Abstrakte Spielbaumsuche für General Game Playing

Abstract Game Tree Search for General Game Playing
Bachelor-Thesis von Tobias Hecker
Tag der Einreichung:

1. Gutachten: Johannes Fürnkranz
2. Gutachten: Tobias Joppen

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
Knowledge Engeneering Group

Abstrakte Spielbaumsuche für General Game Playing
Abstract Game Tree Search for General Game Playing

Vorgelegte Bachelor-Thesis von Tobias Hecker

1. Gutachten: Johannes Fürnkranz
2. Gutachten: Tobias Joppen

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 1. November 2017

(Tobias Hecker)

Zusammenfassung

Suchalgorithmen wie die Breitensuche oder die Bestensuche verwenden einen Suchbaum. Dieser kann, wird die Suche lange genug ausgeführt, sehr viele Knoten enthalten kann. Viele dieser Knoten unterscheiden sich jedoch nur in wenigen Eigenschaften voneinander. Die in dieser Arbeit vorgestellte Cluster Suche fasst ähnliche Knoten zu einem Cluster zusammen und verbindet diese Cluster durch Makro-Operatoren. Somit kann die Größe des Suchbaums stark reduziert werden. Um bewerten zu können, ob dieses Zusammenfassen einen Vorteil gegenüber normalen Suchverfahren bringt, wird die Cluster Suche mit anderen Verfahren verglichen. Dabei wird deutlich, dass die Cluster Suche vergleichbare Ergebnisse im Bezug auf die Performance liefert.

Inhaltsverzeichnis

1 Einleitung **7**
1.1 Aufbau der Arbeit 8

2 General Game Playing **9**
2.1 GVG-AI 9

3 Überblick/Einordnung Suchverfahren **11**
3.1 Markov Decision Process (MDP) 11
3.2 Baumsuche 12
3.2.1 Kosten 13
3.2.2 Redundante Pfade 13
3.2.3 Zielbedingung 13
3.3 Breitensuche 13
3.4 Uniform-Cost Search 14
3.5 Heuristische Suchverfahren 14
3.6 A*-Bestensuche 14
3.7 Suchverfahren mit Makro-Operatoren 15
3.8 Clustering 15

4 Cluster Suche **17**
4.1 Cluster 17
4.2 Initialisierung 18
4.3 Iteration 18
4.3.1 Cluster/Makro-Operator auswählen 18
4.3.2 Makro-Operator expandieren 19
4.3.3 Cluster zuordnen 19
4.3.4 Makro-Operator lernen 19
4.4 Generierung des Lösungspfads 20

5 Experiment **22**
5.1 Implementierung 22
5.1.1 Zielbedingung 22
5.1.2 Clustering 22
5.1.3 Makro-Operatoren 22
5.1.4 Reihenfolge beim Expandieren 23
5.1.5 Abschließende Pfadgenerierung 23
5.2 Auswahl der Spiele 23
5.2.1 Bait 24
5.2.2 Brainman 24
5.2.3 Catapults 24
5.2.4 The Citadel 25

5.2.5 Labyrinth . . . 25
5.2.6 Real Sokoban . . . 25
5.3 Agenten zum Vergleich . . . 25
5.3.1 YOLOBOT . . . 25
5.3.2 Asimov Conform . . . 26
5.3.3 clMCTS . . . 26
5.4 Ablauf . . . 26

6 Auswertung **28**
6.1 Bait . . . 28
6.2 Brainman . . . 29
6.3 Catapults . . . 29
6.4 The Citadel . . . 30
6.5 Labyrinth . . . 30
6.6 Real Sokoban . . . 30
6.7 clMCTS . . . 31

7 Fazit **44**

8 Ausblick **45**

Literaturverzeichnis **45**

Anhang **46**

Abbildungsverzeichnis

2.1 Das Spiel Bait 10

3.1 Ein Baum 12
3.2 Breitensuche 13
3.3 Ein durch Makro-Operatoren reduzierter Suchbaum 15

4.1 Vereinfachte UML Darstellung der Klasse Cluster 17
4.2 Generierung des Lösungspfads 20

5.1 Beispiellevel: Bait 24
5.2 Beispiellevel: Brainman 24
5.3 Beispiellevel: Catapults 24
5.4 Beispiellevel: The Citadel 25
5.5 Beispiellevel: Labyrinth 25
5.6 Beispiellevel: Real Sokoban 25

6.1 Bait: Gewinnrate 32
6.2 Bait: Score 32
6.3 Bait: Game Ticks 33
6.4 Bait: Simulationen 33
6.5 Brainman: Gewinnrate 34
6.6 Brainman: Score 34
6.7 Brainman: Game Ticks 35
6.8 Brainman: Simulationen 35
6.9 Catapults: Gewinnrate 36
6.10 Catapults: Score 36
6.11 Catapults: Game Ticks 37
6.12 Catapults: Simulationen 37
6.13 The Citadel: Gewinnrate 38
6.14 The Citadel: Score 38
6.15 The Citadel: Game Ticks 39
6.16 The Citadel: Simulationen 39
6.17 Labyrinth: Gewinnrate 40
6.18 Labyrinth: Score 40
6.19 Labyrinth: Game Ticks 41
6.20 Labyrinth: Simulationen 41
6.21 Real Sokoban: Gewinnrate 42
6.22 Real Sokoban: Score 42
6.23 Real Sokoban: Game Ticks 43
6.24 Real Sokoban: Simulationen 43

Tabellenverzeichnis

6.1 Farbzuordnung der Agenten in den Diagrammen 28

A.1 Daten für den Agenten Cluster Suche 48
A.2 Daten für den Agenten YOLOBOT 48
A.3 Daten für den Agenten Asimov Conform 49
A.4 Daten für den Agenten clMCTS 49

1 Einleitung

Der Begriff der Künstliche Intelligenz (KI) lässt zunächst Filme wie I, Robot[1] oder auch Minority Report[2] denken. In beiden Filmen geht es um Künstliche Intelligenzen, die direkt und flexibel auf jede Art von *Eingabe* reagieren können und sich selbstständig an neue Gegebenheiten anpassen. Abseits der Filmwelt, tauchen immer wieder Begriffe wie *Deep Learning* und *Neuronales Netz* auf. Beide Verfahren setzen darauf, mit Hilfe von Trainingsdaten Entscheidungen zu treffen. Dabei besteht die Gefahr, dass bereits während der Entwicklung der Fokus zu stark auf das spätere Einsatzgebiet gelegt wird und die Künstliche Intelligenz zu einem *Künstlichen Fachidiot* wird, der außer seiner ursprünglichen Aufgabe keine weitere Leistung bringen kann.

Die Herausforderung besteht also darin, eine KI zu schaffen, die spezialisiert genug ist, um das ihr gestellte Problem zu lösen, jedoch auch universell für andere Probleme genutzt werden kann. Eine solche KI muss ihre Umgebung und die dort geltenden Regeln zu verstehen lernen, um anschließend in der Lage zu sein, auf Änderungen in der Umgebung zu reagieren oder diese gar selbst zu verändern und gestalten.

Ein Gebiet, in dem versucht wird, eine KI in möglichst vielen verschiedenen Umgebungen einzusetzen, ist das General Game Playing. Eine Umgebung wird dabei durch ein Spiel ersetzt. Um ein Spiel gut spielen zu können, muss die KI die Regeln des Spiels lernen. Nach einer getätigten Aktion, etwa einem Schritt in eine Richtung, bekommt die KI von Seiten des Spiels Feedback. Dies geschieht meistens in Form von Punkten. Daraus kann gefolgert werden, ob die Entscheidung für eine Aktion gut oder schlecht war. Durch die getroffene Entscheidung verändert sich der Aufbau des Spiels.

Um dieses Verhalten zu modellieren, wird ein sogenannter Suchbaum generiert. Dabei handelt es sich um einen gerichteten Baum, der nach und nach aufgebaut wird. Algorithmen, die einen Suchbaum generieren, werden Baumsuchalgorithmen genannt.

So wie es verschiedene Spiele gibt, gibt es auch verschiedene Baumsuchalgorithmen, die je nach Art des Spiels besser oder schlechter geeignet sind, das Spiel zu gewinnen. Spiele werden dabei in zwei große Gruppen eingeteilt: Deterministische und stochastische Spiele. Bei deterministischen Spielen ist der Ausgang des Spiels alleine von den Aktionen des Spielers abhängig. Ein Beispiel hierfür ist das 15-Puzzle[3]. In diesem Spiel müssen die Zahlen durch Verschieben in eine aufsteigende Reihenfolge gebracht werden. Bei gleicher Startverteilung der Zahlen kann das Spiel durch die gleiche Abfolge von Aktionen gewonnen werden.

Bei einem stochastischen Spiel wird ein immer gleicher Lösungsweg durch das Hinzufügen einer zufälligen Komponente verhindert. Als Beispiel wird an dieser Stelle Pacman[4] genannt. Ziel des Spiels ist es, den Avatar alle Punkte in einem Labyrinth essen zu lassen. Dabei muss er jedoch den zufällig agierenden Geistern ausweichen. Aufgrund der zufälligen Bewegung der Geister ist es unmöglich, die Punkte in der immer gleichen Reihenfolge einzusammeln.

Das Grundproblem ist jedoch bei beiden Spielarten das gleiche: Es muss eine Abfolge von Aktionen gefunden werden, um von einem Zustand in einen anderen zu gelangen. Diese Problem wird auch *sequentielles Entscheidungsproblem* genannt.

1 http://www.imdb.com/title/tt0343818/
2 http://www.imdb.com/title/tt0181689/
3 https://de.wikipedia.org/wiki/15-Puzzle
4 https://de.wikipedia.org/wiki/Pac-Man

In dieser Arbeit wird die Cluster Suche, eine Erweiterung der klassischen Baumsuche, vorgestellt. Diese Versucht durch Abstraktion den Suchbaum klein zu halten. Dazu werden mehrere Zustände in einem sogenannten Cluster zusammengefasst. Außerdem wird versucht, möglichst lange Sequenzen von Aktionen in einem Makro-Operator zusammen zu fassen, wieder mit dem Ziel, die Größe des Suchbaums zu reduzieren.

Dabei stellt sich folgende Kernfrage:

Kann die Idee der Cluster Suche leistungsmäßig im Vergleich zu anderen Suchalgorithmen bestehen?

1.1 Aufbau der Arbeit

Die nachfolgende Arbeit ist wie folgt aufgebaut: Kapitel 2 gibt einen allgemeinen Einblick in General Game Playing und stellt das GVG-AI Framework vor. In Kapitel 3 werden verschiedene Baumsuchen und Techniken vorgestellt, welche genutzt werden können, um GGP-Spiele zu lösen. In Kapitel 4 wird die Idee der Cluster Suche beschrieben und erklärt, wie eine Iteration umgesetzt werden kann. In den Kapiteln 5 und 6 wird eine Implementierung der Cluster Suche mit anderen Suchalgorithmen verglichen. In Kapitel 7 wird die Kernfrage aus Kapitel 1 beantwortet.

2 General Game Playing

Beim General Game Playing geht es darum, dass ein Agent möglichst viele verschiedene Spiele spielen und gewinnen kann. Ein Agent ist dabei die Umsetzung einer oder mehrerer Algorithmen, welche dieses Ziel erfüllen sollen. Über ein definiertes Leistungsmaß kann der Agent mit anderen Agenten verglichen werden. Bessere Agenten erhalten eine höhere Bewertung. Somit ist ein direkter Vergleich zwischen verschiedenen Algorithmen der künstlichen Intelligenz möglich.

Die Spiele im General Game Playing können von einfachen statischen und deterministischen Spielen, wie zum Beispiel ein Labyrinth, bis hin zu komplexen und dynamischen Spielen, wie Pacman, reichen. Der ideale Agent sollte dabei in der Lage sein, jedes Spiel lösen zu können. Dabei ist die Art und der Schwierigkeitsgrad des Spiels unerheblich.

Zu unterscheiden ist hierbei zwischen Single- und Multi-Player. Während ein Agent im Single-Player nur gegen einen Highscore und eventuelle im Spiel spezifizierte Gegner antritt, muss der Agent im Multi-Player direkt gegen andere Agenten antreten. Da der in dieser Arbeit vorgestellte Algorithmus ausschließlich für deterministische Spiele gedacht ist, wird nur der Single-Player-Bereich von General Game Playing betrachtet.

2.1 GVG-AI

Ein Wettkampf im Bereich General Game Playing ist die seit 2014 jährlich stattfindende GVG-AI Competition. Inzwischen sind über 100 verschiedene Spiele verfügbar. Seit diesem Jahr gibt es sogar einige Spiele in denen physikalische Eigenschaften simuliert werden.

Zur Simulation der Spiele kommt eine Java-Portierung der im Original in Python implementierten Game Description Language (GDL) zum Einsatz. Mit der GDL lassen sich die Regeln eines Spiels beschreiben. Pro Spiel ist es möglich, mehrere verschiedene Level zu spielen. Alle Level sind in einer zweidimensionalen, in Quadrate (Spielfelder) unterteilten Umgebung beschrieben. Sie unterscheiden sich in Größe und Komplexität voneinander. Für jedes Spiel sind daher einfache und schwierige Level verfügbar. So kann Level 0 des in Abbildung 2.1 dargestellte Spiels von einem Menschen innerhalb kürzester Zeit gelöst werden, während die Lösung für Level 3 deutlich schwieriger ist. Auch für eine KI ist Level 0 leichter als Level 3, da alleine die Anzahl der möglichen Züge im kleineren Level deutlich limitierter ist.

Die Zeit, die einem Agenten zur Verfügung steht, um ein Spiel/Level innerhalb des GVG-AI Frameworks, zu lösen ist begrenzt. Nach einer initialen Sekunde, muss sich ein Agent spätestens alle 40 Millisekunden CPU-Zeit für eine der verfügbaren Aktionen entscheiden. Ein Spiel dauert maximal 2000 solcher Recheneinheiten. [8]

Um ein Spiel zu lösen, muss der Agent einen Avatar innerhalb des Spielfelds bewegen. Dazu stehen ihm Bewegungsaktionen in die vier Himmelsrichtungen zur Verfügung. Bei einigen Spielen ist zusätzlich noch eine *USE*-Aktion verfügbar, über welche der Avatar mit anderen Objekten interagieren kann. Dabei ist dem Agenten zu Beginn eines Spiels nicht bekannt, welche Reaktion eine Aktion auslöst. So kann es sein, dass die Aktion *Rechts* statt den Avatar des Agenten der Intuition folgend ein Feld nach rechts zu bewegen, den Avatar erst mit der Blickrichtung nach rechts dreht und erst wenn dieser bereits nach rechts schaut, sich auf das nächste Feld bewegt.

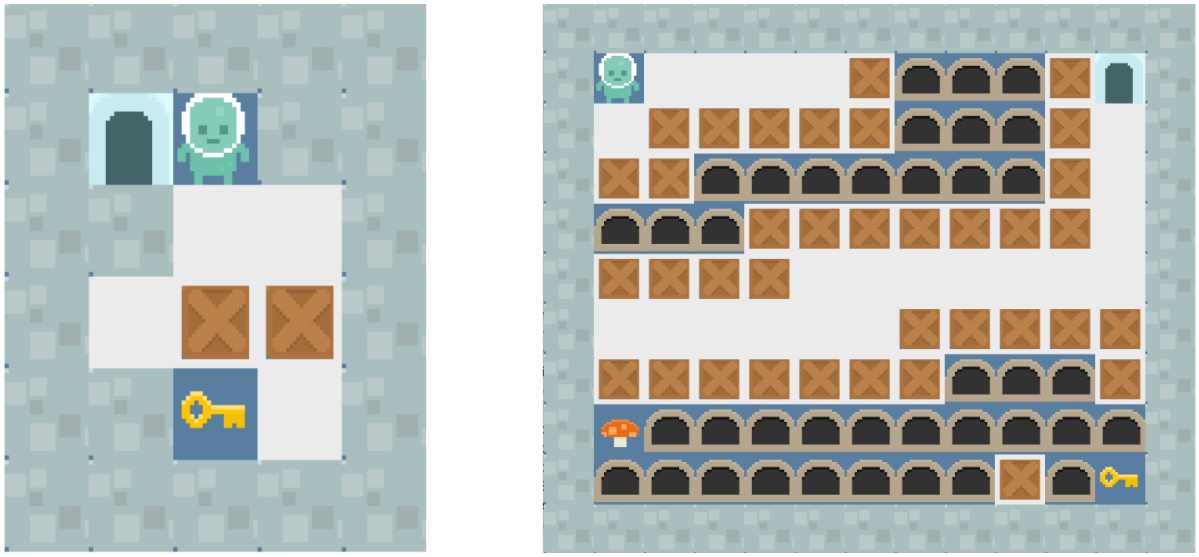

Abbildung 2.1: Das Spiel *Bait* aus dem GVG-AI Framework. Links Level 0, rechts Level 3.

Zusätzlich gibt es in jedem Spiel die Aktion *NIL*. Diese ist als aussetzen zu interpretieren. Der Agent führt also keine Aktion aus.

Da der Agent die Regeln des Spiels nicht kennt, diese jedoch lernen soll, hat er die Möglichkeit, das Ausführen einer Aktion zu simulieren und somit deren Auswirkung zu beobachten. Dazu erhält der Agent Feedback in Form von Score. Außerdem bekommt der Agent die Information, ob er sich in einem Zustand befindet, in dem er das Spiel gewonnen oder verloren hat.

Zusätzlich kann der Agent das gesamte Spielfeld beobachten und somit etwa potentiell interessante Objekte identifizieren und einen Weg zu diesen planen. Dabei kann der Agent abfragen, von welchem Typ ein Objekt ist. Typen sind zum Beispiel *Solid* (ein nicht bewegbares Objekt) oder *NPC*. Der Agent erfährt jedoch nicht, ob ein Objekt auf ihn positive oder negative Auswirkungen hat. Daher weiß er nicht, ob er es aufsuchen oder meiden sollte. Um diese Information zu erhalten, muss direkt mit dem Objekt per Aktion interagiert werden.

3 Überblick/Einordnung Suchverfahren

In diesem Kapitel werden die grundlegenden Suchverfahren, die der Cluster Suche zugrunde liegen, vorgestellt und beschrieben.

3.1 Markov Decision Process (MDP)

In den meisten Fällen ist es für einen Agenten nicht möglich, sein Ziel mit nur einem Schritt zu erreichen. Stattdessen werden mehrere aufeinander aufbauende Schritte benötigt. In diesem Kapitel wird dieses Problem mit Hilfe des Markov Entscheidungsprozesses (MDP) formalisiert. [10]

Durch einen MDP lässt sich modellieren, dass ein Agent eine Aktion a ausführen muss, damit er von einem Zustand s in einen Zustand s' gelangen. Dabei ergeben sich folgende Mengen und Funktionen:

S: eine Menge an Zuständen

A: eine Menge an Aktionen

$A(s)$: die Menge der in Zustand $s \in S$ ausführbaren Aktionen

$Pr(s'|s,a) \rightarrow [0,1]$: die Transaktionsfunktion, welche die Wahrscheinlichkeit angibt, von einem Zustand $s \in S$ mit der Aktion $a \in A$ in einen Zustand $s' \in S$ zu gelangen

$R(s) \rightarrow \mathbb{R}$: eine Belohnungsfunktion, die dem Übergang zum Zustand $s \in S$ eine Belohnung $\in \mathbb{R}$ zuordnet

s_0: ein Startzustand $s_0 \in S$

Nach Definition gilt, dass sich ein Agent zu einem Zeitpunkt immer nur in einem Zustand s^t befinden kann. Um den Zustand zu wechseln muss der Agent eine Aktion $a \in A(s^t)$ auswählen, die ausgeführt werden soll. Der Agent gelangt somit mit der Wahrscheinlichkeit $Pr(s'|s^t,a])$ in den Zustand s'.

Führt der Agent eine Aktion aus, erhält er anschließend die Information, in welchem neuen Zustand $s' \in S$ er sich befindet. Außerdem erfährt er, welche Belohnung $R(s')$ er für das Ausführen der Aktion erhalten hat.

Das Ziel hierbei ist, eine Strategie $\pi : S \times A \rightarrow [0,1]$ zu finden, die jedem Zustand-Aktions-Paar eine Wahrscheinlichkeit $\in [0,1]$ zuordnet, dass die Aktion $a \in A(s)$ im Zustand $s \in S$ ausgeführt wird, sodass die Belohnung über die Zeit maximiert wird.

Bei gegebener Strategie kann für jedes Zustand-Aktions-Paar ein Wert entsprechend der Belohnung definiert werden.

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

mit $\gamma \in [0,1]$ als Discountfaktor, der es ermöglicht, zeitlich frühere Zustände höher zu bewerten als zeitlich spätere. $P^t(s)$ gibt an, wie wahrscheinlich es ist, dass der Agent sich zum Zeitpunkt t in Zustand s befindet und berechnet sich iterativ über

$$P^t(s_0) = 1$$

und

$$P(s^{t+1}) = \sum_{s \in S} P^t(s) \sum_{a \in A(s)} Pr(s^{t+1}|a, s^t)\pi(s^t, a).$$

Somit ergibt sich die optimale Strategie π^* für das Entscheidungsproblem mit

$$\pi^*(a|s) = \text{argmax}_\pi \sum_{a \in A(s_0)} Q^\pi(s_0, a)\pi(s0, a).$$

Für einen MDP gibt es einige Sonderfälle. Dies ist beispielsweise der in dieser Arbeit verwendete deterministische MDP. Hierbei gilt, dass es nur einen Nachfolgezustand s^* geben kann, wenn in einem Zustand s eine Aktion a ausgeführt wird.

$$\forall s \in S, a \in A \exists s^* \in S : (Pr(s^*|a, s) = 1) \wedge (\forall s^\Diamond \in S \setminus \{s^*\} : Pr(s^\Diamond|a, s) = 0)$$

3.2 Baumsuche

Ein Baum ist ein spezieller Typ von Graph. Der Graph ist dabei gerichtet und zusammenhängend, jedoch nicht geschlossen [7]. Ein Baum (siehe Abbildung 3.1) besteht aus Knoten und Kanten, eine Kante verbindet jeweils zwei Knoten. In jedem Baum existiert genau ein Knoten, der nur abgehende Kanten hat. Dieser Knoten wird Wurzelknoten, oder kurz Wurzel, genannt. Hat ein Knoten nur eingehende Kanten, nennt man ihn Blatt.

In einem Suchbaum stellen die Knoten einen Zustand dar. Eine Kante modelliert einen Zustandsübergang, welcher durch das Ausführen einer Aktion erreicht wird. Der Wurzelknoten stellt den initialen Zustand dar, in dem der Agent seine Suche startet [9]. Um in einen anderen Zustand zu gelangen, wird ein Knoten expandiert. Expandieren bedeutet hierbei, dass der Agent jede mögliche Aktion ausführt und somit eine neue Menge an Zuständen erzeugt. Diese neu expandierten Zustände werden als Nachfolger an den alten Zustand angehängt. Im Folgenden muss entschieden werden, welcher der noch nicht expandierten Zustände als nächstes ausgewählt und expandiert werden soll. Abhängig von der Art, wie der Zustand ausgewählt wird, unterscheiden sich die verschiedenen Baumsuchverfahren.

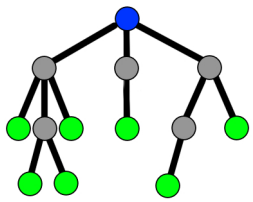

Abbildung 3.1: Ein Baum: Wurzel (blau), Knoten (grau), Blatt (grün) und Kante (schwarz)

Nachdem ein neuer Zustand zum Expandieren ausgewählt wurde, wird dies solange fortgesetzt, bis ein terminaler Knoten (der Zielzustand) gefunden wurde. Da je nach gewählten Auswahlverfahren der gefundene Pfad von der Wurzel zum Zielzustand nicht der optimale ist, kann der Baum solange weiter expandiert werden, bis keine weiteren Knoten mehr vorhanden sind. Dies ist auch die Abbruchbedingung für den Fall, dass keine Lösung gefunden wurde.

3.2.1 Kosten

Um zu bestimmen, ob der Pfad zu einem gegebenen Zustand besser oder schlechter ist als ein anderer, wird bestimmt, welche Kosten erzeugt wurden, um in diesen Zustand zu gelangen. Die Kosten für einen Zustand setzen sich dabei aus der Sequenz der Aktionen zusammen, wobei angenommen wird, das jedes Zustand-Aktions-Paar, also jede Aktion aus einem Zustand heraus, fest definierte Kosten verursacht. Ziel einer Baumsuche ist es, eine Folge von Aktionen zu einen terminalen Zustand zu finden, bei dem die Kosten minimal sind.

3.2.2 Redundante Pfade

Natürlich kann es auch vorkommen, dass ein Suchverfahren mehr als einen Pfad findet, der von einem Zustand in einen anderen führt. Diese alternierenden Pfade werden redundante Pfade genannt. Glücklicherweise ist es in den meisten Fällen nur notwendig den besten Pfad zu einem Zustand zu kennen, so dass alle teureren Pfade vergessen werden können oder gar nicht erst gespeichert werden müssen.

3.2.3 Zielbedingung

Über die Zielbedingung wird angegeben, welche Bedingungen erfüllt sein müssen, damit ein Knoten innerhalb eines Suchbaums als terminaler Knoten gewertet wird. Wie genau die Zielbedingung definiert ist, hängt dabei jeweils vom Anwendungsgebiet ab. Bei dem in Abbildung 5.2 dargestellten Spiel muss der Avatar beispielsweise das Ziel erreichen. Außerdem können durch Einsammeln der Diamanten Extrapunkte erzielt werden. Die Zielbedingung für dieses Spiel lautet daher: *Sammle möglichst viele Diamanten ein und gehe dann zum Ziel.*

3.3 Breitensuche

Bei der Breitensuche [9] handelt es sich um eine Suchstrategie, bei der immer alle Knoten einer Tiefe expandiert werden. Tiefe bezeichnet hierbei den Abstand zur Wurzel, also die Anzahl an Kanten, bis der Knoten von der Wurzel aus erreicht wurde.

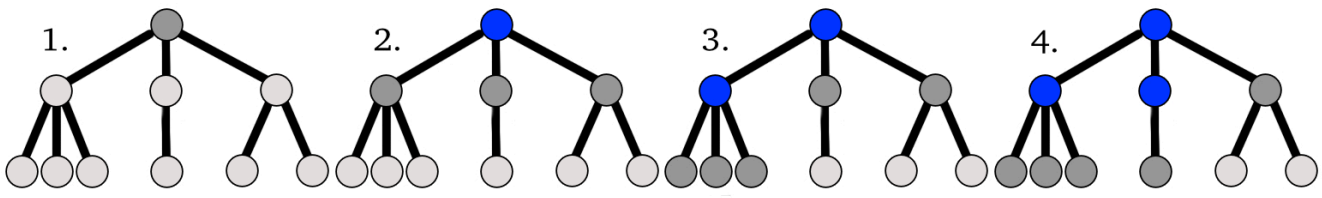

Abbildung 3.2: Suchreihenfolge einer Breitensuche mit den Iterationen 2., 3. und 4. sowie dem initialen Zustand 1.

Die Breitensuche kann durch eine FIFO-Warteschlange realisiert werden. Dazu werden neu expandierte Knoten (in Abbildung 3.2 in Dunkel-Grau dargestellt) jeweils am Ende der Warteschlange eingefügt und der *älteste* Knoten vorne aus der Warteschlange entfernt. Wird ein

Knoten aus der Warteschlange entfernt, wird dieser auch expandiert. Ist bei den neu expandierten Knoten ein Knoten dabei, der die Zielbedingung erfüllt, so wird dieser als Zielzustand markiert. Um redundante Pfade zu vermeiden, wird eine Liste über alle bereits expandierten Zustände angelegt. Wird ein Zustand expandiert, der sich bereits in der Liste befindet, wird dieser nicht zur Warteschlange hinzugefügt. Da die Breitensuche immer den flachsten Pfad zuerst findet, muss der neue Zustand daher mindestens so tief oder gar tiefer sein. Hierbei fällt auf, dass die Breitensuche nur einen optimalen Pfad findet, wenn die Pfadkosten eine nicht fallende Funktion zur Knotentiefe darstellen.

3.4 Uniform-Cost Search

Die Uniform-Cost Search [9] (Suche mit einheitlichen Kosten) ist eine einfache Erweiterung der Breitensuche. Statt immer den flachsten Knoten zu expandieren, wird der Knoten n expandiert, welcher die geringsten Pfadkosten $g(n)$ hat. Somit wird auch bei variierenden Schrittkosten immer der optimale Pfad gefunden.

Statt einer FIFO-Warteschlange werden die Knoten bei der Uniform-Cost Search entsprechend ihrer Kosten sortiert. Dadurch wird immer der Knoten mit den geringsten Gesamtpfadkosten vorne in der Warteschlange sein. Somit kann schnell und effizient der Pfad mit den geringsten Kosten gefunden und expandiert werden. Alle neu expandierten Zustände ordnen sich automatisch entsprechend ihrer Kosten in die Warteschlange ein.

Im Gegensatz zur Breitensuche wird die Zielbedingung erst überprüft, wenn ein Knoten zum Expandieren ausgewählt wird. Dies ist notwendig, da nicht garantiert werden kann, dass der Pfad zu einem Knoten optimal ist, wenn dieser gerade expandiert wurde.

Bei der Uniform-Cost Search kann es vorkommen, dass für einen Pfad ein redundanter Pfad *günstiger* ist als der vorher gefundene. Daher muss die Suche mittels eines Tests sicherstellen, dass immer der günstigste Pfad zu einem Knoten gespeichert wird.

3.5 Heuristische Suchverfahren

Sowohl Breitensuche als auch Uniform-Cost Search verwenden bei der Auswahl des zu expandierenden Zustandes nur Informationen, die sich aus der Historie des aktuellen Zustands ableiten lassen. Ein heuristisches Suchverfahren führt hierzu eine Funktion $h(n)$ für einen Knoten n ein, welche die Kosten von Knoten n bis zum Zielknoten schätzt. Für den Fall, dass n die Zielbedingung erfüllt, gilt $h(n) = 0$.

Nach welchen Bedingungen eine heuristische Funktion $h(n)$ die Kosten schätzt, ist dabei vom jeweiligen Problem abhängig. Um eine gute Schätzfunktion zu finden, ist ein fundiertes Hintergrundwissen nötig.

3.6 A*-Bestensuche

Eine A*-Bestensuche [9], im Folgenden nur noch Bestensuche genannt, expandiert als erstes den Knoten n mit dem die Wahrscheinlichkeit, einen Zielzustand zu erreichen, maximal ist. Dazu verwendet sie einen heuristischen Schätzer $h(n)$. Um sich nicht in Endlosschleifen zu

verfangen, werden außerdem die Kosten $g(n)$ betrachtet, welche auf dem Pfad zu Knoten n entstanden sind. Die gesamten Kosten $f(n)$ für Knoten n setzen sich daher wie folgt zusammen:

$$f(n) = g(n) + h(n).$$

Ansonsten verhält sich eine Bestensuche wie die Uniform-Cost Search. Dabei wird immer der Knoten als nächstes expandiert, der die geschätzt *günstigste* Lösung bietet.

3.7 Suchverfahren mit Makro-Operatoren

Alle bisher vorgestellten Suchverfahren gehen davon aus, dass beim Expandieren eines Knoten jeweils genau eine Aktion ausgeführt wird und darauf ein neuer Knoten folgt. Nun kann es aber sein, dass beim Ausführen einer Aktion α lediglich ein Zwischenknoten d erreicht wird. Von diesem muss eine weitere Aktion ausgeführt werden, um zu dem eigentlich neuen Knoten e zu gelangen. Die verfügbaren Aktionen in Knoten d beschränken sich dabei auf eine Aktion δ Richtung Knoten e, sowie eine Aktion zurück zu Knoten a. Da die Aktion zurück wenig Sinn macht, kann gesagt werden, dass Knoten a von Knoten d mit der Aktion $\alpha\delta$ erreichbar ist. Diese neue Aktion wird im Folgenden Makro-Operatoren genannt. [6]

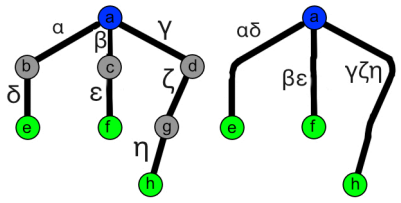

Abbildung 3.3: Links der normale Suchbaum, rechts der durch Makro-Operatoren reduzierte Suchbaum.

Makro-Operatoren ermöglichen, einen Suchbaum abstrakter zu betrachten, als es durch die Darstellung jeder Aktion durch eine Kante möglich ist. In Abbildung 3.3 wird links ein normaler Suchbaum mit Zwischenzuständen und mehreren atomaren Aktionen zwischen der Wurzel (blau) und einem Blatt (grün) dargestellt. Der Suchbaum rechts wurde reduziert, indem die Aktionen zu einem Blatt zusammengefasst wurden und somit der Zwischenzustand weggelassen werden kann.

Makro-Operatoren können von einem Agenten während der Ausführung gelernt oder vordefiniert werden. Die in dieser Arbeit verwendeten Makro-Operatoren werden während der Ausführung gelernt und dazu genutzt, eine Verbindung zwischen den Clustern herzustellen.

3.8 Clustering

Beim Clustering werden mehrere *ähnliche* Zustände zu einem Cluster zusammengefasst [4]. Somit kann die Menge der Zustände signifikant verringert werden.

Kriterien, nach denen Zustände als ähnlich gelten, können vordefiniert oder während der Ausführung gelernt werden. Wird das Ähnlichkeitsmaß während der Ausführung gelernt, ist es möglich, dass ein Zustand während der Ausführung besser in einen anderen Cluster passt als in seinen aktuellen. Ist dies der Fall, muss der Cluster gewechselt werden.

In der in dieser Arbeit vorgestellten Cluster Suche, kann das Wechseln von Clustern zu großen Problemen führen. Daher ist es zwingend notwendig, dass Ähnlichkeitsmaß vorher so zu definieren, dass Clusterwechsel nicht möglich sind.

4 Cluster Suche

Die Cluster Suche versucht, durch die Kombination von Clustering und das Lernen von Makro-Operatoren die Anzahl der notwendigen Berechnungen minimal zu halten. Dabei ist nicht festgelegt in welcher Reihenfolge expandiert wird. Um den Algorithmus nicht auf einen Anwendungsfall festzulegen und die Anwendung außerhalb der General Game Playing Domain zu erleichtern, ist die Beschreibung allgemein gehalten. Genauere Informationen zur Implementierung, welche im Zuge des Experiments erstellt wurde, sind in Kapitel 5.1 zu finden.

Im Folgenden wird die für die Cluster Suche benötigte Datenstruktur *Cluster* eingeführt. Anschließend wird erläutert, welche initialen Schritte notwendig sind, um mit der Ausführung des Algorithmus zu beginnen. Danach wird eine Iteration detailliert vorgestellt. Da die Cluster Suche abweichend von einer normalen Baumsuche einen Graphen generiert, muss nach Beenden der Iterationen noch ein abschließender Schritt getätigt werden, um einen Pfad vom Start- zum Zielknoten zu erhalten.

4.1 Cluster

Die Cluster Suche setzt auf einer normalen Baumsuche auf. Dabei wird ein Knoten durch einen Cluster und eine Kante durch eine Makro-Operation ersetzt. Jeder Cluster kennt alle möglichen Makro-Operationen zu anderen Clustern. In Abbildung 4.1 ist der Aufbau eines Clusters vereinfacht ohne Sichtbarkeiten dargestellt.

Cluster
ID: Number offene Ziele: Liste von Makro-Operatoren Nachfolger: Liste von Clustern Vorgänger: Liste von Clustern

Abbildung 4.1: Vereinfachte UML Darstellung der Klasse Cluster

Jeder Cluster erhält eine individuelle ID zur Unterscheidung. Diese ID wird durch eine Hash-Funktion generiert. Die Hash-Funktion sollte dabei so aufgebaut werden, dass sie für jeden zum Cluster gehörenden Zustand den gleichen Wert ausgibt. Somit ist es problemlos möglich, einen Zustand einem Cluster zuzuordnen.

Außerdem enthält jeder Cluster eine Liste mit offenen Zielen. Ein Ziel bezeichnet hierbei einen Makro-Operator, welcher ausgeführt werden muss, um in einen anderen Cluster zu gelangen. Zu einem Makro-Operator gehört jeweils auch der Zustand, von dem dieser ausgeführt wird.

Wird ein Makro-Operator ausgeführt, wird der neu erreichte Cluster in der Liste der Nachfolger gespeichert. Gleichzeitig wird der alte Cluster im neuen Cluster in der Liste der Vorgänger gespeichert. Die Listen enthalten neben den Clustern jeweils noch die benötigten Makro-Operatoren, sowie den zugehörigen Zustand, der es ermöglicht, den Clusterwechsel herbei zu führen. Durch die doppelte Verkettung mit Vorgänger und Nachfolger ist es möglich, den Graphen in beliebiger Richtung zu durchlaufen.

4.2 Initialisierung

Da bei der Cluster Suche ein Knoten durch einen Cluster ersetzt wird, muss der Cluster vor Beginn der Iterationen aus dem Startzustand erzeugt werden. Dabei müssen auch bereits alle Makro-Operatoren für den Cluster gelernt werden. Das Lernen von Makro-Operatoren wird im folgenden Kapitel genauer behandelt, so dass an dieser Stelle darauf verzichtet wird.

Während der Iteration werden die Cluster in eine Liste von offenen Clustern geschrieben und aus dieser gelesen. Ein Cluster ist solange in dieser Liste, bis alle Makro-Operatoren einmal ausgeführt wurden. Um mit der ersten Iteration zu beginnen, wird der Start-Cluster in die Liste der offenen Cluster geschrieben.

4.3 Iteration

Im Folgenden wird eine Iteration der Cluster Suche beschrieben. Wie bereits am Anfang des Kapitels erwähnt, ist die Beschreibung allgemein gehalten und spezifiziert keine genauen Methoden, die etwa zum Lernen von Makro-Operatoren genutzt werden. Somit ist auch eine genaue Abschätzung der Laufzeit an dieser Stelle nicht möglich. Bei der Beschreibung der einzelnen Schritte wird jedoch kurz auf ihre jeweilige Auswirkung auf die Gesamtlaufzeit eingegangen.

Eine Iteration der Cluster Suche lässt sich in folgende 4 Schritte aufteilen:

1. Cluster/Makro-Operator zum Expandieren auswählen
2. Makro-Operator expandieren
3. erreichten Zustand einem Cluster zuordnen
4. Makro-Operatoren zu möglichen anderen Clustern lernen

Dabei kann die Iteration durch zwei Bedingungen beendet werden:

- Es wurde ein Zustand gefunden, der die Zielbedingung erfüllt.
- Die Liste der offenen Cluster ist leer.

4.3.1 Cluster/Makro-Operator auswählen

Zu Beginn einer Iteration wird ein Cluster aus der Liste der offenen Cluster ausgewählt. Welcher Cluster dies ist, hängt von der festgelegten Reihenfolge ab. Als nächstes wird innerhalb des Clusters ein Makro-Operator aus der Liste der offenen Ziele ausgewählt. Die Reihenfolge ist abermals von der festgelegt Reihenfolge ab. Durch kontinuierliches Herausnehmen von Zielen wird die Länge der Liste solange reduziert, bis kein Ziel mehr offen ist. Wurden in einem Cluster alle Nachfolgezustände expandiert, wird dieser aus der Lister der offenen Cluster entfernt.

Das Auswahlverfahren kann hierbei auf unterschiedliche Weise ablaufen. So ist eine Kombination von verschiedenen Baumsuchen möglich, wie etwa ein nach Uniform-Cost Search ausgewählter Cluster mit anschließend nach Bestensuche ausgewählter Makro-Operation.

Die Komplexität dieses Schrittes leitet sich von den verwendeten Datenstrukturen ab. Da es für die meisten Warteschlangen sehr effiziente Verfahren gibt, sollte die Auswahl innerhalb eines Iterationsschritts nur wenig Einfluss auf die Gesamtlaufzeit haben.

4.3.2 Makro-Operator expandieren

Als nächstes wird der ausgewählte Makro-Operator expandiert, um in einen neuen Zustand zu gelangen und den Graphen zu erweitern. Für den Fall, dass der neue Zustand die Zielbedingung erfüllt, wird die Iteration abgebrochen und ein Lösungspfad generiert.

Die Laufzeit dieses Schrittes ist davon abhängig, wie viele atomare Aktionen in einem Makro-Operator zusammengefasst sind. Allgemein lässt sich sagen, dass sich die Laufzeit aus der Anzahl der Aktionen multipliziert mit der Zeit, die benötigt wird um eine solche Aktion auszuführen, zusammensetzt.

4.3.3 Cluster zuordnen

Im dritten Schritt wird der neu expandierte Zustand einem Cluster zugeordnet. Hierzu ist es notwendig zu bestimmen, ob der Zustand zu einem bereits existierenden Cluster passt oder ob ein neuer Cluster erstellt werden muss. Dabei kommt die bereits in Kapitel 4.1 erwähnte *Hash-Funktion* zum Einsatz. Entspricht der zurückgegebene Wert der ID eines bereits existierenden Clusters, wird der Zustand diesem zugeordnet. Ansonsten wird ein neues Cluster erstellt.

Im Fall eines neuen Clusters wird dieser in die Liste der offenen Cluster eingefügt.

Ein einmal einem Cluster zugeordneter Zustand verbleibt bis zur Terminierung des Algorithmus in diesem. Gäbe es diese feste Zuordnung nicht, müssten nach jeder Iteration alle Cluster neu überprüft werden. Da dies, gerade bei sehr großen Graphen, mit einem hohen Rechenaufwand verbunden ist, wird an dieser Stelle das Clustering in seiner Flexibilität zu Gunsten der Performance beschränkt.

Das Zuordnen sollte dabei keinen großen Einfluss auf die Gesamtlaufzeit haben und kann mit konstanter Laufzeit abgeschätzt werden.

4.3.4 Makro-Operator lernen

Im abschließenden Schritt ist es notwendig, Makro-Operatoren vom Cluster zu den Zielen zu lernen. Für den Fall, dass kein neuer Cluster erstellt wurde, kann dieser Schritt in den in Kapitel 4.4 beschriebenen finalen Schritt verschoben werden. Hierdurch können, wenn dies der *Lernalgorithmus* unterstützt Makro-Operatoren selektiv nach Bedarf gelernt und somit Rechenzeit gespart werden.

Doch was, wenn es sich um einen neuen Cluster handelt oder die Berechnung nicht ausgelagert werden soll? In diesem Fall müssen alle Makro-Operatoren gelernt werden, die potentiell in einen neuen Cluster führen. Die einfachste Methode dies zu realisieren, ist über eine Breitensuche mit der Zielbedingung, dass der Hash-Wert des neuen Knotens von der ID des aktuellen Clusters abweicht. Der Pfad zu diesem Knoten ist dann ein neuer Makro-Operator. Um alle Makro-Operatoren zu lernen, muss die Breitensuche so lange ausgeführt werden, bis kein Knoten mehr zum Expandieren verbleibt. Dabei ist ersichtlich, dass die Cluster Suche mit dieser Art des Lernens niemals besser als eine Breitensuche sein kann. Es ist also notwendig eine Lernmethode zu finden, die Makro-Operatoren innerhalb eines Bruchteils der Zeit lernen kann, die eine normale Baumsuche benötigt, um eine Lösung zu finden.

Ähnlich einer heuristischen Funktion ist auch für einen guten Lerner einiges an Hintergrundwissen notwendig und sollte ohne das Expandieren von Knoten in irgendeiner Art vonstatten-

gehen. Nur so kann die Laufzeit so gering wie möglich gehalten werden, um die Cluster Suche effizient einsetzten zu können. Die Laufzeit dieses Schrittes ist stark davon abhängig, wie die Makro-Operatoren gelernt werden. Sie ist auch in großem Maße für die Gesamtlaufzeit verantwortlich.

4.4 Generierung des Lösungspfads

Im Gegensatz zu einer normalen Baumsuche generiert die Cluster Suche einen Graphen und keinen Suchbaum. Da ein Cluster mehr als nur einen Vorgänger haben kann, ist es möglich, dass sich Zyklen bilden. Diese Zyklen repräsentieren einen redundanten Pfad. Der Pfad zu einem Knoten, welcher die Zielbedingung erfüllt, kann somit nicht direkt zurückverfolgt werden. Aus diesem Grund muss nach der Erstellung des Graphen auch der explizite Lösungspfad generiert werden.

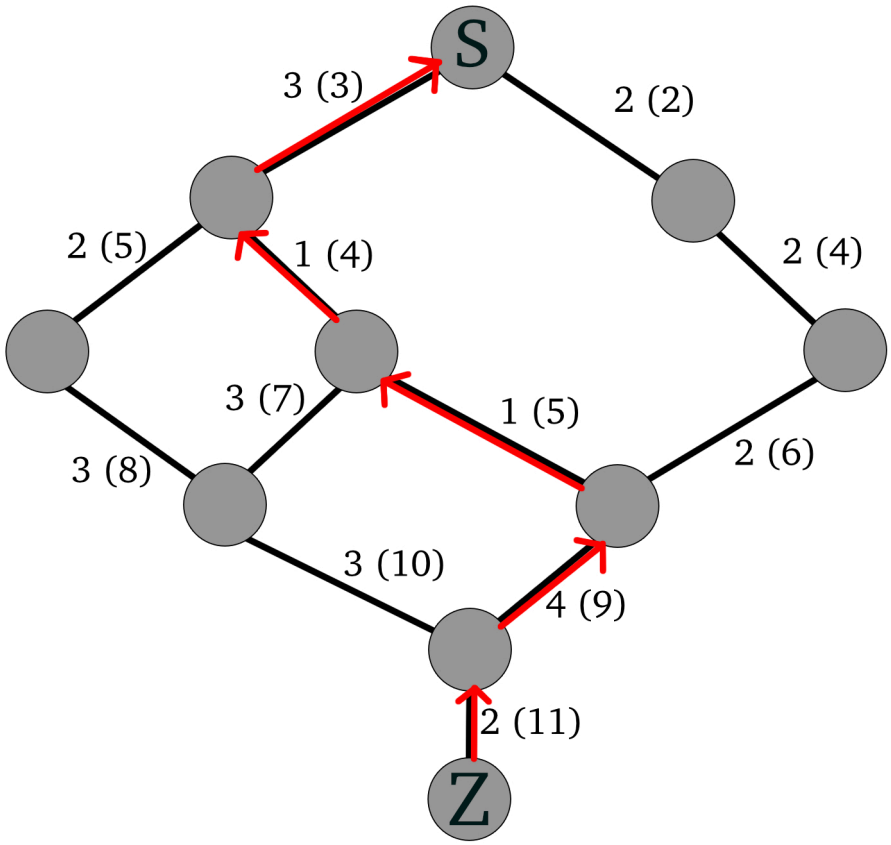

Abbildung 4.2: Rückwärtssuche: Lösungspfad (in rot) von Zielknoten Z zu Startknoten S. An den Kanten sind die Kosten und, in Klamern, die minimalen Gesamtkosten dargestellt.

Wird beim Expandieren ein Cluster erreicht, in dem die Zielbedingung erfüllt ist, muss durch den Graphen der Pfad gefunden werden, in dem die Kosten von der Wurzel zum Ziel-Knoten minimal sind. Hierzu wird vom Ziel-Knoten aus eine Rückwärtssuche gestartet. Dafür kann ein beliebiger Algorithmus verwendet werden, der in einem Graphen den minimalen Pfad zwischen zwei Knoten findet. Die Rückwärtssuche ist in Abbildung 4.2 dargestellt. Dabei wird jeweils der Vorgänger mit den geringeren Gesamtkosten gewählt. Dies ergibt den roten Pfad.

In Kapitel 4.3.4 wird erwähnt, dass bei bereits existierendem Cluster das Lernen der alternativen Makro-Operatoren in diesen Schritt ausgelagert werden kann. Wurde diese Entscheidung getroffen, müssen nun während der Rückwärtssuche für jeden Cluster, der mehr als einen Vorgänger hat, Makro-Operatoren nachgelernt werden. Da bekannt ist, von welchem Zustand a in welchen Nachfolgezustand b gewechselt werden soll, ist es nicht notwendig, alle Makro-Operatoren zu lernen. Es reicht, lediglich den Operator von a nach b zu bestimmen. Somit kann viel CPU-Zeit eingespart werden. Allerdings muss der zum Lernen verwendete Algorithmus dieses selektive Lernen auch unterstützen.

5 Experiment

Um die Performance der in Kapitel 4 beschriebenen Cluster Suche bewerten zu können, wird diese als Agent für das GVG-AI Framework (siehe Kapitel 2.1) implementiert. Anschließend wird der Agent mit anderen Agenten, welche für den GVG-AI Wettbewerb entwickelt wurden, verglichen. Da es sich bei der Cluster Suche um einen rein deterministischen Agenten handelt, der nicht mit stochastischen Effekten umgehen kann, werden Spiele ausgewählt, die diesem Schema entsprechen.

5.1 Implementierung

Da in Kapitel 4 der Algorithmus ohne spezielle Anwendungsdomäne beschrieben wird, müssen für die Implementierung einige Entscheidungen getroffen werden. Diese werden im folgenden Kapitel erläutert.

Die Implementierung ist ausschließlich für deterministische Spiele gedacht. Um kein Level durch unüberlegte Aktionen *ungewinnbar* zu machen, wird erst ein gesamter Lösungsweg gesucht, bevor eine echte Aktion ausgeführt wird. Dies wird erreicht, indem am Ende der 40 ms eines Game Ticks solange die NIL-Aktion zurückgegeben wird, bis eine Lösung gefunden wurde.

5.1.1 Zielbedingung

Damit die Cluster Suche entsprechend der Beschreibung in Kapitel 4.3 terminieren kann, wird die Zielbedingung wie folgt formuliert:

> Der Agent sucht solange nach einer Lösung, bis er die Lösung mit dem maximalen Score gefunden hat. Alternativ führt der Agent die vom Score her aktuell beste Lösung (ohne Sieg) aus, sollten die verbleibenden Game Ticks genau der Länge dieser Lösung entsprechen.

Die alternative Bedingung dient hierbei dem Zweck, dass der Agent zumindest noch einige Punkte erreicht, auch wenn er innerhalb der Zeitschranke keinen Weg findet das Spiel zu gewinnen.

5.1.2 Clustering

Um das Clustering innerhalb des GVG-AI Frameworks zu realisieren, werden alle Zustände zusammengefasst, die sich, mit Ausnahme der Position des Avatars, nicht unterscheiden. Dazu wird mittels *Brent Hash* [1] die Position aller Objekte gehashed. Somit werden nur dann unterschiedliche Hash-Werte generiert, wenn ein Objekt verschoben wurde, verschwunden ist oder neu aufgetaucht ist.

5.1.3 Makro-Operatoren

Damit für das Lernen der Makro-Operatoren keine zeitaufwändigen Simulationen benötigt werden, wurde die Knowledge Base (KB) von YOLOBOT, welche in [5] genauer beschrieben ist, in

den Agenten integriert. Durch die KB besteht die Möglichkeit, mittels eines A*-Algorithmus eine Kette von Aktionen zu bestimmen, welche ausgeführt werden müssen, um von der aktuellen Position zu einem Objekt zu gelangen. Dabei werden Wände und andere Objekte berücksichtigt und umgangen. Da nicht bekannt ist, ob es sich bei dem Objekt um eine schiebbare Kiste oder einen aufsammelbaren Diamanten handelt, wird für jedes Objekt je Himmelsrichtung ein Makro-Operator gelernt. Somit wird später beim Expandieren jedes Objekt vier mal, einmal aus jeder Richtung, angesteuert.

Makro-Operatoren werden jeweils nur dann gelernt, wenn ein Cluster neu erstellt wurde. Ein Operator von einem alternativen Startzustand wird im abschließenden Schritt selektiv gelernt.

5.1.4 Reihenfolge beim Expandieren

Um den Vergleich mit den anderen Agenten nicht durch eine unpassende Schätzfunktion zu verfälschen, werden die Cluster in Form einer Breitensuche expandiert. Die Liste der offenen Cluster ist somit eine FIFO-Warteschlange. Erst wenn alle offenen Ziele in einem Cluster expandiert wurden, wird der nächste Cluster aus der Warteschlange geladen.

5.1.5 Abschließende Pfadgenerierung

Damit sich aus der Cluster Suche eine ausführbare Lösung ergibt, wird abschließend der beste Pfad durch den Graphen generiert. Als Maß für die Güte $f(a, b)$ eines Pfads zwischen zwei Clustern a, b wird die Anzahl der Schritte (steps) mit den erhaltenen Punkten (points) verrechnet. Dazu wird folgende Formel verwendet:

$$f(a, b) = 10 \cdot \text{points} + \text{steps}$$

Es wird also festgelegt, dass ein Punkt 10 Schritte und somit auch 10 verbrauchte Game Ticks wert ist. Ziel ist es, den Pfad zu finden, der das Ergebnis der Funktion $f(s, z)$, mit Startknoten s und Zielknoten z, maximiert.

Wird ein Vorgänger gewählt, dessen Makro-Operator noch nicht bekannt ist, wird dieser mit dem selben A* wie in Kapitel 5.1.3 gelernt.

5.2 Auswahl der Spiele

Der in Kapitel 2.1 vorgestellte Wettbewerb umfasst mehr als 100 verschiedene Spiele. Viele dieser Spiele enthalten stochastische Effekte wie beispielsweise Gegner, die an zufälligen Positionen spawnen oder in einem nicht erkennbaren Muster über das Spielfeld laufen. Aufgrund der deterministischen Limitierung eignen sie sich nicht, um den in Kapitel 4 beschriebenen Algorithmus mit den in Kapitel 5.3 vorgestellten Agenten zu vergleichen. Die Spielauswahl beschränkt sich daher auf rein deterministische Spiele, wie Schiebepuzzles und Labyrinthe.

5.2.1 Bait

In diesem Spiel geht es darum, mit Hilfe eines Schlüssels in das Ziel zu gelangen. Auf dem Weg zum Ziel muss der Spieler Kisten verschieben. Dabei muss der Spieler darauf achten, dass er in keines der Löcher im Boden fällt. Mit Hilfe einer Kiste kann ein Loch jedoch verfüllt und somit unschädlich gemacht werden. Durch Einsammeln eines Pilzes ist es möglich, Extra-Punkte zu erhalten. [3]

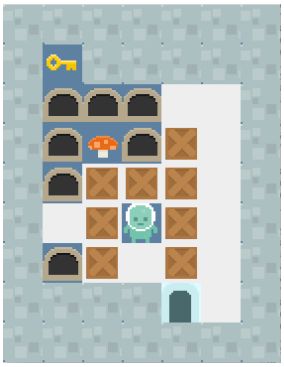

Abbildung 5.1: Beispiellevel: Bait

5.2.2 Brainman

Der Spieler muss einen oder mehrere Schlüssel in die Türen schieben. Hat er das erreicht, öffnet sich die Tür und der Weg zum Ziel ist nicht mehr blockiert. Nachdem ein Schlüssel angeschoben wurde, rutscht er solange weiter, bis sein Weg durch ein anderes Objekt blockiert wird. Die Edelsteine können eingesammelt werden und geben Punkte. [3]

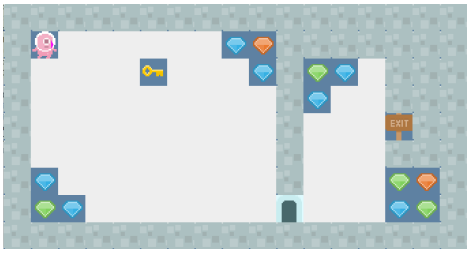

Abbildung 5.2: Beispiellevel: Brainman

5.2.3 Catapults

Ziel des Spiels ist es, den Ausgang zu erreichen. Da der Spieler Nichtschwimmer ist, muss er mit Hilfe der Katapulte über das Wasser springen. Jedes Katapult kann nur einmal verwendet werden. [3]

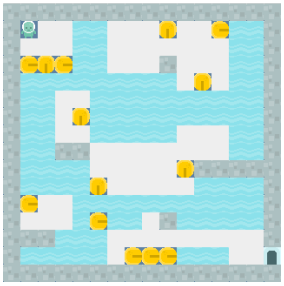

Abbildung 5.3: Beispiellevel: Catapults

5.2.4 The Citadel

Während Steine in diesem Spiel nur nacheinander verschoben werden können, ist es möglich, mehrere Kisten auf einmal zu bewegen. Kisten und Steine füllen dabei verschiedene Arten von Löchern. Das Spiel ist gewonnen, wenn der Spieler das Ziel erreicht. [3]

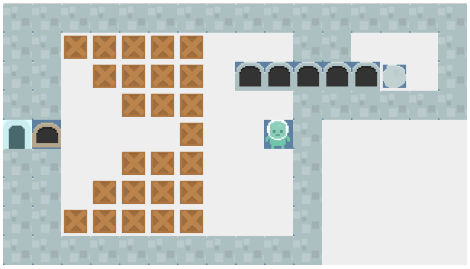

Abbildung 5.4: Beispiellevel: The Citadel

5.2.5 Labyrinth

In diesem Spiel muss der Ausgang aus dem Labyrinth gefunden werden. Dabei gibt es Fallen, die der Spieler meiden muss, da er ansonsten das Spiel verliert. [3]

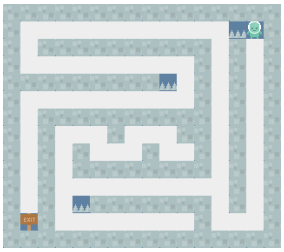

Abbildung 5.5: Beispiellevel: Labyrinth

5.2.6 Real Sokoban

Der Spieler muss mit Kisten alle Löcher zur gleichen Zeit bedecken. Eine Kiste ist dabei nicht fest auf einem Loch, sondern kann auch wieder von diesem herunter geschoben werden. [3]

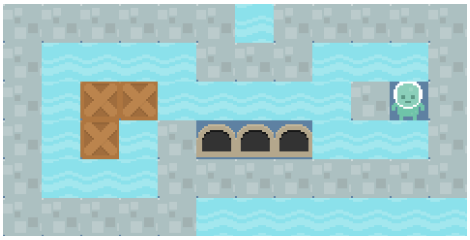

Abbildung 5.6: Beispiellevel: Real Sokoban

5.3 Agenten zum Vergleich

Um die Leistungsfähigkeit der Cluster Suche einordnen zu können, wird diese mit verschiedenen Agenten verglichen. Dafür werden Agenten verwendet, welche für den GVG-AI Wettbewerb entwickelt wurden. Die verwendeten Agenten werden im Folgenden beschrieben.

5.3.1 YOLOBOT

YOLOBOT ist der Gewinner aus dem Jahr 2015. Der Agent versucht, ein Spiel solange mittels einer heuristischen Suche zu lösen, bis ein stochastischer Effekt auftritt. In diesem Fall wird

versucht, das Spiel mittels MCTS zu lösen. Während der Agent versucht ein Spiel zu lösen, wird kontinuierlich eine Knowledge Base gelernt. Dies ermöglicht dem Agenten, Vorhersagen über den nächsten Spielzustand im Verlauf eines Spiels zu präzisieren. Durch seine Knowledge Base ist YOLOBOT beispielsweise in der Lage bestimmte Aktionen, etwa das Laufen gegen eine Wand, vor deren Ausführung zu erkennen. Es ist auch möglich, das Spawnen von Monstern vorherzusagen. Somit lassen sich überflüssige Simulationen vermeiden und die zur Verfügung stehende Rechenzeit kann für andere Dinge genutzt werden. [5]

5.3.2 Asimov Conform

Asimov Conform stammt aus der GVG-AI Competition 2016 und wurde im Zuge der Veranstaltung *Praktikum aus Künstliche Intelligenz* an der TU Darmstadt von vier Studenten entwickelt. Es handelt sich hierbei um einen Solver, der für stochastische Spiele MCTS verwendet und für deterministische eine Bestensuche. Während der implementierte MCTS-Algorithmus lediglich im Mittelfeld einzuordnen ist, handelt es sich bei der Bestensuche um eine sehr leistungsfähige Implementierung, welche die meisten deterministischen Spiele dominierte.

Der Agent ist in der Lage, noch während des Ausführens einer Lösung nach möglichen besseren Lösungen zu suchen. Außerdem wird versucht, Spielzustände mit gleichem Aufbau zu erkennen und als einen Zustand zu betrachten.

5.3.3 clMCTS

Hierbei handelt es sich um die Beispiel-Implementierung eines Closed-Loop MCTS Algorithmus [2] aus dem GVG-AI Framework. Da der MCTS-Agent im Gegensatz zu den beiden anderen Agenten und der Cluster Suche nicht für deterministische Spiele ausgelegt ist, wird ein vergleichsweise schlechtes Abschneiden erwartet.

5.4 Ablauf

Um die Cluster Suche im Vergleich zu den in Kapitel 5.3 vorgestellten Agenten einordnen zu können, werden alle vier Agenten auf den in Kapitel 5.2 vorgestellten Spielen ausgeführt. Damit jedem Agenten die gleichen Hardwareressourcen zur Verfügung stehen, wird der Arbeitsspeicher auf 4 GB beschränkt. Außerdem wird kein Multithreading unterstützt. Um eventuelle Zufallseffekte kompensieren zu können, wird jedes Level in jedem Spiel auf jedem Agenten 25 mal ausgeführt. Die Anzahl wurde hierbei auf 25 gesetzt, da einige Testläufe sowie auch die ermittelten Ergebnisse zeigen, dass die Werte keine erheblichen Abweichungen von ihrem Mittelwert aufzeigen. Dies ist vor allem deswegen der Fall, da es sich bei den Spielen nur um deterministische Spiele handelt.

Für jedes ausgeführte Spiel werden folgende Werte notiert:

- gewonnen oder verloren
- Anzahl an erhaltenem Score
- Benötigte Anzahl an Game Ticks
- Benötigte Anzahl an Simulationen

Im Anschluss wird für jede Agent-Spiel-Level-Kombination der Mittelwert bestimmt. Somit sind die Ergebnisse einfacher miteinander zu vergleichen.

Da die Anzahl der Game Ticks, sowie die verfügbaren Hardwareressourcen für ein Spiel begrenzt sind, besteht die Möglichkeit, dass ein Spiel von einem Agenten nicht gewonnen wird.

6 Auswertung

In diesem Kapitel werden die erzeugten Daten mit Hilfe einer grafischen Darstellung ausgewertet. Zur Darstellung wurden Balkendiagramme gewählt. Jeder Agent besitzt hierbei seine eigene feste Balkenfarbe (siehe Tabelle 6.1). Für jedes Spiel wurden vier Diagramme erstellt: Anzahl gewonnene Spiele, erreichter Score, benötigte Game Ticks und die benötigten Simulationen. Die den Diagrammen zugrunde liegenden Daten können in denen im Anhang befindlichen Tabellen A.1, A.2, A.3 und A.4 nachgelesen werden.

Tabelle 6.1: Farbzuordnung der Agenten in den Diagrammen

Agent	zugeordnete Farbe
Cluster Suche	blau
YOLOBOT	orange
Asimov Conform	grau
clMCTS	grün

Jeder Balken repräsentiert den berechneten Mittelwert der 25 simulierten Spiele. Zusätzlich ist die Abweichung zum minimalen und maximalen Wert mit eingezeichnet, um den Schwankungsbereich der Ergebnisse besser einschätzen zu können.

Der clMCTS-Agent wird nicht für jedes Spiel einzeln analysiert, sondern gesondert in Kapitel 6.7 ausgewertet, da es sich bei ihm nicht um einen auf deterministische Spiele spezialisierten Agenten handelt.

6.1 Bait

Beim Spiel Bait (Kapitel 5.2.1) handelt es sich um ein klassisches Schiebepuzzle. Der Spieler erhält für jede Kiste, die in ein Loch geschoben wird, einen Punkt. Sammelt er den Pilz ein, erhält er ebenfalls einen Punkt. Findet der Spieler den Schlüssel und gelangt anschließend ins Ziel erhält er 5 Punkte.

Die Anzahl der verschiebbaren Kisten pro Level ist dabei wie folgt: Level 0 zwei Kisten, Level 1 zwei Kisten, Level 2 sechs Kisten, Level 3 46 Kisten und Level 4 acht Kisten. Über die Anzahl der Kisten kann ein Rückschluss auf die Schwierigkeit der einzelnen Level gezogen werden. Es gilt: Je mehr Kisten vorhanden sind, desto schwieriger ist ein Level.

In Abbildung 6.1 wird diese Aussage direkt bestätigt. Level 3, das Level mit den meisten Kisten, wird von nur einem Agenten gelöst. Insgesamt fällt auf, dass Asimov Conform in jedem Level den meisten Score erreicht (siehe Abbildung 6.2) und auch mit Blick auf die der Anzahl der benötigten Game Ticks bis auf Level 2 immer vorne mit dabei ist (siehe Abbildung 6.3). Dass Asimov Conform hier mehr Zeiteinheiten zum Lösen benötigt, liegt daran, dass der Agent in diesem Level einen Punkt mehr holt, als die anderen Agenten. DAfür muss er mehr Schritte auf dem Spielfeld machen.

Die Cluster Suche ist über alle fünf Level gesehen vergleichbar mit YOLOBOT. Eine Ausnahme bildet Level 4, in dem die Cluster Suche zwar den gleichen Score erzielt, jedoch mehr Game

Ticks und mehr Simulationen (siehe Abbildung 6.4) benötigt. Dieses Verhalten lässt sich durch die in der Cluster Suche verwendete Breitensuche erklären, da die Anordnung der Kisten in diesem Level eher suboptimal ist.

6.2 Brainman

Bei Brainman (Kapitel 5.2.2) handelt es sich um ein Schiebepuzzle, bei dem der Score durch das Einsammeln von Diamanten maximiert werden kann. Dabei ist jeder eingesammelte Diamant einen Punkt wert. Wird mit einem Schlüssel eine Tür geöffnet, erhält der Spieler vier Punkte. Für das Betreten des Ziels wird der Spieler mit 10 Punkten belohnt.

Level 0 und 1 sind dabei als relativ einfach anzusehen, da sie jeweils nur einem Schlüssel und eine überschaubare Anzahl an Diamanten beinhalten. Level 2 und 3 haben jeweils drei Schlüssel, ein großes Spielfeld und dementsprechend viele Diamanten. In Level 4 gibt es auch drei Schlüssel, allerdings ist das Spielfeld nicht so groß wie in den beiden vorherigen Level und besitzt daher auch eine geringere Anzahl an Diamanten.

Dass das Lösen Level 2 und 3 aus Such der Agenten am schwirigsten ist, fällt bei Betrachtung von Abbildung 6.5 auf. Kein Agent schafft es, hier ein Spiel zu gewinnen. Die Cluster Suche kann beim Score (siehe Abbildung 6.6) im Vergleich zu Asimov Conform mithalten. Lediglich in den beiden nicht gewinnbaren Leveln erzielt die Cluster Suche weniger Score. Allerdings benötigt die Cluster Suche deutlich mehr Game Ticks als Asimov Conform (siehe Abbildung 6.7). Auffällig ist, dass in Level 4 die gesamten 2000 Zeiteinheiten benötigt werden, um eine Lösung zu finden. Da die Anzahl der Game Ticks jedoch nicht von Spiel zu Spiel variiert ist und jedes Spiel in Level 4 von der Cluster Suche gewonnen wird, ist anzunehmen, dass es sich hierbei um einen Bug handelt. Dieser tritt vermutlich durch die in der Zielbedingung formulierte Scoremaximierung auf. Die Cluster Suche benötigt, über alle Level gesehen, eine vergleichbare Menge an Simulationen wie Asimov Conform (siehe Abbildung 6.8).

YOLOBOT liegt in diesem Spiel weit hinter Asimov Conform und der Cluster Suche. Dies wird bereits durch die Beobachtung klar, dass Level 1 nur von der Cluster Suche und Asimov Conform gewonnen werden.

6.3 Catapults

Catapults (Kapitel 5.2.3) ist ein labyrinth-artiges Spiel, bei dem mit Hilfe von Katapulten Wassergräben überwunden werden müssen. Für jedes benutzte Katapult erhält der Spieler einen Punkt.

Die Komplexität der einzelnen Level unterscheidet sich dabei kaum, da jedes Level gleich groß ist und eine ähnliche Anzahl an Katapulten beinhaltet. So ist es nicht verwunderlich, dass sowohl die Cluster Suche, als auch YOLOBOT und Asimov Conform sämtliche Level gewinnen (siehe Abbildung 6.9).

Auch beim erzielten Score gibt es wenige Unterschiede zwischen den drei Agenten (siehe Abbildung 6.10). YOLOBOT holt in Level 0 einen Punkt mehr als die anderen beiden Agenten. Asimov Conform erziehlt in Level 3 nicht immer die volle Punktzahl von fünf Punkten. Dem entsprechend ähnlich ist auch die Anzahl der benötigten Game Ticks (siehe Abbildung 6.11). Wird jedoch die Anzahl der benötigten Simulationen (siehe Abbildung 6.12) betrachtet, fällt auf, dass die Cluster Suche hier im Vergleich zu YOLOBOT und Asimov Conform deutlich weniger Simulationen benötigt.

6.4 The Citadel

The Citadel (Kapitel 5.2.4) ist ein Schiebepuzzle mit bewegbaren Kisten und Steinen. Für jede Kiste oder jeden Stein, der in ein passendes Loch geschoben wird, erhält der Spieler einen Punkt. Wird das Ziel erreicht, erhält der Spieler fünf Punkte.

Während Level 0 und Level 4 nur Kisten enthalten, muss in den restlichen drei Leveln mindestens ein Stein in ein Loch geschoben werden, um das Spiel zu gewinnen. Dabei ist die Anzahl der verschiebbaren Objekte in Level 1 und Level 2 besonders hoch, somit handelt es sich bei diesen beiden Leveln um die komplexesten. Level 3 ist aufgrund der vorhanden Steine etwas komplizierter als Level 0 und Level 4.

Die beiden schwierigsten Level werden von keinem der Agenten gelöst (siehe Abbildung 6.13). Bei den restlichen Leveln liegt die Siegrate für die Cluster Suche, YOLOBOT und Asimov Conform jeweils bei 100%. Der erhaltene Score ist bei allen drei Agenten etwa gleich (siehe Abbildung 6.14), wobei Asimov Conform in Level 2 deutlich mehr Score bekommt als seine zwei Konkurrenten. In Level 3 erzielt die Cluster Suche jedes mal zwei Punkte mehr als die anderen Agenten, benötigt dafür in diesem Level auch viel mehr Game Ticks und Simulationen. Ansonsten liegt die Anzahl der benötigten Simulationen der Cluster Suche unter der von YOLOBOT und Asimov Conform (siehe Abbildung 6.16). In Level 0 verbraucht die Cluster Suche weniger Game Ticks, um das Ziel zu finden. In Level 4 ist die benötigte Anzahl ausgeglichen (siehe Abbildung 6.15).

6.5 Labyrinth

Labyrinth (Kapitel 5.2.5) ist das einfachste Spiel im Experiment. Der Spieler erhält einen Punkt, wenn er das Spiel gewinnt und einen Negativpunkt, wenn er durch eine der Fallen das Spiel verliert.

Da der Prozentsatz der gewonnenen Spiele (siehe Abbildung 6.17), der erhaltene Score (siehe Abbildung 6.18) und die benötigten Game Ticks (siehe Abbildung 6.19) für die Cluster Suche, YOLOBOT und Asimov Conform keinen Unterschied aufweisen, kann ein Vergleich nur über die Anzahl der benötigten Simulationen (siehe Abbildung 6.20) gemacht werden. Hierbei ist dem Betrachter ersichtlich, dass die Cluster Suche in jedem Level die wenigsten Simulationen benötigt.

6.6 Real Sokoban

Spielziel des Schiebepuzzles Real Sokoban (Kapitel 5.2.6) ist es, jedes auf dem Spielfeld vorhandene Loch mit einer Kiste zu bedecken. Dabei erhält der Spieler für jede Kiste, die ein Loch bedeckt einen Punkt. Entfernt er eine Kiste wieder von einem Loch, wird der Punkt wieder abgezogen.

In den fünf Leveln gibt es unterschiedliche Anzahlen von Kisten. In Level 3 und Level 4 befinden sich nur zwei Kisten. In Level 1 sind es drei, in Level 0 vier Kisten. Level 2 enthält mit sechs Kisten die meisten. Die Anzahl der Kisten lässt jedoch nicht direkt auf die Schwierigkeit eines Levels schließen, da diese vor allem durch den Aufbau des Levels definiert wird. Somit ist die aufsteigende Reihenfolge entsprechend der Schwierigkeit: Level 3, Level 0, Level 2, Level 4 und Level 1.

Bei der Betrachtung der gewonnen Spiele (siehe Abbildung 6.21) fällt auf, dass Asimov Conform bei diesem Spiel nicht so gut abschneidet wie bei anderen Spielen. Dies ist vermutlich auf die verwendete Heuristik zurückzuführen. Da der volle Score nur erreicht werden kann, wenn das Spiel gewonnen wird, ist es nachvollziehbar, dass der erreichte Score (siehe Abbildung 6.22) in einem direkten Verhältnis zu den gewonnenen Spielen steht. Auffällig ist, dass die Cluster Suche im Bezug auf die benötigten Game Ticks (siehe Abbildung 6.23) und Simulationen (siehe Abbildung 6.24) deutlich schlechtere Werte liefert als YOLOBOT.

6.7 clMCTS

Der clMCTS-Agent liefert in allen Spielen die schlechtesten Ergebnisse und schneidet somit deutlich schlechter ab als die drei anderen Agenten. Da der MCTS-Algorithmus nicht für deterministische Spiele implementiert wurde, ist dieses Verhalten logisch. Dabei ist vor allem zu beachten, das clMCTS nicht erst die gesamte Lösung plant, bevor diese ausgeführt wird, sondern sich jeweils nach 40 ms für eine Aktion entscheidet. Handelt es sich bei dem Spiel um Schiebepuzzle, kann es somit vorkommen, dass clMCTS eine Kiste so verschiebt (etwa in eine Ecke), dass das Spiel im Anschluss unlösbar wird.

Dies lässt sich direkt aus den gewonnenen Spielen ablesen. So kann clMCTS noch einige einfache Level wie Bait 0 oder Labyrinth 0 gewinnen. Reichen die Rollouts des MCTS-Algorithmus allerdings nicht mehr weit genug, geht die Gewinnrate gegen Null.

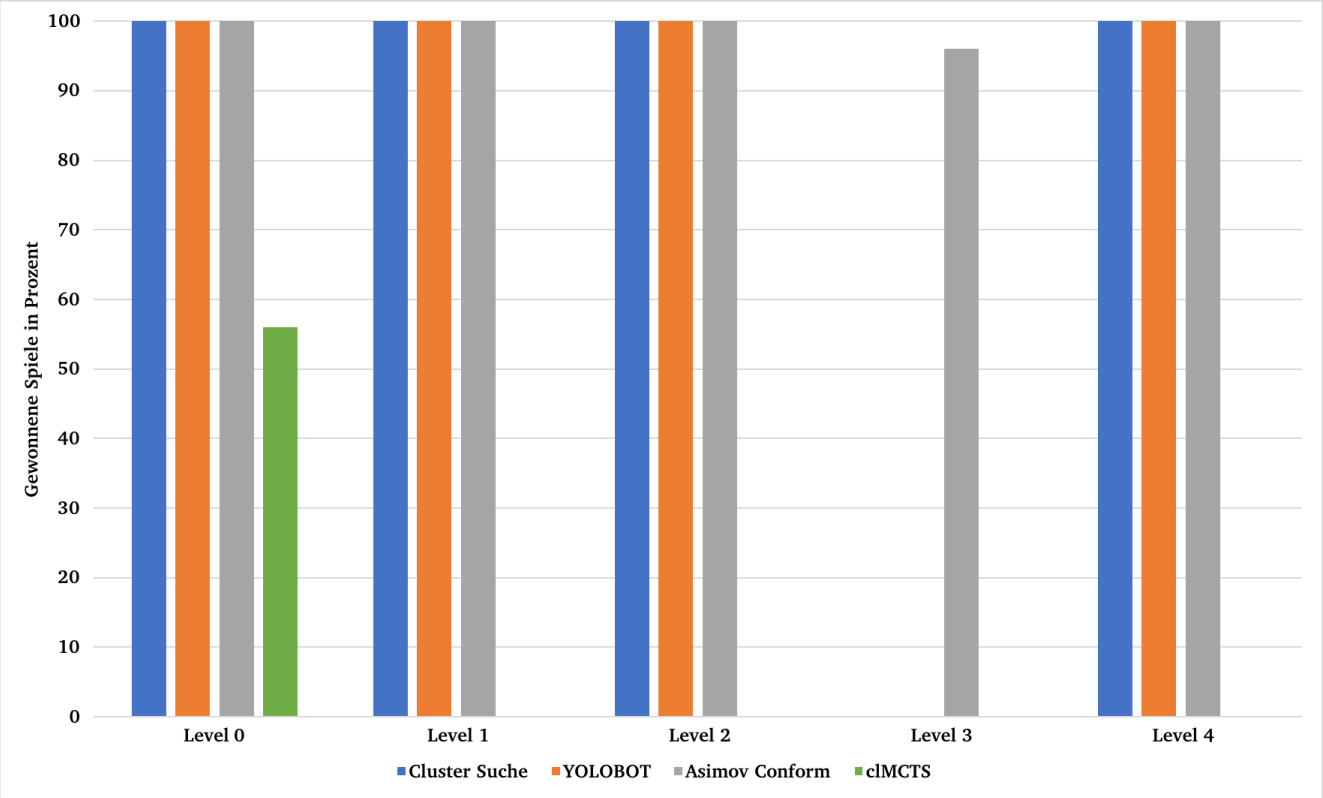

Abbildung 6.1: Bait: Prozentsatz der gewonnenen Spiele pro Level

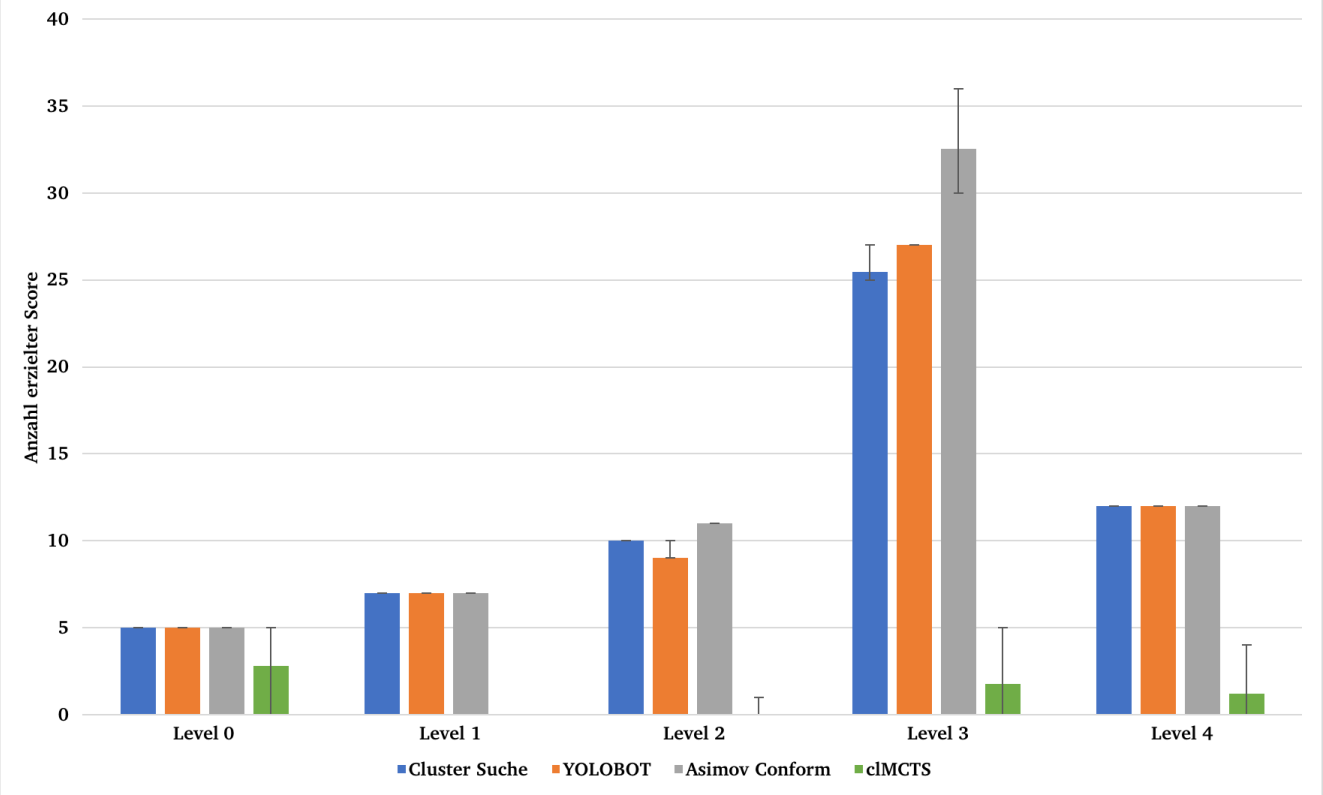

Abbildung 6.2: Bait: Mittelwert des erzielten Scores pro Level

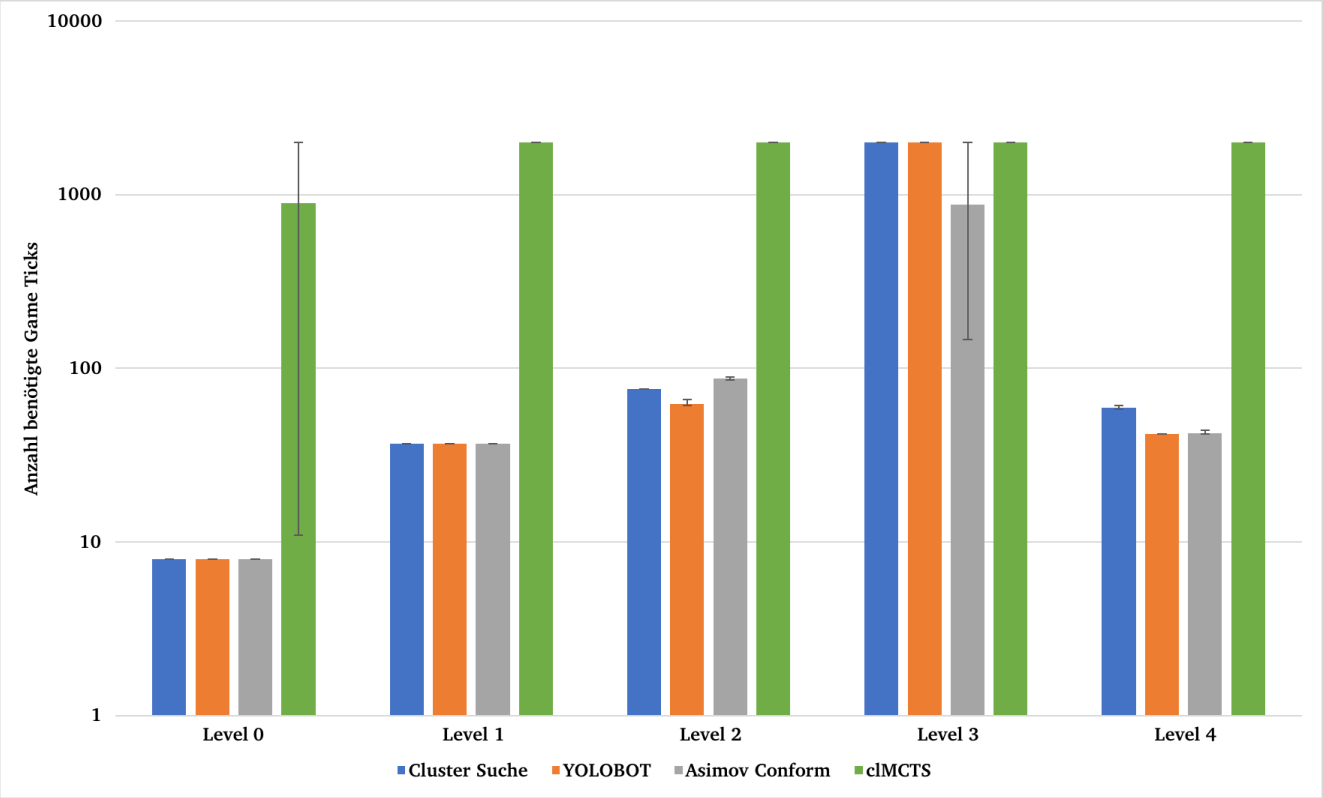

Abbildung 6.3: Bait: Logarithmische Darstellung des Mittelwerts der benötigten Game Ticks pro Level

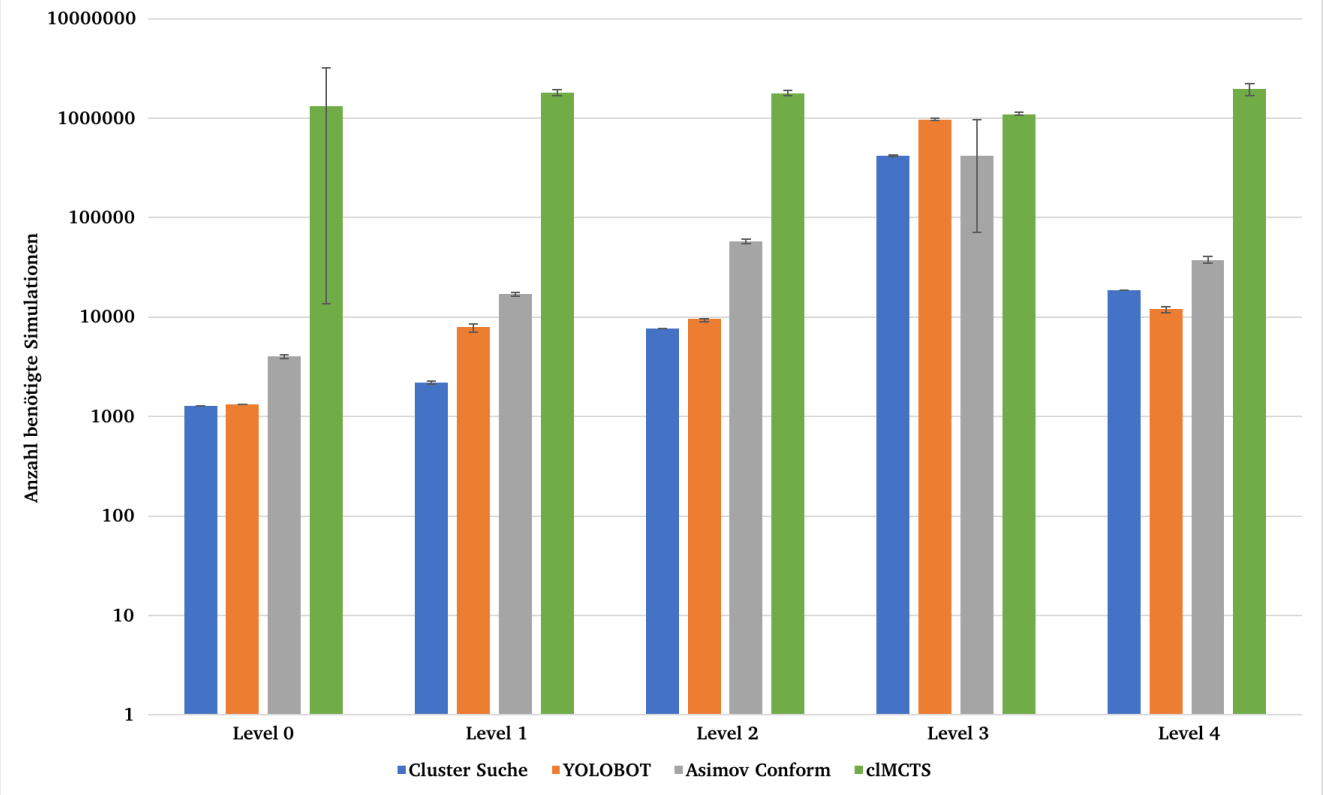

Abbildung 6.4: Bait: Logarithmische Darstellung des Mittelwerts der benötigten Simulationen pro Level

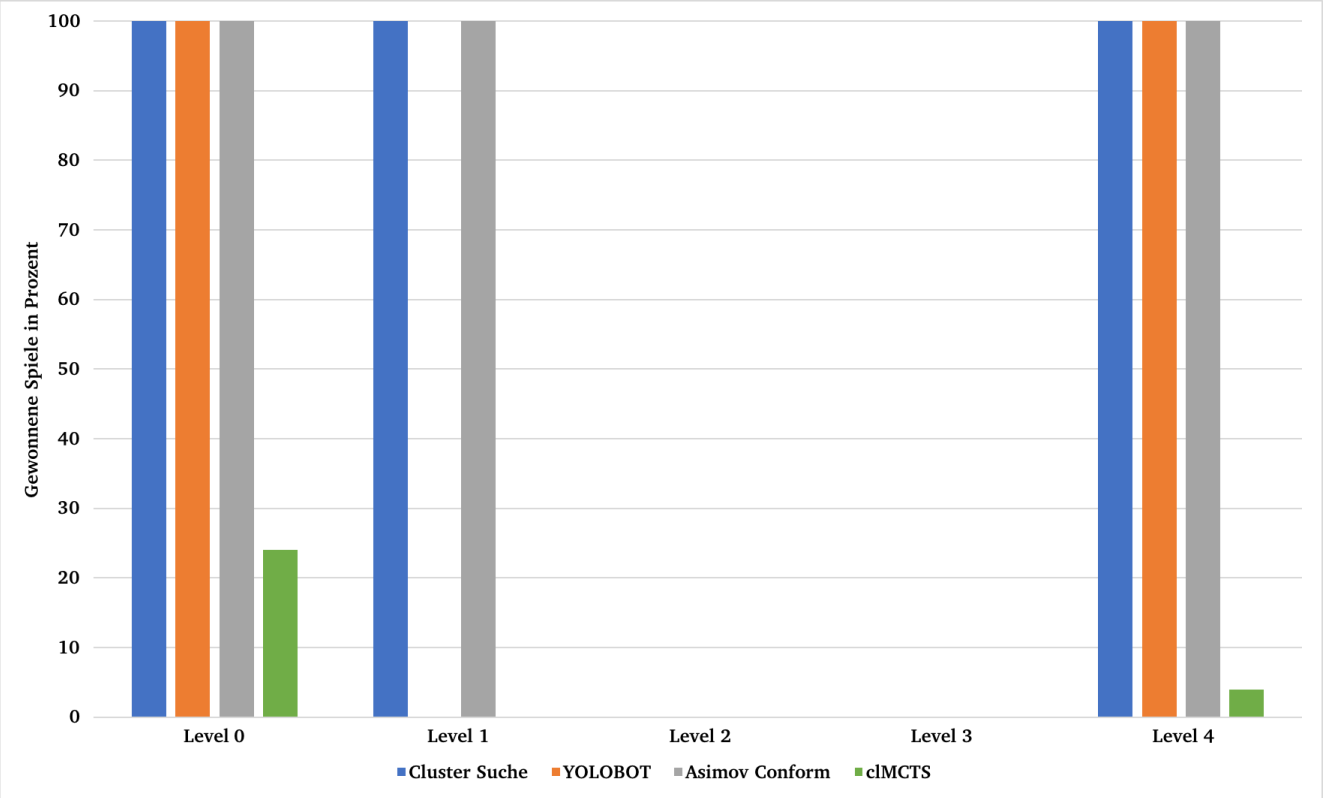

Abbildung 6.5: Brainman: Prozentsatz der gewonnenen Spiele pro Level

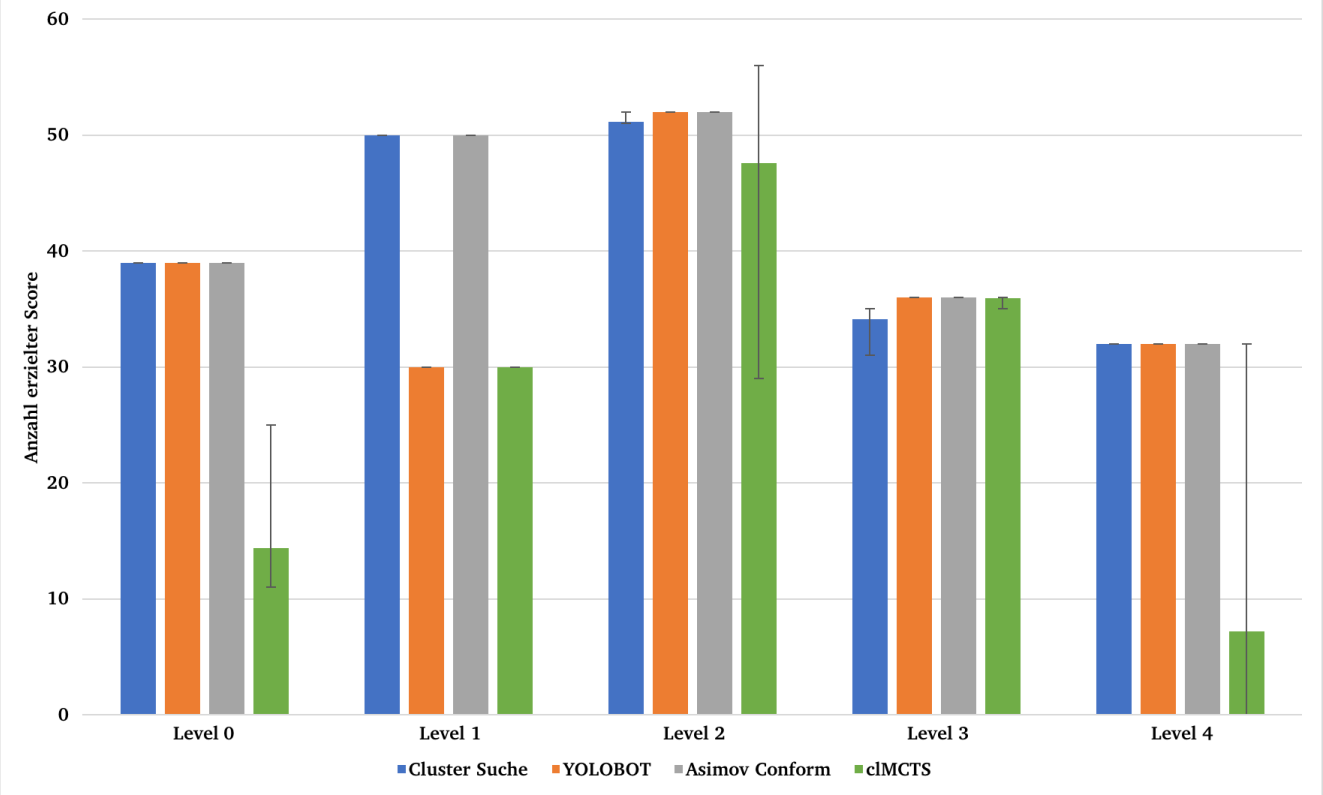

Abbildung 6.6: Brainman: Mittelwert des erzielten Scores pro Level

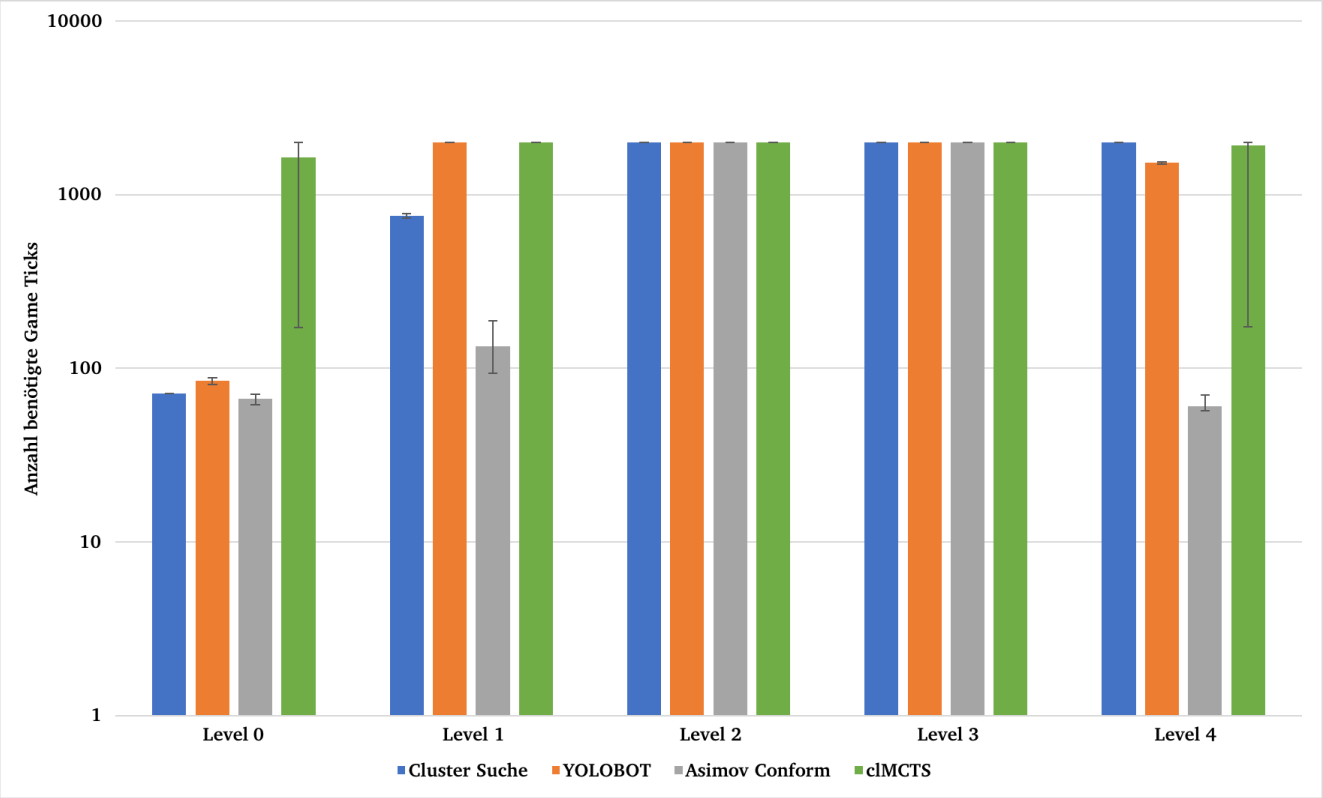

Abbildung 6.7: Brainman: Logarithmische Darstellung des Mittelwerts der benötigten Game Ticks pro Level

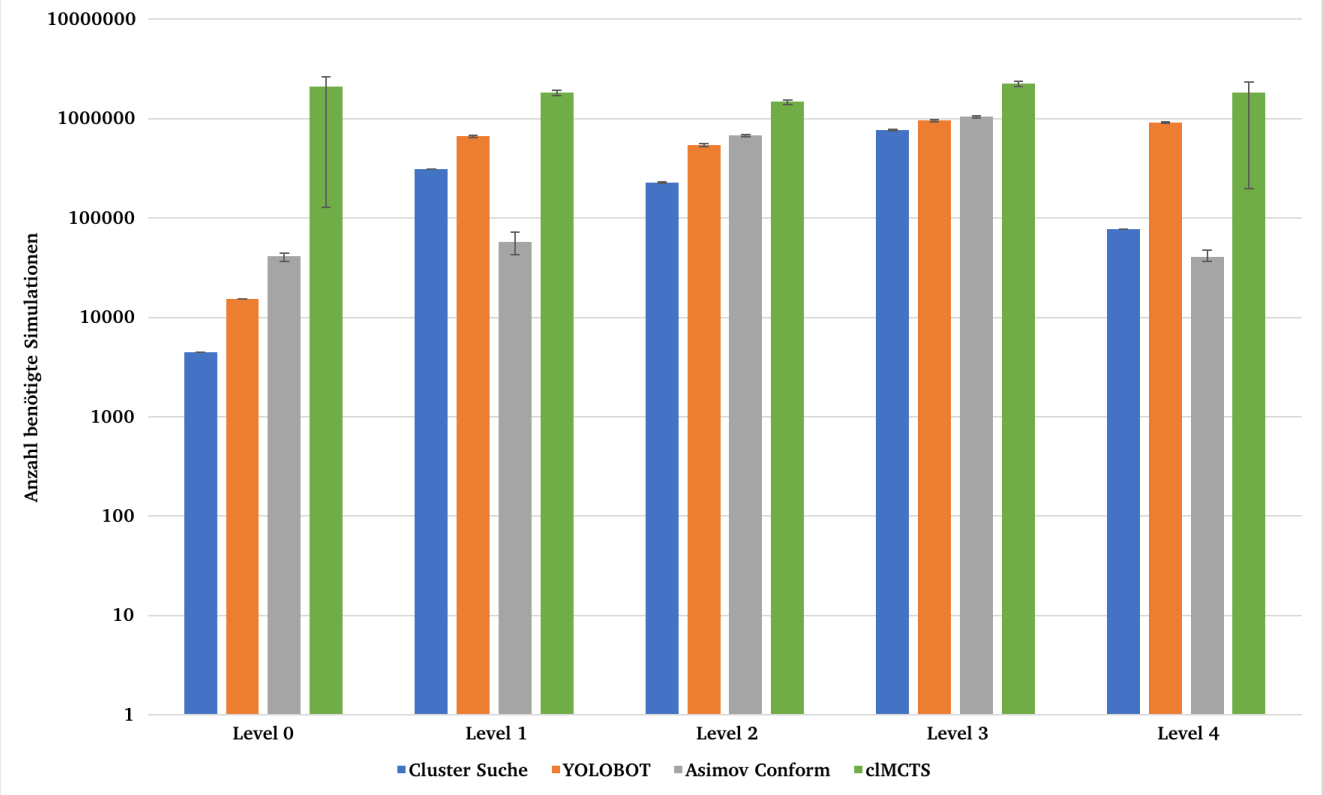

Abbildung 6.8: Brainman: Logarithmische Darstellung des Mittelwerts der benötigten Simulationen pro Level

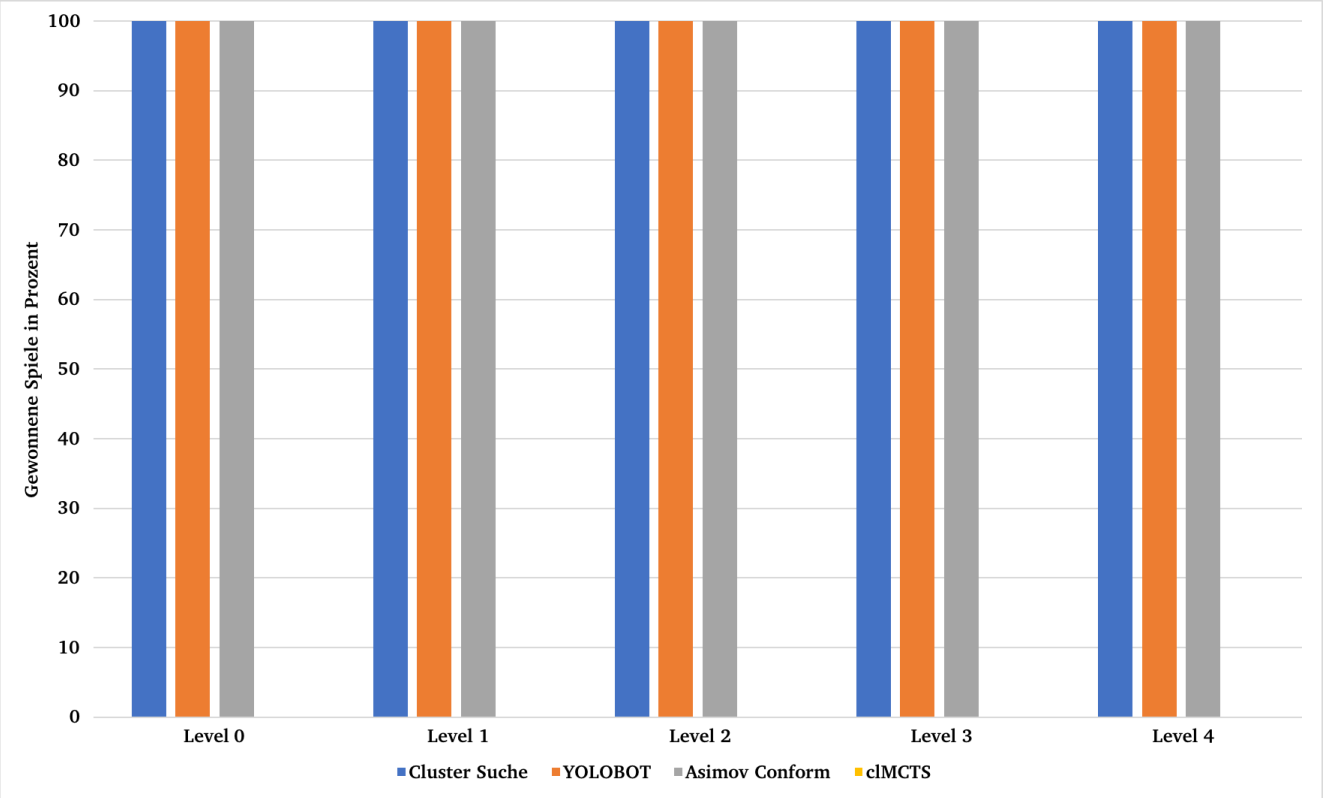

Abbildung 6.9: Catapults: Prozentsatz der gewonnenen Spiele pro Level

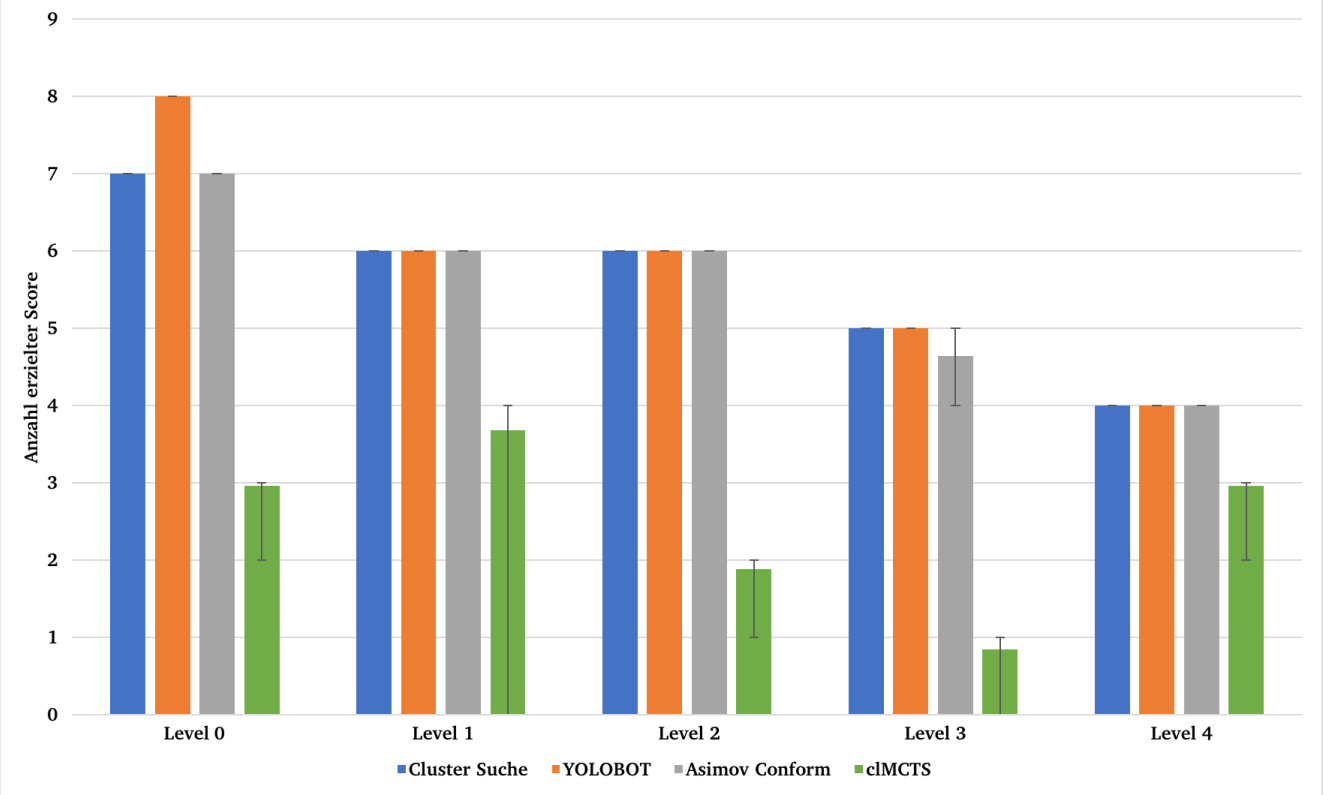

Abbildung 6.10: Catapults: Mittelwert des erzielten Scores pro Level

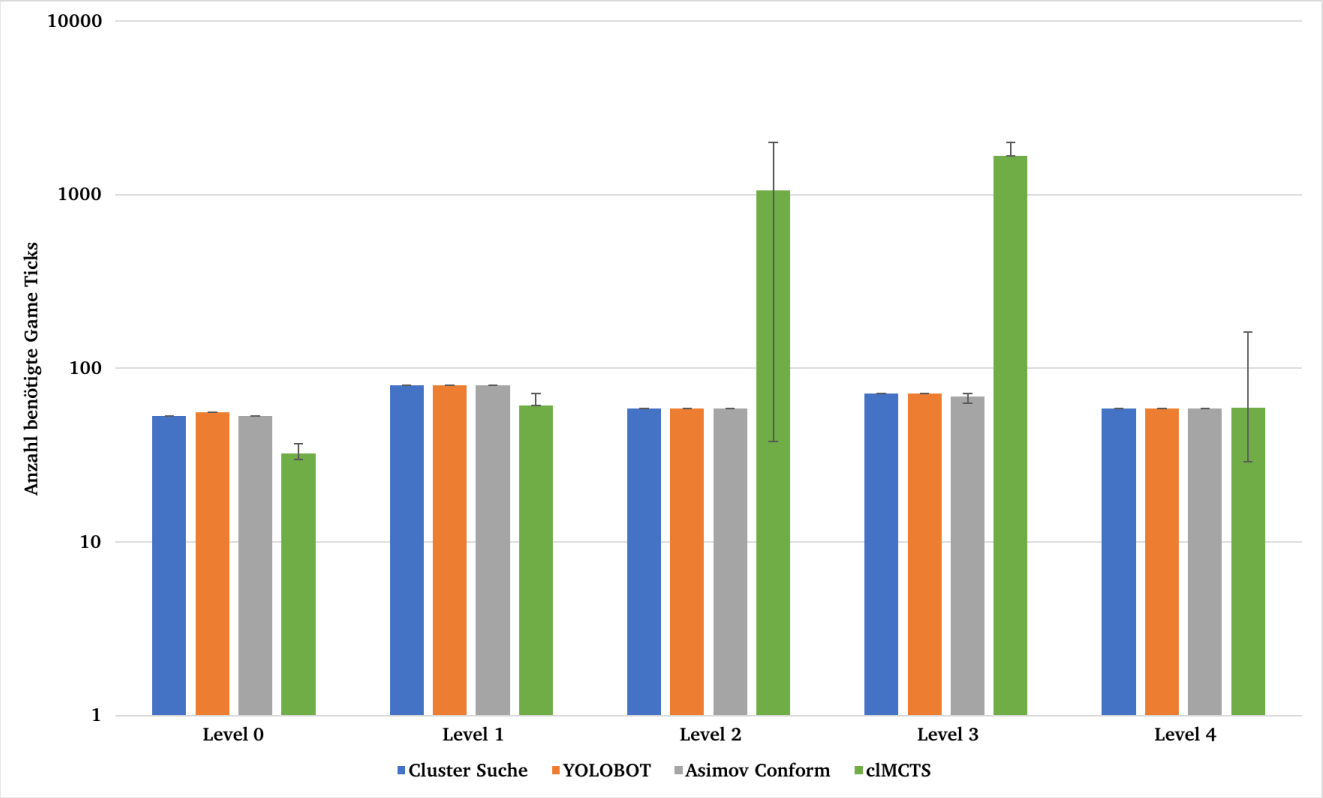

Abbildung 6.11: Catapults: Logarithmische Darstellung des Mittelwerts der benötigten Game Ticks pro Level

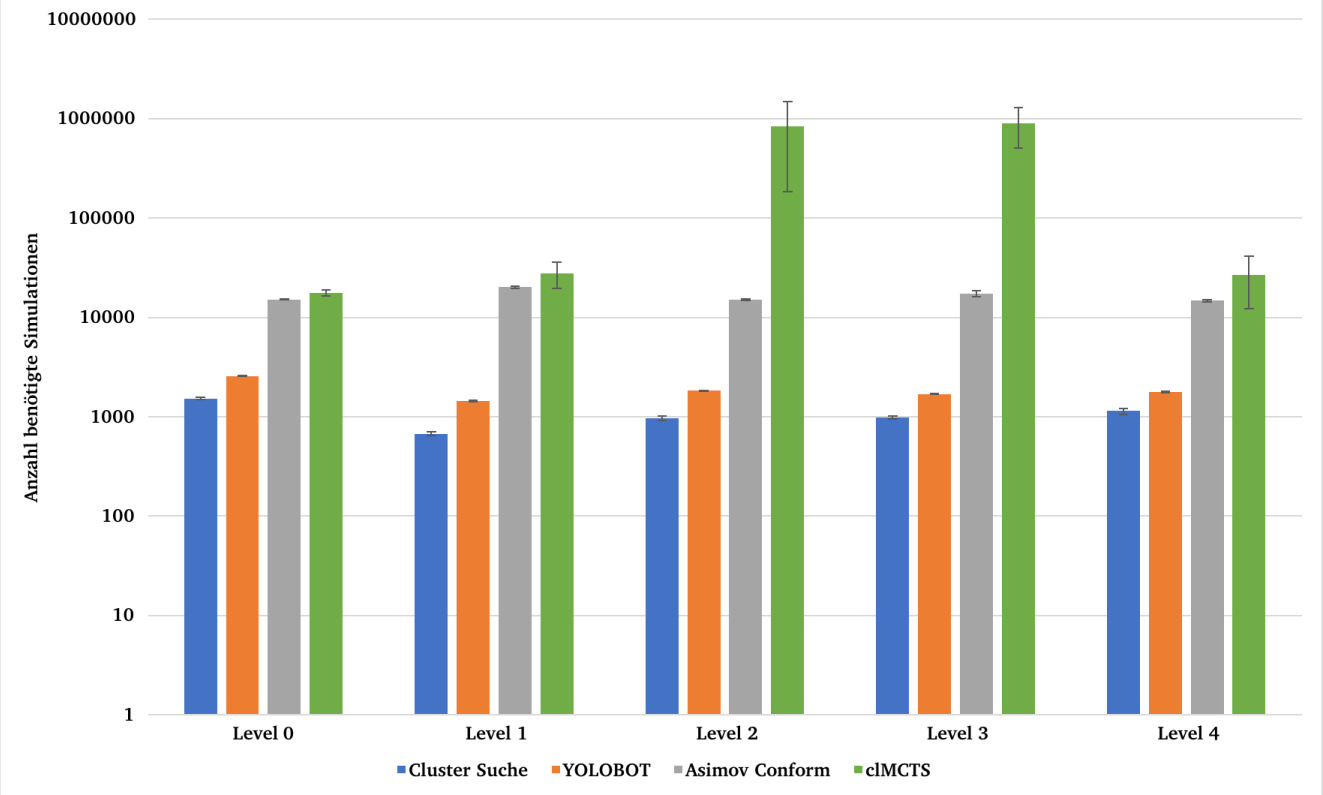

Abbildung 6.12: Catapults: Logarithmische Darstellung des Mittelwerts der benötigten Simulationen pro Level

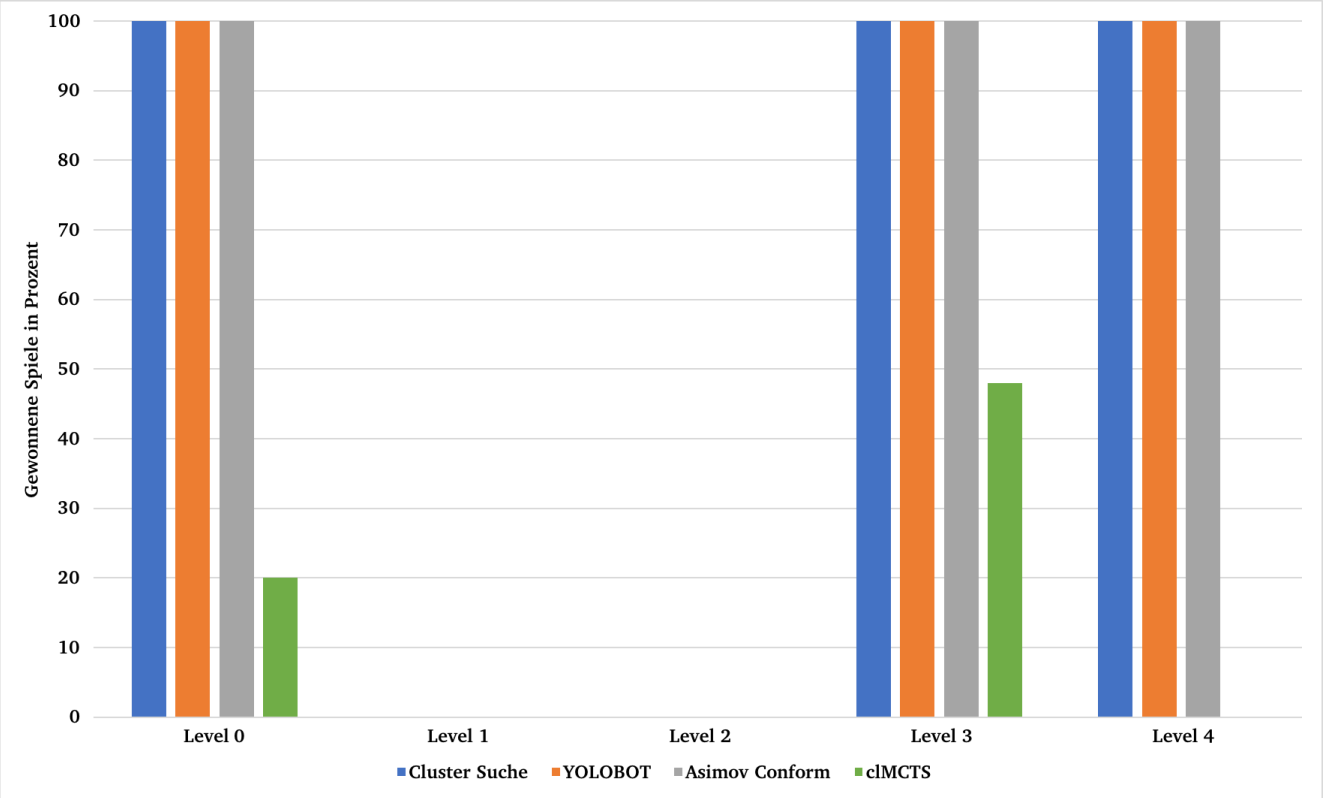

Abbildung 6.13: The Citadel: Prozentsatz der gewonnenen Spiele pro Level

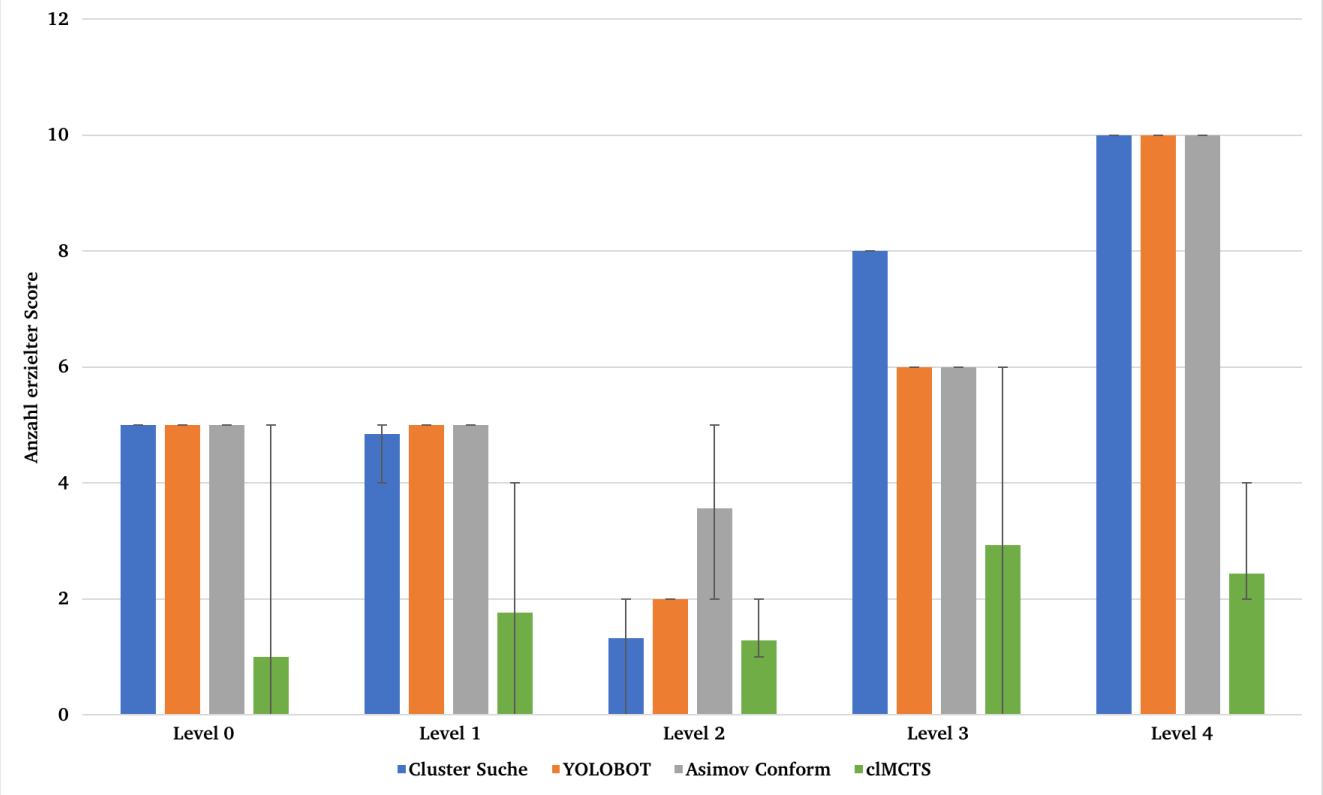

Abbildung 6.14: The Citadel: Mittelwert des erzielten Scores pro Level

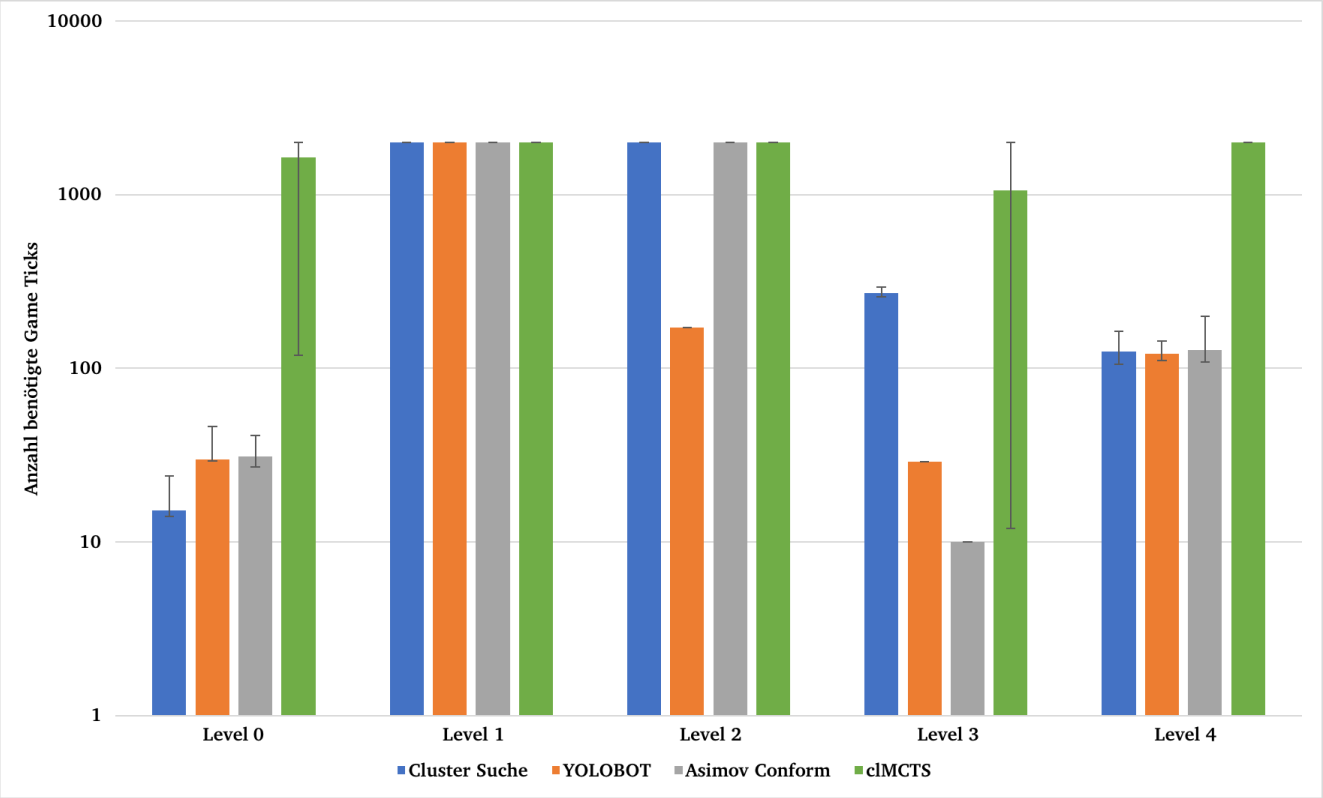

Abbildung 6.15: The Citadel: Logarithmische Darstellung des Mittelwerts der benötigten Game Ticks pro Level

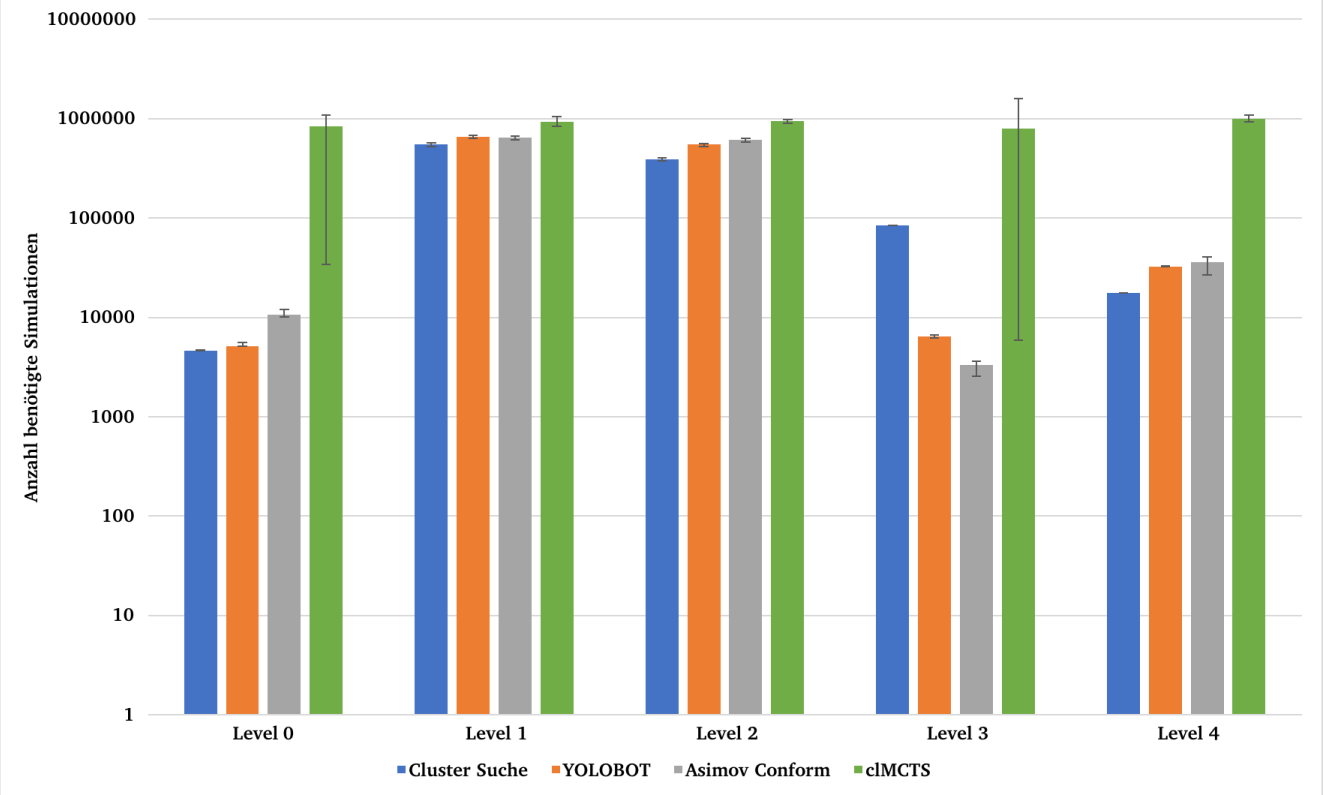

Abbildung 6.16: The Citadel: Logarithmische Darstellung des Mittelwerts der benötigten Simulationen pro Level

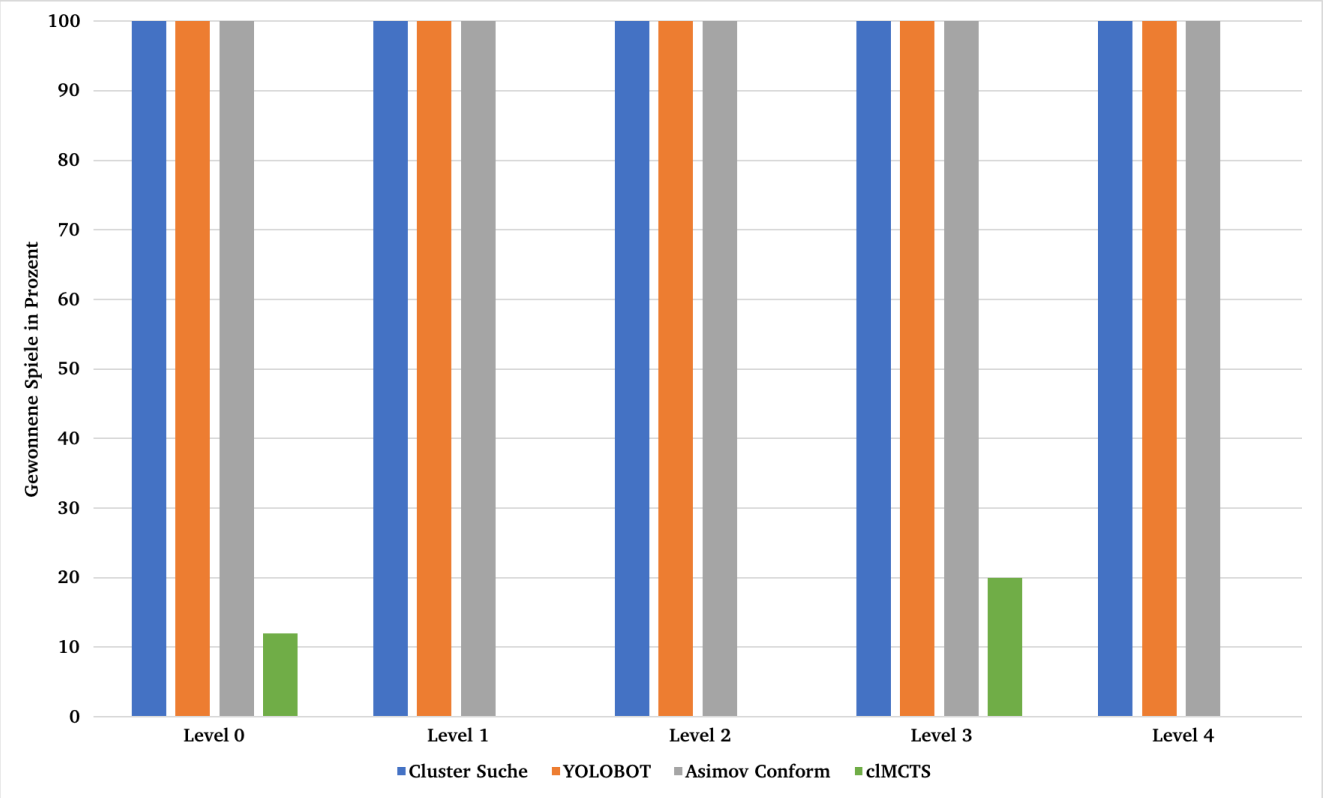

Abbildung 6.17: Labyrinth: Prozentsatz der gewonnenen Spiele pro Level

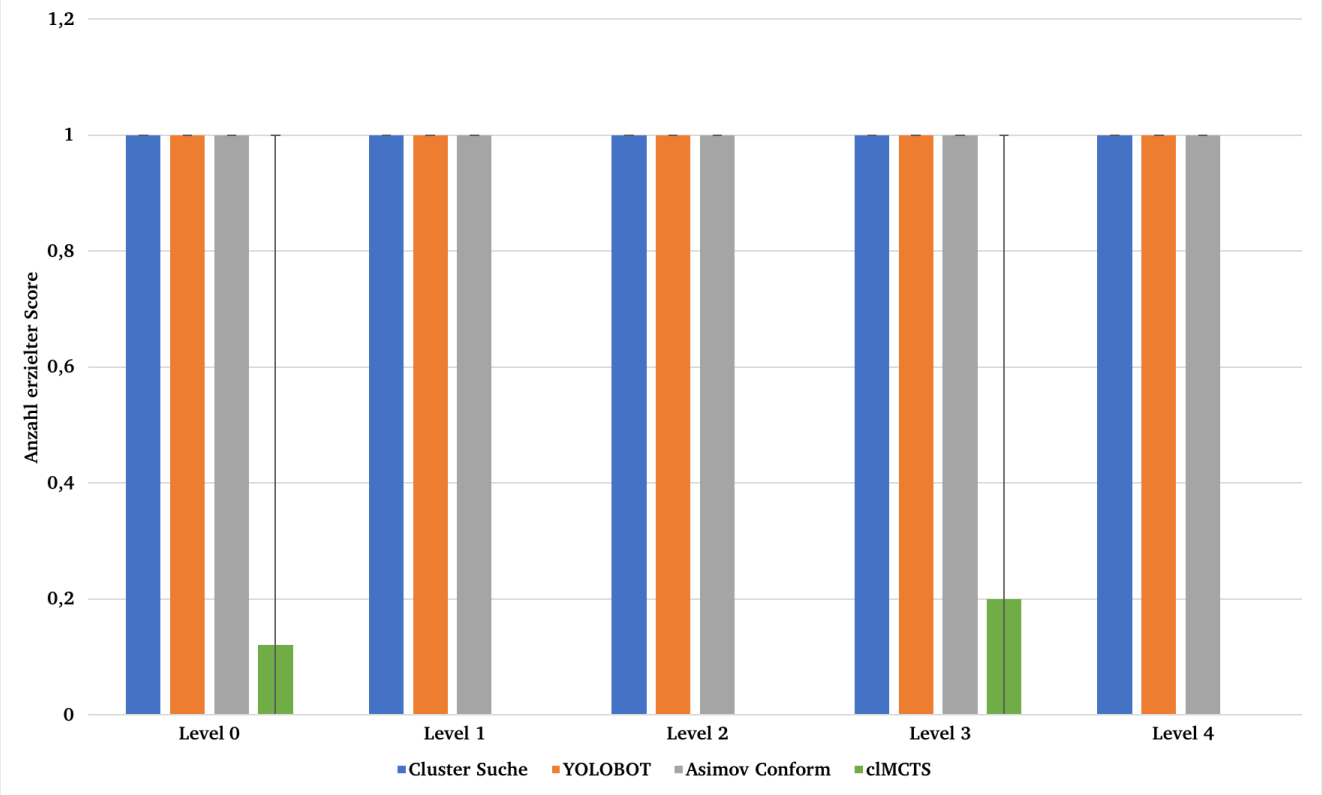

Abbildung 6.18: Labyrinth: Mittelwert des erzielten Scores pro Level

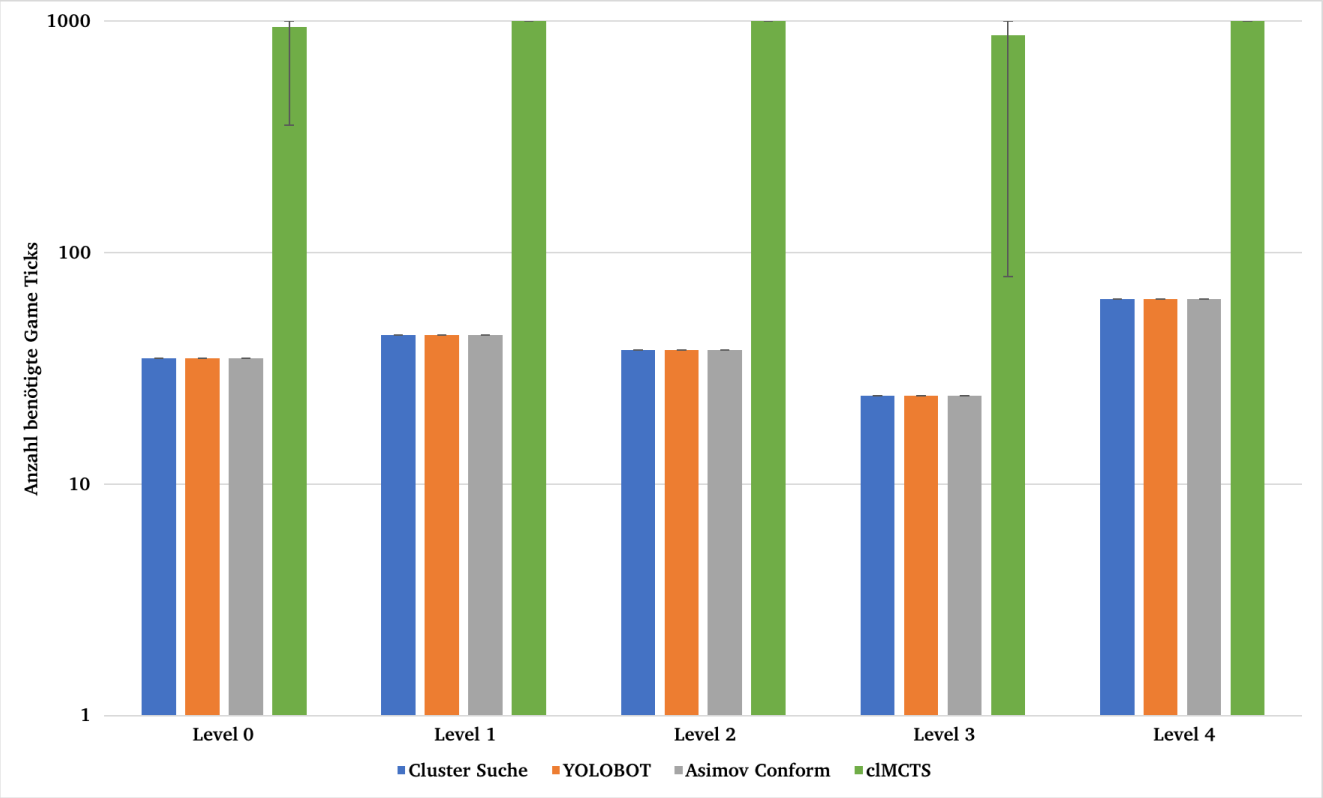

Abbildung 6.19: Labyrinth: Logarithmische Darstellung des Mittelwerts der benötigten Game Ticks pro Level

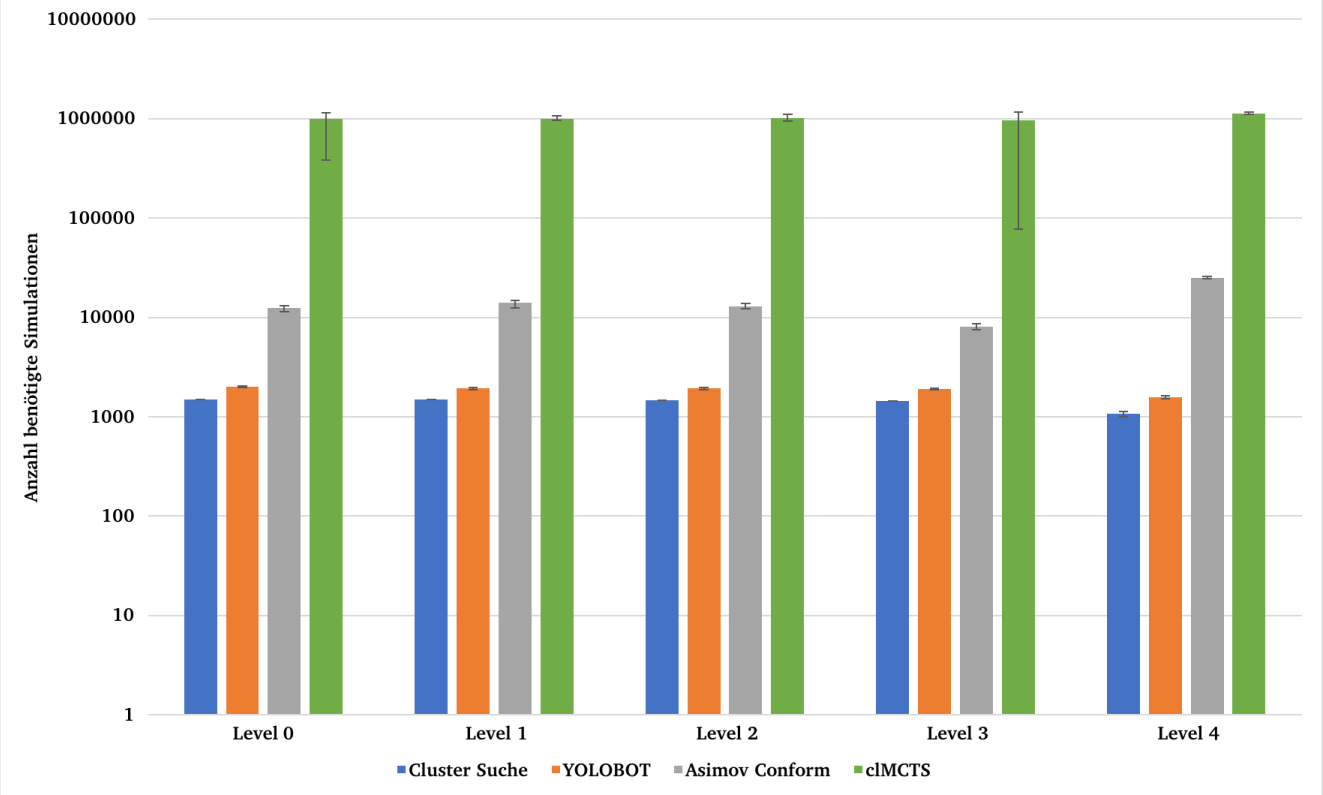

Abbildung 6.20: Labyrinth: Logarithmische Darstellung des Mittelwerts der benötigten Simulationen pro Level

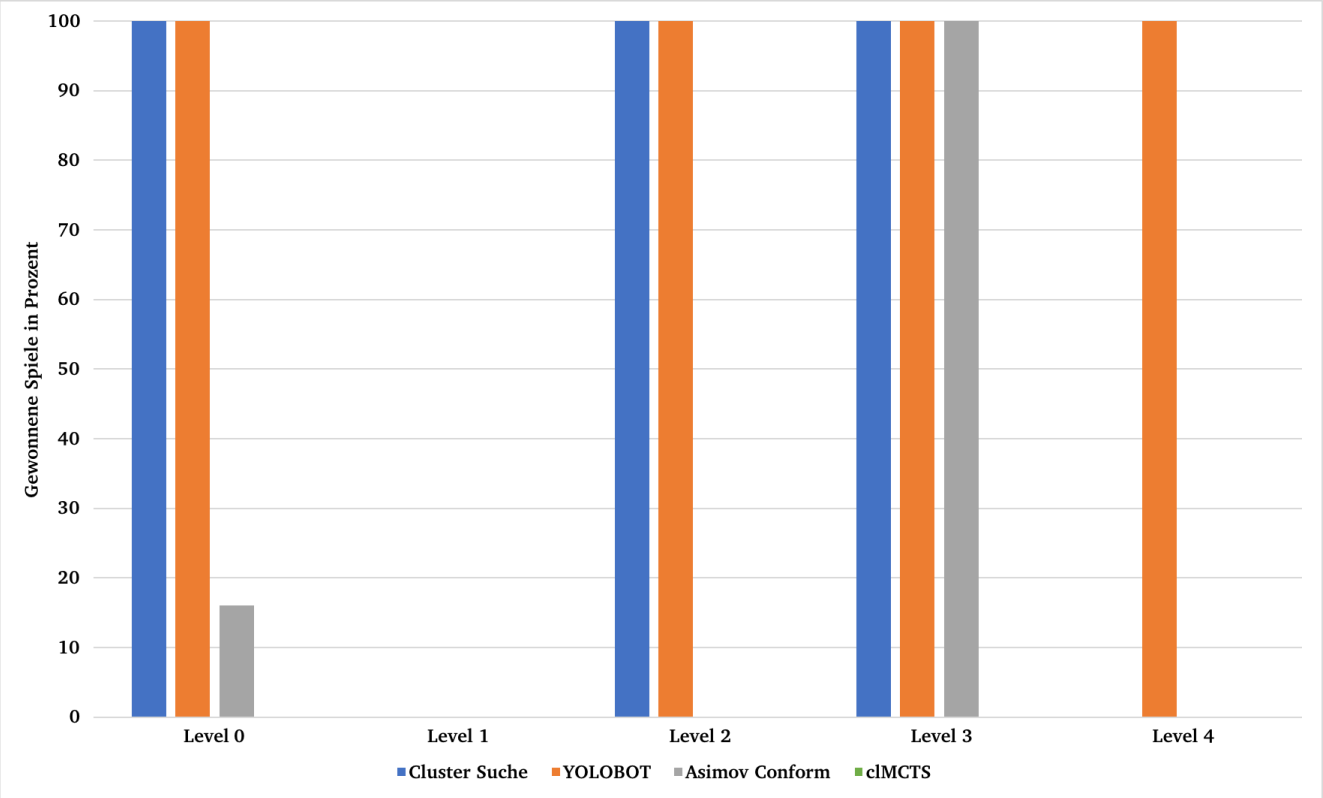

Abbildung 6.21: Real Sokoban: Prozentsatz der gewonnenen Spiele pro Level

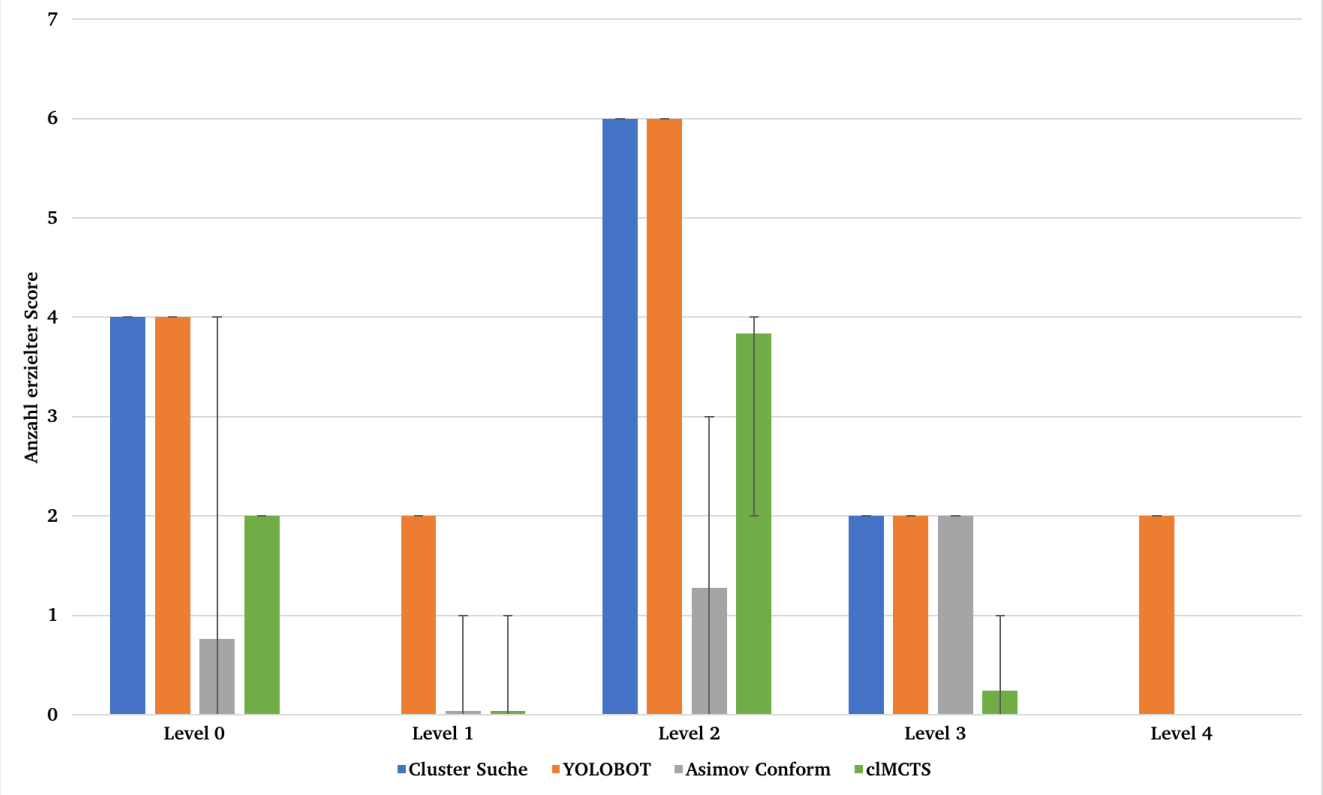

Abbildung 6.22: Real Sokoban: Mittelwert des erzielten Scores pro Level

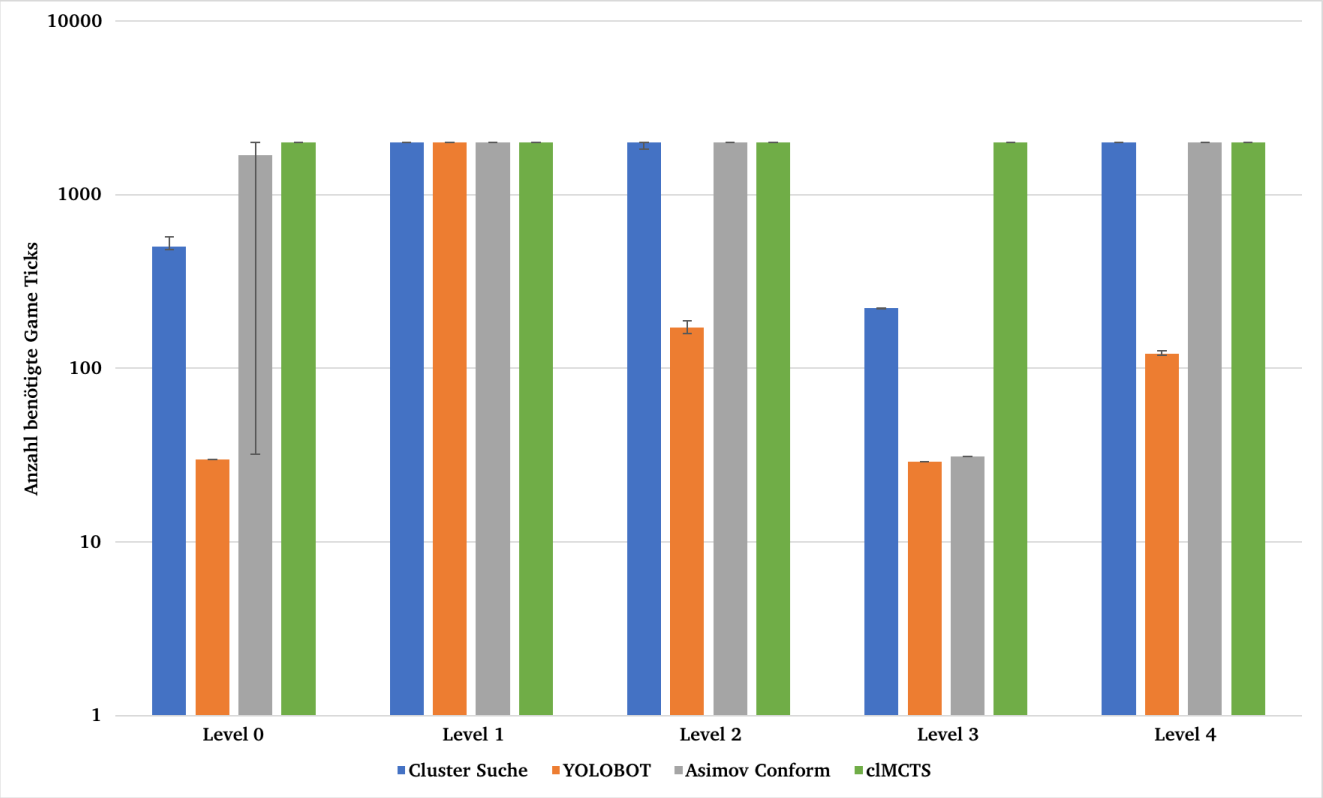

Abbildung 6.23: Real Sokoban: Logarithmische Darstellung des Mittelwerts der benötigten Game Ticks pro Level

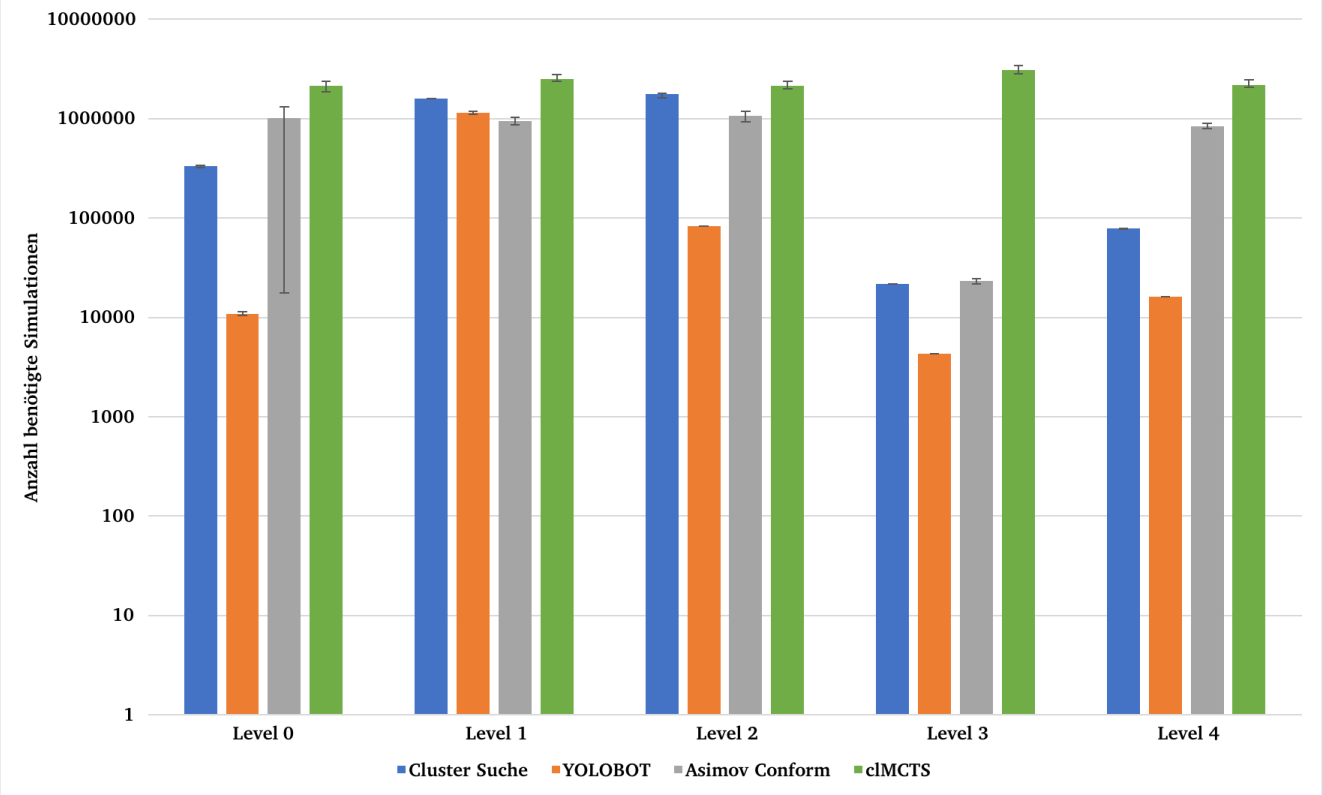

Abbildung 6.24: Real Sokoban: Logarithmische Darstellung des Mittelwerts der benötigten Simulationen pro Level

7 Fazit

In Kapitel 4 wurde der Suchalgorithmus *Cluster Suche* vorgestellt. Dieser versucht durch Clustering und Lernen von Makro-Operatoren, ein sequentielles Entscheidungsproblem zu lösen. Um die in Kapitel 1 gestellte Kernfrage zu beantworten, wurde der Algorithmus entsprechend den Angaben in Kapitel 5.1 implementiert und in einem Experiment mit anderen Algorithmen verglichen.

Dazu wurden 6 Spiele mit je 5 Leveln aus dem GVG-AI Wettbewerb verwendet. Bei den anderen Algorithmen handelt es sich um zwei auf deterministische Spiele spezialisierte Bestensuchen und einen MCTS-Algorithmus.

Im Vergleich zum nicht für deterministische Spiele ausgelegten MCTS-Algorithmus schneidet die Cluster Suche deutlich besser ab. Dieses Verhalten war jedoch zu erwarten, da der verwendete MCTS-Algorithmus jede 40 ms eine Aktion ausführt, während die Cluster Suche erst beginnt Aktionen auszuführen, wenn sie eine Lösung für ein Spiel gefunden hat oder die Zeit zur Neige geht.

Der Vergleich der Ergebnisse der Cluster Suche mit den beiden auf deterministische Spiele spezialisierten Breitensuch-Agenten, YOLOBOT und Asimov Conform, zeigt, dass Asimov Conform in den meisten Spielen den höchsten Score bei gleichzeitig niedrigsten Game Ticks holt. An zweiter Stelle befindet sich YOLOBOT, die Cluster Suche ordnet sich an dritter Stelle ein.

Jedoch ist zu bedenken, dass die Cluster Suche eine Breitensuche implementiert, die gegen Bestensuchen mit guten Heuristiken antritt. Aufgrund des Leistungsunterschiedes von Breiten- zu Bestensuche wäre ein schlechtes Abschneiden der Cluster Suche durchaus möglich. Die Cluster Suche kann jedoch durch Clustering und Makro-Operatoren viele Simulationen im Vergleich zu den Bestensuchen, einsparen.

Somit lässt sich die Frage

Kann die Idee der Cluster Suche leistungsmäßig im Vergleich zu anderen Suchalgorithmen bestehen?

mit einem vorsichtigem ja beantworten. Dies ist damit begründbar, dass eine Bestensuche selektiv die Knoten expandiert, die der Erwartung nach das beste Ergebnis erbringen. Wird dieses Verhalten auf die Cluster Suche übertragen, werden viele Makro-Operatoren, welche der Erwartung nach kein gutes Ergebnis bringen vermutlich nie expandiert. Somit lassen sich viele weitere Simulationen und damit auch Game Ticks, die auf das Suchen einer besseren Lösung verwendet werden können, einsparen.

8 Ausblick

Da die Cluster Suche vermutlich vor allem nur wegen der verwendeten Breitensuche im Vergleich zu den Bestensuchen schlechter abschneidet, kann die Idee der Cluster Suche weiter verfolgt werden.

Ein großer Nachteil der Cluster Suche ist allerdings, dass sie nicht direkt einen Suchbaum, sondern einen Graphen generiert, der in einem abschließenden Schritt auf der Suche nach der besten Lösung einmal durchlaufen werden muss. Dies könnte vermieden werden, wenn der Graph dauerhaft den *besten* Pfad bereit hält. Allerdings ist es möglich, dass eine bessere Verbindung zwischen zwei Clustern erst spät gefunden wird. Somit müsste an dieser Stelle der gesamte nachfolgende Graph aktualisiert werden. Damit stellt sich die Frage, ob dies im Vergleich zu der in dieser Arbeit vorgestellten Lösung Rechenzeit einspart.

Im Experiment wurde die Cluster Suche als eine Breitensuche implementiert. Im Fazit wird die Vermutung angestellt, dass die Cluster Suche mit einer Bestensuche als Grundlage deutlich bessere Ergebnisse liefert als eine Bestensuche ohne Clustering. Um diese Behauptung zu überprüfen, ist ein weiteres Experiment notwendig. Die Schwierigkeit hierbei besteht darin, eine geeignete Heuristik zu finden, welche mit den Makro-Operatoren funktioniert.

Der im Experiment verwendete A*-Algorithmus von YOLOBOT zum Lernen von Makro-Operatoren ist ursprünglich zur Wegfindung für einen MCTS-Algorithmus gedacht gewesen. Da dieser neben den Makro-Operatoren noch viele weiter Dinge berechnet, verbraucht der Algorithmus sehr viel Zeit um alle Makro-Operatoren für einen Cluster zu lernen. Speziell auf den Bereich deterministischer GVG-AI Spiele wäre es daher denkbar, einen Lernalgorithmus zu entwickeln, der die stochastischen Effekte außer acht lässt und somit wertvolle Rechenzeit einspart.

Literaturverzeichnis

[1] Richard P Brent. Reducing the retrieval time of scatter storage techniques. *Communications of the ACM*, 16(2):105–109, 1973.

[2] 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*, 4:1–43, 2012.

[3] Diego Perez. The gvg-ai competition. http://gvgai.net, 31.10.2017.

[4] John A Hartigan and JA Hartigan. *Clustering algorithms*, volume 209. Wiley New York, 1975.

[5] Tobias Joppen, Miriam Moneke, Nils Schröder, Christian Wirth, and Johannes Fürnkranz. Informed hybrid game tree search. Technical Report TUD–KE–2016–01, Knowledge Engineering Group, Technische Universität Darmstadt, 2016.

[6] Richard E Korf. Learning to solve problems by searching for macro-operators. Technical report, Department of Computer Science, Carnegie Mello University, 1983.

[7] Sven Oliver Krumke, Hartmut Noltemeier, and Hans-Christoph Wirth. *Graphentheoretische Konzepte und Algorithmen*. Springer, 2009.

[8] Diego Perez-Liebana, Spyridon Samothrakis, Julian Togelius, Tom Schaul, Simon M Lucas, Adrien Couëtoux, Jerry Lee, Chong-U Lim, and Tommy Thompson. The 2014 general video game playing competition. *IEEE Transactions on Computational Intelligence and AI in Games*, 8(3):229–243, 2016.

[9] Stuart Russell and Peter Norvig. *Künstliche Intelligenz, Ein moderner Ansatz*. Pearson, 2012.

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

Anhang

Tabelle A.1: Daten für den Agenten *Cluster Suche* (Minimaler und Maximaler Wert in Klammern)

Game	WinState	Score	GameTicks	Simulationen
bait 0	25	5 (5, 5)	8 (8, 8)	1279 (1279, 1279)
bait 1	25	7 (7, 7)	37 (37, 37)	2197,96 (2134, 2261)
bait 2	25	10 (10, 10)	76 (76, 76)	7726 (7726, 7726)
bait 3	0	25,48 (25, 27)	2000 (2000, 2000)	420861,12 (408450, 428183)
bait 4	25	12 (12, 12)	59,44 (58, 61)	18507,68 (18481, 18541)
brainman 0	25	39 (39, 39)	72 (72, 72)	4448 (4448, 4448)
brainman 1	25	50 (50, 50)	752,76 (734, 778)	309059,32 (308831, 309148)
brainman 2	0	51,12 (51, 52)	2000 (2000, 2000)	226875,56 (222948, 230514)
brainman 3	0	34,12 (31, 35)	2000 (2000, 2000)	771861,32 (758047, 788080)
brainman 4	25	32 (32, 32)	1999 (1999, 1999)	77266 (77266, 77266)
catapults 0	25	7 (7, 7)	53 (53, 53)	1527,32 (1484, 1569)
catapults 1	25	6 (6, 6)	80 (80, 80)	674,76 (651, 704)
catapults 2	25	6 (6, 6)	59 (59, 59)	966,64 (919, 1017)
catapults 3	25	5 (5, 5)	72 (72, 72)	984,8 (951, 1024)
catapults 4	25	4 (4, 4)	59 (59, 59)	1144,36 (1058, 1214)
labyrinth 0	25	1 (1, 1)	35 (35, 35)	1485 (1485, 1485)
labyrinth 1	25	1 (1, 1)	44 (44, 44)	1485 (1485, 1485)
labyrinth 2	25	1 (1, 1)	38 (38, 38)	1467 (1467, 1467)
labyrinth 3	25	1 (1, 1)	24 (24, 24)	1440 (1440, 1440)
labyrinth 4	25	1 (1, 1)	63 (63, 63)	1064,28 (999, 1127)
realsokoban 0	25	4 (4, 4)	503,4 (484, 573)	334608,48 (322155, 339827)
realsokoban 1	0	0 (0, 0)	2000 (2000, 2000)	1604928 (1603761, 1605731)
realsokoban 2	25	6 (6, 6)	1992,12 (1827, 1999)	1766157,8 (1625064, 1815894)
realsokoban 3	25	2 (2, 2)	222,12 (221, 223)	21660,24 (21658, 21662)
realsokoban 4	0	0 (0, 0)	2000 (2000, 2000)	78141,2 (77885, 78466)
thecitadel 0	25	5 (5, 5)	15,16 (14, 24)	4652,08 (4650, 4668)
thecitadel 1	0	4,84 (4, 5)	2000 (2000, 2000)	548064,08 (522004, 567614)
thecitadel 2	0	1,32 (0, 2)	2000 (2000, 2000)	388500,2 (376379, 402685)
thecitadel 3	25	8 (8, 8)	270,4 (258, 295)	84096,8 (84072, 84146)
thecitadel 4	25	10 (10, 10)	125,04 (106, 163)	17548,08 (17510, 17624)

Tabelle A.2: Daten für den Agenten *YOLOBOT* (Minimaler und Maximaler Wert in Klammern)

Game	WinState	Score	GameTicks	Simulationen
bait 0	25	5 (5, 5)	8 (8, 8)	1328 (1328, 1328)
bait 1	25	7 (7, 7)	37 (37, 37)	7988,16 (7056, 8539)
bait 2	25	9,04 (9, 10)	62,32 (61, 66)	9616,36 (9017, 9645)
bait 3	0	27 (27, 27)	2000 (2000, 2000)	968909,64 (942821, 996609)
bait 4	25	12 (12, 12)	42 (42, 42)	12129,48 (11088, 12718)
brainman 0	25	39 (39, 39)	84,64 (81, 88)	15277,92 (15236, 15370)
brainman 1	0	30 (30, 30)	2000 (2000, 2000)	664362,6 (648975, 674127)
brainman 2	0	52 (52, 52)	2000 (2000, 2000)	542927 (519745, 566079)
brainman 3	0	36 (36, 36)	2000 (2000, 2000)	955010,68 (926629, 975781)
brainman 4	25	32 (32, 32)	1531,12 (1506, 1550)	916880,6 (904076, 932591)
catapults 0	25	8 (8, 8)	56 (56, 56)	2562,76 (2504, 2603)
catapults 1	25	6 (6, 6)	80 (80, 80)	1432,72 (1398, 1470)
catapults 2	25	6 (6, 6)	59 (59, 59)	1822,28 (1779, 1877)
catapults 3	25	5 (5, 5)	72 (72, 72)	1697,6 (1655, 1748)
catapults 4	25	4 (4, 4)	59 (59, 59)	1774,28 (1702, 1829)
labyrinth 0	25	1 (1, 1)	35 (35, 35)	2001,36 (1974, 2035)
labyrinth 1	25	1 (1, 1)	44 (44, 44)	1926 (1878, 1973)
labyrinth 2	25	1 (1, 1)	38 (38, 38)	1937,2 (1855, 1982)
labyrinth 3	25	1 (1, 1)	24 (24, 24)	1914,96 (1877, 1950)
labyrinth 4	25	1 (1, 1)	63 (63, 63)	1577,72 (1517, 1636)
realsokoban 0	25	4 (4, 4)	30 (30, 30)	10906,16 (10387, 11472)
realsokoban 1	0	2 (2, 2)	2000 (2000, 2000)	1144456,2 (1098022, 1183869)
realsokoban 2	25	6 (6, 6)	172,44 (159, 187)	83239,88 (83213, 83269)
realsokoban 3	25	2 (2, 2)	29 (29, 29)	4307 (4307, 4307)
realsokoban 4	25	2 (2, 2)	121,36 (119, 126)	16246,72 (16242, 16256)
thecitadel 0	25	5 (5, 5)	14,68 (14, 31)	5162,6 (5099, 5643)
thecitadel 1	0	5 (5, 5)	2000 (2000, 2000)	655157,12 (634414, 675596)
thecitadel 2	0	2 (2, 2)	2000 (2000, 2000)	548289,36 (527550, 563463)
thecitadel 3	25	6 (6, 6)	14 (14, 14)	6464,64 (6213, 6673)
thecitadel 4	25	10 (10, 10)	164,72 (154, 187)	32664,44 (32643, 32709)

Tabelle A.3: Daten für den Agenten *Asimov Conform* (Minimaler und Maximaler Wert in Klammern)

Game	WinState	Score	GameTicks	Simulationen
bait 0	25	5 (5, 5)	8 (8, 8)	3998,92 (3837, 4161)
bait 1	25	7 (7, 7)	37 (37, 37)	16949,48 (16245, 17673)
bait 2	25	11 (11, 11)	87,36 (86, 89)	57621,16 (55201, 60820)
bait 3	24	32,56 (30, 36)	878,68 (147, 2000)	418939,56 (70888, 963370)
bait 4	25	12 (12, 12)	42,48 (42, 44)	37446,76 (34817, 40632)
brainman 0	25	39 (39, 39)	67,04 (62, 71)	41478,44 (36587, 44566)
brainman 1	25	50 (50, 50)	134,4 (94, 188)	57539,44 (42738, 71966)
brainman 2	0	52 (52, 52)	2000 (2000, 2000)	680091,6 (659282, 695703)
brainman 3	0	36 (36, 36)	2000 (2000, 2000)	1042230,2 (1016408, 1076494)
brainman 4	25	32 (32, 32)	60,48 (57, 70)	40924,28 (36761, 47258)
catapults 0	25	7 (7, 7)	53 (53, 53)	15178,92 (14872, 15460)
catapults 1	25	6 (6, 6)	80 (80, 80)	20130,8 (19184, 21319)
catapults 2	25	6 (6, 6)	59 (59, 59)	15045,16 (14231, 15568)
catapults 3	25	4,64 (4, 5)	68,76 (63, 72)	17396,52 (15356, 19463)
catapults 4	25	4 (4, 4)	59 (59, 59)	14751,56 (13797, 15324)
labyrinth 0	25	1 (1, 1)	35 (35, 35)	12555,64 (11491, 13227)
labyrinth 1	25	1 (1, 1)	44 (44, 44)	14152,32 (12548, 15131)
labyrinth 2	25	1 (1, 1)	38 (38, 38)	12827,8 (12189, 13471)
labyrinth 3	25	1 (1, 1)	24 (24, 24)	8025 (7485, 8661)
labyrinth 4	25	1 (1, 1)	63 (63, 63)	25074,44 (24520, 26182)
realsokoban 0	4	0,76 (0, 4)	1685,12 (32, 2000)	1011263,52 (17587, 1311430)
realsokoban 1	0	0,04 (0, 1)	2000 (2000, 2000)	943820,56 (872478, 1035949)
realsokoban 2	0	1,28 (0, 3)	2000 (2000, 2000)	1060994,16 (921740, 1176457)
realsokoban 3	25	2 (2, 2)	31 (31, 31)	23310,08 (21900, 24552)
realsokoban 4	0	0 (0, 0)	2000 (2000, 2000)	830971,76 (790703, 893787)
thecitadel 0	25	5 (5, 5)	31,2 (27, 41)	10668,4 (10059, 12056)
thecitadel 1	0	5 (5, 5)	2000 (2000, 2000)	650816,68 (611579, 665505)
thecitadel 2	0	3,56 (2, 5)	2000 (2000, 2000)	607282,52 (580391, 629615)
thecitadel 3	25	6 (6, 6)	10 (10, 10)	3320,12 (2561, 3605)
thecitadel 4	25	10 (10, 10)	127,08 (109, 199)	36138,04 (26678, 40544)

Tabelle A.4: Daten für den Agenten *clMCTS* (Minimaler und Maximaler Wert in Klammern)

Game	WinState	Score	GameTicks	Simulationen
bait 0	14	2,8 (0, 5)	890,4 (11, 2000)	1322367,08 (13538, 3180136)
bait 1	0	0 (0, 0)	1999,96 (1999, 2000)	1810096,8 (1698954, 1938518)
bait 2	0	0,04 (0, 1)	1999,84 (1996, 2000)	1784548,32 (1676052, 1901766)
bait 3	0	1,76 (0, 5)	1998,8 (1995, 2000)	1097618,84 (1076389, 1157671)
bait 4	0	1,208333333 (0, 4)	1999,791667 (1998, 2000)	1972620,25 (1691451, 2205921)
brainman 0	6	14,36 (11, 25)	1643,32 (171, 2000)	2099360,92 (127902, 2641369)
brainman 1	0	30 (30, 30)	2000 (2000, 2000)	1818780,56 (1710361, 1924337)
brainman 2	0	47,56 (29, 56)	2000 (2000, 2000)	1477887,68 (1387299, 1547892)
brainman 3	0	35,96 (35, 36)	2000 (2000, 2000)	2260481,6 (2117256, 2388792)
brainman 4	1	7,2 (0, 32)	1926,92 (173, 2000)	1833344,96 (199140, 2336111)
catapults 0	0	2,96 (2, 3)	32,44 (30, 37)	17667,44 (16144, 20767)
catapults 1	0	3,68 (0, 4)	61,12 (0, 72)	27679,68 (364, 31831)
catapults 2	0	1,88 (1, 2)	1055,28 (38, 2000)	841320,44 (13793, 1660354)
catapults 3	0	0,84 (0, 1)	1679,6 (0, 2000)	902091,92 (541, 1129217)
catapults 4	0	2,96 (2, 3)	59,28 (29, 162)	26691,2 (10824, 82043)
labyrinth 0	3	0,12 (0, 1)	943,72 (354, 1000)	1003368,92 (381357, 1138572)
labyrinth 1	0	0 (0, 0)	1000 (1000, 1000)	1004004,76 (955570, 1072905)
labyrinth 2	0	0 (0, 0)	1000 (1000, 1000)	1010474,84 (942875, 1100171)
labyrinth 3	5	0,2 (0, 1)	870,36 (79, 1000)	962081,12 (77906, 1160951)
labyrinth 4	0	0 (0, 0)	1000 (1000, 1000)	1133315,36 (1105401, 1156581)
realsokoban 0	0	2 (2, 2)	2000 (2000, 2000)	2130892,96 (1860506, 2376476)
realsokoban 1	0	0,04 (0, 1)	2000 (2000, 2000)	2523825,4 (2373628, 2787390)
realsokoban 2	0	3,84 (2, 4)	2000 (2000, 2000)	2127843,6 (1987776, 2364841)
realsokoban 3	0	0,24 (0, 1)	2000 (2000, 2000)	3096325,96 (2845978, 3437749)
realsokoban 4	0	0 (0, 0)	2000 (2000, 2000)	2166911,4 (2073649, 2461580)
thecitadel 0	5	1 (0, 5)	1635,88 (119, 2000)	838137,32 (33993, 1081168)
thecitadel 1	0	1,76 (0, 4)	2000 (2000, 2000)	937455,12 (834878, 1046235)
thecitadel 2	0	1,28 (1, 2)	2000 (2000, 2000)	943733,44 (890725, 984826)
thecitadel 3	12	2,92 (0, 6)	1055,2 (12, 2000)	794021,92 (5908, 1590719)
thecitadel 4	0	2,44 (2, 4)	2000 (2000, 2000)	996192,08 (933252, 1088885)