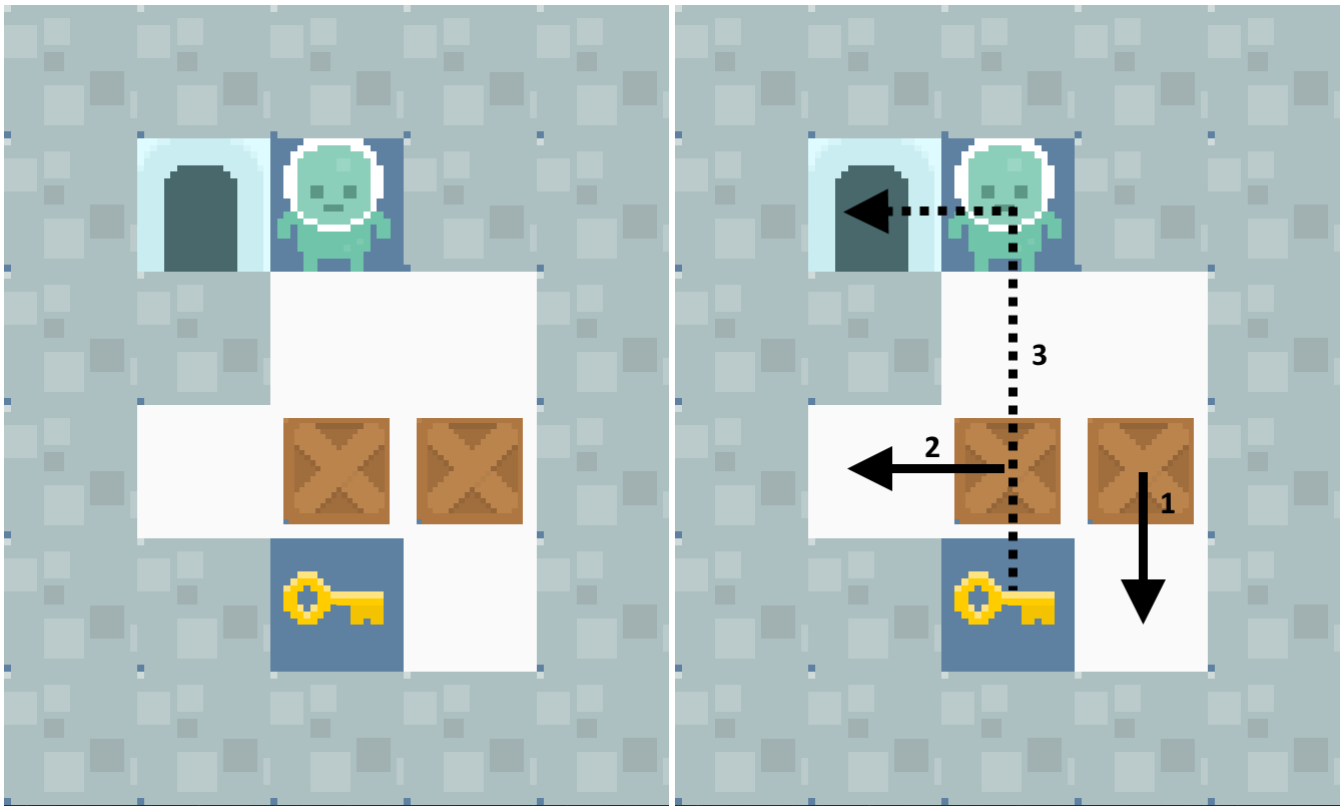

Abbildung 11: Bait Level 0 inklusive Lösung

gelangen und mit diesem die Tür erreichen. Dies wird erschwert durch bewegliche Objekte (Kisten). Hierbei ist es sehr leicht in den ersten zwei Zügen einen sogenannten *Trap State* zu kreieren, der das Spiel unlösbar macht, da die Kiste auf den Schlüssel geschoben werden kann. Weiterhin erlangt man in diesem Spiel nur eine Belohnung, wenn man den Schlüssel in das Tor bringt und damit das Spiel löst.
Die Optimale Lösung ist wie folgt:

1. Die rechte Kiste muss nach unten bewegt werden (vergleiche Abbildung 11)
2. Die rechte Kiste in die rechte Ecke (vergleiche Abbildung 11)
3. Der Schlüssel muss aufgenommen und in das Tor gebracht werden.

Diese Schritte kann man nicht vertauschen und es gibt auch keine andere Lösung.

5.2 Die Algorithmen

Zwischen den hier getesteten Algorithmen *Breitensuche, Tiefensuche und Monte Carlo Tree Search* gibt es diverse Unterschiede. Die hauptsächlichen Unterschiede sind wie folgt:

- Informationsgewinnung der zu erreichenden Belohnung in Zuständen. *Breitensuche und Tiefensuche* bewerten Zustände im Suchbaum anhand von vorhandenen Kindzuständen. *MCTS* hingegen bewertet anhand von Simulationen.
- Unterschiedliche Update Schritte: *MCTS* nutzt *UCT* um nicht nur zu exploiten, sondern auch zu explorieren.

5.3 Setup der Algorithmen

Um gleiche Bedingungen zu schaffen wurden alle Algorithmen mit dem gleichen Spiel getestet. Weiterhin wurde als Belohnungsfunktion der zu erreichende Punktestand zu einem Zustand genommen. Sollte der Agent einen terminalen Zustand erreichen indem er gewinnt, wird eine Belohnung von 1000 auf den Punktestand addiert.
Hierbei ist zu beachten, dass die *Score/Punkte* nur ganzzahlig sind und 1000 ein immenser Wert im Vergleich zu den erreichenden Punkten ist.

$$r(s) = \begin{cases} \text{score} + 1000 & \text{if s is win state} \\ \text{score} & \text{else} \end{cases}$$

Weiterhin zu beachten ist, dass diese Art der Belohnung eine sehr simple Strategie ist und sich deutlich verbessern lässt. Für einen Funktionstest ist es allerdings ausreichend.

5.4 Ergebnisse

Alle Algorithmen wurden jeweils 100 (einhundert) mal unter gleichen Voraussetzungen getestet. Die Ergebnisse sind wie folgt:

Algorithmus	Win	Lose
Breitensuche	32	68
Tiefensuche	43	57
Monte Carlo Tree Search	71	29
BFS + MCTS reward	70	30
MCTS + BFS reward	39	61

Tabelle 1: Ergebnisse von 100 Spielen Bait, Level 0

Wie man in der Tabelle erkennen kann, schneiden die Standard Spielbaum Suchalgorithmen (*Breitensuche, Tiefensuche)* am schlechtesten ab. Dies lässt sich dadurch erklären, dass sowohl *Breitensuche* als auch *Tiefensuche* nach Ablauf der Rechenzeit für einen Zug die beste Aktion wiedergeben und nicht einen besten Weg planen, der Sie zum Ziel führt. Weiterhin existiert im Spiel *Bait*, wie in Kapitel 5.1 erwähnt, ein *Trap State*. Damit ist gemeint, dass der Agent durch

eine falsche Aktion das gesamte Spiel unlösbar machen kann.
Dieser *Trap State* wird durch den *Rollout* von *MCTS* besser erkannt und kann umgangen werden. Das bedeutet, dass durch den Einsatz des *Rollout* des *MCTS-reward Moduls*, in einfachen Setups ein größerer Ausschlag in der Gewinnchance erzielt werden kann.
Dies lässt sich durch den Test von *BFS + MCTS reward* belegen. Da die Gewinnchancen von *BFS* sich durch den Einsatz des *MCTS Rollouts* um 38% gesteigert hat.
Um die Gewinnchancen der Algorithmen weiter zu steigern, müssen bessere *Heuristiken* entwickelt werden, dies ist aber je nach Domäne unterschiedlich und dementsprechend ein Freiheitsgrad des Nutzers.
Die Testergebnisse haben gezeigt, dass die Algorithmen wie erwartet, in ihrer Grundform funktionieren. Weiterhin konnte man zeigen, dass dem Nutzer des Frameworks ein Mehrwert durch die Modularität geboten wird. Das soll heißen, dass die Algorithmen durch die Möglichkeit des schnellen Austauschen der Module ein gleich gutes, beziehungsweise ein besseres Ergebnis liefern.

5.4.1 Breitensuche versus Tiefensuche

Neben der klar unterschiedlichen Gewinnchance zwischen *Breiten-* beziehungsweise *Tiefensuche* und *MCTS* von $30-40\%$, ist noch ein geringerer Unterschied zu erkennen. Wie man in der obigen Tabelle erkennen kann, existiert schon ein Gewinnchancenunterschied von 11%, zwischen *Breiten-* und *Tiefensuche*. Dieser Unterschied ergibt sich durch die zwei verschiedenen Ansätze der beiden Algorithmen. Da *Breitensuche*, wie der Name schon indiziert, erst in die Breite geht, ist die Chance das eine Lösung vor dem Entstehen des *Trap States* zu finden, sehr gering. Dies lässt sich dadurch erklären, dass die hier implementierte beziehungsweise getestete *Breitensuche* nicht auf identische Zustände achtet und nicht auf gleiche Zustände prüft benutzt.
Durch diesen Aspekt hat *Tiefensuche* einen geringen Vorteil. Da der *Zustandsraum* von *Bait* Level null sehr klein ist, ist die Chance, dass ein terminaler Zustand indem der Agent gewinnt höher, als im Fall der *Breitensuche*. Dennoch ist festzuhalten: beide Algorithmen mit der simplen Herangehensweise weniger als 50% Gewinnchance haben.

5.5 Rollout versus kein Rollout

Algorithmus	Win	Lose
BFS + MCTS reward (Rollout)	70	30
MCTS + BFS reward (Keine Rollouts)	39	61

Tabelle 2: Rollout versus Kein Rollout

Wie man in der oben abgebildeten Tabelle erkennen kann, ist die Gewinnchance abhängig von dem Ansatz der *Rollout* Strategie von *MCTS* (vergleiche dazu 2.8).
Diese Tatsache lässt sich dadurch erklären, dass im *Reward Modul* von MCTS Nachfolgezustände bis zu einer Tiefe $n \in \mathbb{N}$ simuliert werden und der daraus resultierende Zustand bewertet wird. Dies hat zur Folge, dass bei einem kleinen Zustandsraum ein Zustand indem der Agent gewinnt schneller gefunden werden kann (im weiteren *Winstate* genannt).
Ansätze ohne diese *Rollout* Strategie betrachten und bewerten nur Zustände im Baum, beziehungsweise Zustände, welche direkt in den Baum eingepflegt werden. Daher bewerten sie nur

direkte Nachfolgezustände. Die *Rollout* Strategie hingegen versucht durch den Ansatz der Simulation *n* Schritte in die Zukunft zu machen und den Ausgangszustand mit einem möglichen Spielresultat zu bewerten.
Hierbei ist zu beachten, dass die verwendete *Rollout* Strategie sehr simple ist, was heißen soll, sie verwendet zur Simulation ausschließlich zufällige Aktionen. Hierbei handelt es sich auch um Aktionen, die nach dem Ausführen wieder im gleichen Zustand resultieren.
Dennoch hat dieser Ansatz eine hohe Erfolgschance in Spielen mit kleinem Zustandsraum. Des Weiteren können *Trap States* besser erkannt werden und dementsprechend umgangen werden.

6 Related Work

In diesem Kapitel werden ähnliche Arbeiten vorgestellt und ihr Unterschied zu dieser Arbeit aufgezeigt.
Eine dieser Arbeiten trägt den Namen *Trial-based Heuristic Tree Search for Finite Horizon MDPs* [10].
Dort wird ein Framework vorgestellt, welches sich mit dem Zusammenfassen von Algorithmen befasst. Diese Algorithmen sollen *MDPs* mit großen Zustandsräumen lösen. Folgende Algorithmen werden dort betrachtet:

- AO^*:
  Eine nicht deterministische Version des globalen heuristischen Suchalgorithmus A^*
- *RTDP*:
  Ein Asynchroner dynamischer Programmieransatz
- *UCT*:
  Vergleiche Kapitel: 2.9

Hierbei wurde festgestellt, dass die oben genannten Algorithmen in drei alternierende Phasen unterteilt werden können. Nämlich in *Selektion, Expansion* und *Backup*, diese Unterteilung ähnelt den Schritten von *MCTS* und wurde auch in dieser Arbeit verwendet. Nun werden die Schritte in fünf Unterschritte zerlegt, diese lauten wie folgt:

- *Action selection*: Welche Aktion soll im aktuellen Zustand gewählt werden.
- *Outcome selection*: Wie hoch ist die Chance das diese gewählt wird.
- *Trial length*: Wie viel Zeit ist noch übrig bevor eine Entscheidung getroffen werden muss.
- *Heuristic function*: Eine heuristische Funktion.
- *Backup function*: Eine Funktion zum Aktualisieren der Elternknoten

Diese Herangehensweise ähnelt der in dieser Arbeit vorgestellten Herangehensweise, d.h. beide Arbeiten haben verschiedene Algorithmen auf eine *MCTS*-artige Struktur zurückgeführt. Diese Struktur wurde benutzt, um verschiedene Algorithmen in einem Framework zu vereinen. Der Ausschlaggebende Unterschied zwischen den Ansätzen ist, dass in der hier vorgestellten Arbeit Verschiedene Algorithmen modularisiert wurden und nicht auf eine allgemeine Performanz im Vordergrund lag. Sondern die Möglichkeit der einfachen Kombination verschiedener Algorithmen.

7 Future Work

In diesem Kapitel wird erläutert, wie man dieses Framework weiter entwickeln kann und wie weitere Arbeiten daran getätigt werden können.
Durch das modularisieren der Algorithmen (Breitensuche, Tiefensuche, BFS, HS und MCTS), kann der Nutzer des Frameworks schon mit einfachen Mitteln verschiedene Algorithmen schnell nutzen. Eine weitere Möglichkeit das Framework zu erweitern, besteht darin mehr Algorithmen zu implementieren und sie damit schnell zugänglich zu machen.
Eine zusätzliche Erweiterung wären mehr *Rollout Policies*, welche nicht mit Zufälligen Aktionswahlen arbeiten. Dies würde die Effizienz von *MCTS* beziehungsweise von allen Algorithmen erhöhen, welche ein Modul mit *Rollouts* nutzen. Die Tests haben gezeigt, dass diese Module sehr hohe Gewinnchancen haben und dementsprechend ein Mehrwert darstellen.
Die Möglichkeit des Prunens von gleichen Zuständen kann den Suchaufwand deutlich verringern. Mit *Pruning* ist das Verwerfen von Aktionen, beziehungsweise Teilbäumen gemeint. Dies passiert unter gewissen Umständen zum Beispiel, wenn man nach dem Anwenden einer Aktion wieder in den Ausgangszustand gelangt. Durch *Pruning* würde man Spiele mit großen Zustandsräumen besser beziehungsweise effizienter lösen können.
Zurzeit arbeitet das Framework mit Knoten (Nodes), diese Knoten repräsentieren momentan die Baumstruktur. Man könnte die Knotenstruktur entfernen und nur auf Zuständen arbeiten, da jeder Zustand eine einzigartige Identifikationsnummer besitzt, kann man durch erweitern der *state-action matrix* die Eltern-/Kindeigenschaft der Knoten ersetzen und spart sich das Erstellen der Knoten.

8 Fazit

Es wurde erfolgreich ein Framework implementiert, indem verschiedene Spielbaum Suchalgorithmen implementiert und modularisiert wurden. Hierbei wurden die Algorithmen auf ein allgemeines Schema gebracht, welches aus den folgenden vier Schritten besteht Init, Reward, Propagation und Update. Durch diese Änderung wurde es dem Nutzer dieses Frameworks möglich gemacht, verschiedene grundsätzliche Algorithmen zur schnellen Verfügung und Kombination verwenden zu können. Algorithmen können auf Grund der Modularisierung miteinander kombiniert werden. Auch das Austauschen einzelner Teilschritte hat dazu beigetragen ein positives Resultat zu erzielen. In einem Testlauf wurde gezeigt, dass *MCTS Rollout* Strategien in simplen Implementationen und mit simplen Heuristiken ein sehr mächtiges Werkzeug sind, um optimale Lösungen zu finden.
Hierbei muss man aber beachten, dass dieses Resultat nur bei Spielen mit kleinem Zustandsraum möglich ist. In diesem Test wurde auch gezeigt, dass *Breitensuche* mit den Strategien von *MCTS* eine weit besseres Resultat erzielt als in der Standard Implementation.
Durch das Verallgemeinern der Algorithmen wurde gezeigt, dass es nicht notwendig ist bei *Breitensuche, Tiefensuche, usw.* eine *open* bzw. *closed list* zu verwenden. Hierbei wurde eine Matrix verwendet (*state-action-matrix*), die den Vorteil der Wiederverwendbarkeit bietet, da jeder Algorithmus Gütewerte der Zustände in dieser Matrix speichert und andere Algorithmen auf dieser Matrix auch zugreifen können.
Das hier implementierte Framework erweist sich als unkomplizierte Möglichkeit verschiedene Algorithmen schnell und einfach Nutzern zur Verfügung zu stellen. Um die Ergebnisse der Algorithmen zu verbessern, hat der Nutzer einen hohen Freiheitsgrad. Damit ist gemeint, dass die Heuristiken an die jeweilige Domäne angepasst werden muss. Die momentanen Heuristiken sind sehr simple gehalten und können von Domäne zu Domäne unterschiedlich effizient sein, hierbei ist es aber am Nutzer diese anzupassen und auf die Problemstellung zu erweitern.

Literatur

[1] Pieter Abbeel and Andrew Y Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the twenty-first international conference on Machine learning*, page 1. ACM, 2004.

[2] Bruce Abramson. Expected-outcome: A general model of static evaluation. *IEEE transactions on pattern analysis and machine intelligence*, 12(2):182–193, 1990.

[3] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256, 2002.

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

[5] Weiwei Cheng, Johannes Fürnkranz, Eyke Hüllermeier, and Sang-Hyeun Park. Preference-based policy iteration: Leveraging preference learning for reinforcement learning. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 312–327. Springer, 2011.

[6] Frederick J Dorey. In brief: statistics in brief: confidence intervals: what is the real result in the target population? *Clinical Orthopaedics and Related Research®*, 468(11):3137–3138, 2010.

[7] Johannes Fürnkranz and Eyke Hüllermeier. *Preference Learning*. Springer, 2011.

[8] Johannes Fürnkranz, Eyke Hüllermeier, Weiwei Cheng, and Sang-Hyeun Park. Preference-based reinforcement learning: a formal framework and a policy iteration algorithm. *Machine learning*, 89(1-2):123–156, 2012.

[9] Sebastian Haufe, Daniel Michulke, Stephan Schiffel, and Michael Thielscher. Knowledge-based general game playing. *KI-Künstliche Intelligenz*, 25(1):25–33, 2011.

[10] Thomas Keller and Malte Helmert. Trial-based heuristic tree search for finite horizon mdps. In *ICAPS*, 2013.

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

[12] Marc Lanctot, Mark HM Winands, Tom Pepels, and Nathan R Sturtevant. Monte carlo tree search with heuristic evaluations using implicit minimax backups. In *2014 IEEE Conference on Computational Intelligence and Games*, pages 1–8. IEEE, 2014.

[13] Bruno Lecoutre and Jacques Poitevineau. The significance test controversy and the bayesian alternative. *StatProb: The Encyclopedia Sponsored by Statistics and Probability Societies, http://statprob. com/encyclopedia/SignificanceTestControversyAndTheBayesianAlternative. html, accessed: 2016-09-28*, 2010.

[14] Tom Pepels, Mark HM Winands, and Marc Lanctot. Real-time monte carlo tree search in ms pac-man. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(3):245–257, 2014.

[15] Stuart Jonathan Russell and Peter Norvig. Artificial intelligence: a modern approach (3rd edition), 2009.

[16] Peter Norvig Stuart Russell. *Artificial Intelligence: A Modern Approach*. Pearson Education, 3 edition, 2014.

[17] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[18] Umar Syed and Robert E Schapire. A game-theoretic approach to apprenticeship learning. In *Advances in neural information processing systems*, pages 1449–1456, 2007.

[19] Thomas Thomsen. Lambda-search in game treesŮwith application to go. In *International Conference on Computers and Games*, pages 19–38. Springer, 2000.

[20] Yufan Zhao, Michael R Kosorok, and Donglin Zeng. Reinforcement learning design for cancer clinical trials. *Statistics in medicine*, 28(26):3294–3315, 2009.

[21] Masrour Zoghi, Shimon Whiteson, Rémi Munos, Maarten de Rijke, et al. Relative upper confidence bound for the k-armed dueling bandit problem. In *JMLR Workshop and Conference Proceedings*, number 32, pages 10–18. JMLR, 2014.