Monte Carlo Tree Search Optimierungen für Sokoban Puzzle Generierung

Bachelorarbeit

Lukas Koczorowski
410835

11. Dezember 2023

Betreuer: Prof. Dr. Benjamin Blankertz
Zweitbetreuer: Dr. Stefan Fricke

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

Kurzfassung

Sokoban ist ein Puzzlespiel, welches ein eigenes Genre geprägt hat. In dieser Arbeit wird die Generierung von Sokoban-Leveln mittels simuliertem Spielen mithilfe vom *Monte Carlo Tree Search* betrachtet.
Ziel der Arbeit ist es auf die Implementierung eines solchen Algorithmus einzugehen, sowie dem Erstellen von nützlichen Optimierungen. Es werden Optimierungen vorgestellt und in Experimenten verglichen. Ein großer Fokus dieser Optimierungen ist die Verringerung von unnötigen Aktionen. Auch werden Speicherverwaltung, UCT Erweiterungen und weitere Ideen betrachtet.
Es werden ebenfalls Bewertungsmetriken für die Levelerstellung betrachtet und erweitert.
Am Ende kann festgestellt werden, dass viele der vorgestellten Optimierungen nützlich sind und das Erstellen von vielen Leveln in sehr kurzer Zeit ermöglichen.

Abstract

Sokoban is a puzzle game which shaped its own genre. The focus of this work is the generation of sokoban levels through simulated gameplay with *Monte Carlo Tree Search*.
The aim of this work is to go in detail for the implementation of such an algorithm and to also produce useful optimizations. These optimizations will be presented and compared in experiments. A great focus of these optimizations is the removal of pointless actions. On top of that memory management, uct enhancements and more will be presented and discussed.
Reward metrics for level generation will also be discussed and further evaluated.
This work was able to produce useful optimizations and to make an implementation which is able to produce many levels in a short amount of time.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Sokoban

Sokoban ist ein Computerspiel, welches auf einem $n \times m$ Feld spielt. Auf diesem Feld befinden sich Boxen und Zielpositionen. Der Spieler darf sich horizontal und vertikal bewegen und muss eine jede Box auf eine Zielposition bringen, um das Level zu lösen. Der Spieler kann die Boxen schieben indem er sich gegen die Boxen bewegt. Auch beim Schieben bewegt sich der Spieler mit. Der Spieler kann maximal eine Box auf einmal bewegen. Außerdem existieren Blöcke, welche unüberwindbare Hindernisse sind für den Spieler und die Boxen. Sokoban ist zudem PSPACE-Vollständig [5].

Abbildung 1.1 zeigt ein beschriftetes Beispiel eines Sokobanlevels für die verwendeten Begriffe in der Arbeit. Die Texturen für die Erstellung der Abbildungen von Sokoban-Leveln in dieser Arbeit stammen von [9]. Dieselben Texturen wurden auch im Programmierprojekt verwendet.

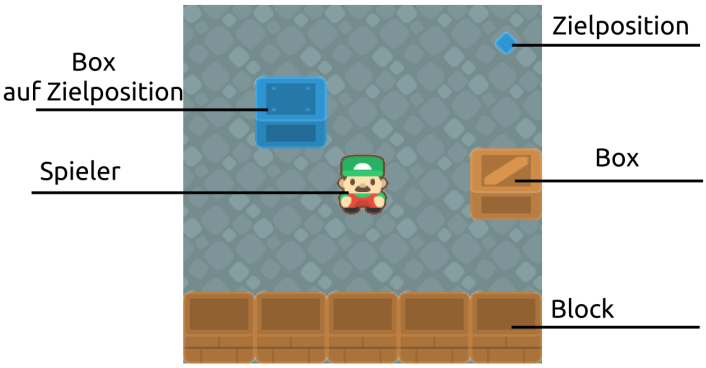

Abbildung 1.1: Beschriftung Sokoban

Texturen aus [9]

1.2 Problem

Die Levelerstellung in Spielen ist normalerweise ein Prozess der sehr händisch erfolgt.
Wenn in Computerspielen Algorithmen zur Levelerstellung verwendet werden, dann entweder bei der Produzierung des Spiels selber, oder dynamisch im laufenden Spiel. Gerade im letzteren Fall ist die Laufzeit und Speichereffizienz wichtig. Dahingehend kann man an der angegeben Literatur auch sehen, dass diese Form von Generierungen gerade bei Sokoban oftmals an einer sehr langsamen Laufzeit leiden. Zudem können algorithmisch generierte Level oft unter Repetition und einem Mangel von

Schwierigkeit leiden.

1.3 Literatur

Levelgenerierung im Generellen und auch bei Sokoban erfolgen in einem ähnlichen Muster wie in [12]. Dort hat man sich *Templates* erstellt und diese mittels Algorithmen kombiniert und dann belegt mit Objekten. Solche Methoden tendieren dazu, Brute-Force-Methoden zu sein.
Eine generelle Analyse hinsichtlich *Procedural-Content-Generation* findet sich in [13].
Der erste Ansatz für das Generieren von Sokoban-Leveln mit simuliertem Spielen kam von [8]. Die grundlegende Form des Algorithmus ist die Formalisierung der Levelgenerierung in einer Form, welche es erlaubt den UCT anzuwenden.
Die Verfeinerung des ganzen erfolgte in [7], dort hat man den Algorithmus leicht verändert und Metriken angepasst. Zudem haben sie eine Studie durchgeführt, in der sie versucht haben, eine tatsächliche Korrelation zwischen den Metriken und dem empfundenen Schwierigkeitsgrad aufzustellen. In dieser Arbeit gab es keine Pseudocode Angaben und die Erklärungen für bestimmte Aktionen sind nicht optimal. Zudem wurde dort auf keine Optimierungen eingegangen. Für die Erstellung der höchstbewerteten Level kann es mehrere Minuten dauern.
In dieser Arbeit wir die Form der Levelgenerierung aus [7] möglichst nah nacherstellt. Eine genaue Replikation der Bewertungen für die fünf Beispiellevel aus [7] war nicht möglich, jedoch wurden die Metriken alle so erstellt und parametrisiert, dass diese möglichst nah an die Level herankommen.
[3] war die Hauptliteratur für den *Monte Carlo Tree Search* beziehungsweise MCTS und den UCT. Für bestimmte Erweiterungen vom UCT gibt es die jeweilige Lektüre [2], [1]. In [3] werden ebenfalls generische Strategien für das Optimieren von MCTS vorgestellt.

1.4 Ziel der Arbeit

Das Ziel der Arbeit ist es interessante Sokoban-Level möglichst speichereffizient und schnell zu berechnen. Generiert werden diese Level mittels simulierten Spielen in einer Form, welche es erlaubt den *Monte Carlo Tree Search* anzuwenden.
Hierbei werden für diese spezifische Implementierung vom *Monte Carlo Tree Search* vielfältige Optimierungsmethoden betrachtet und evaluiert.

1.5 Struktur

Diese Arbeit beginnt mit einer kurzen Einleitung zum *Monte Carlo Tree Search* und dessen Variante, welche für die Arbeit wichtig ist. Hier werden ebenfalls die grundlegenden Funktionen eingeleitet, welche später erweitert werden.
Daraufhin werden die drei Grundelemente und zwar Zustände, Aktionen und Bewertungen betrachtet. Dies zeigt die Formulierung des Problems in einer Form, in welcher es anwendbar für den *Monte Carlo Tree Search* ist. Hierbei wird ebenfalls auf Anpassungen eingegangen.
Nach dem theoretischen Hintergrund und der generellen Struktur des Algorithmus folgt der Kern der Arbeit und zwar die Optimierungen. Bei diesen werden Optimierungen vorgestellt, welche spezifisch

für das Problem sind und ebenfalls welche, die generell anwendbar auf *MCTS* Probleme sind. Hierbei werden auch die vorgestellten Grundfunktionen des MCTS erweitert. Es wird ebenfalls darauf eingegangen, welche Beobachtungen und Ideen zu der finalen Optimierung geführt haben.
Danach folgen die Experimente, in welchen die Optimierungen ihrer Eingangsvariante gegenübergestellt werden. Diese werden dann interpretiert, sowie bestimmte Schlüsse gezogen.

1.6 Projekt

Das Programmierprojekt für diese Arbeit wurde in C++ verfasst, um eine möglichst schnelle Laufzeit zu sichern. Dahingehend spielt auch manuelle Speicherverwaltung eine wichtige Rolle.
Das Programm bietet eine grafische Oberfläche an für das Betrachten und Spielen der generierten Level. Die graphische Oberfläche wurde realisiert mit raylib v4.5[1].
Zudem können die hier vorgestellten Optimierungen und weitere Parameter angepasst werden. Genaueres kann man im Projekt nachlesen.

[1] raylib: https://www.raylib.com/ (17.11.2023)

2 Methoden

2.1 Monte Carlo Tree Search

2.1.1 Standard MCTS

Monte Carlo Tree Search ist ein Algorithmus, welcher einen Suchbaum aufbaut mittels dem Expandieren und Ausspielen von Knoten durch die Wahl zufälliger Aktionen.
Dieser lässt sich sowohl auf Mehrspieler-Spiele, als auch Einzelspieler-Spiele anwenden. In dieser Arbeit wird das Problem grundlegend als ein Einzelspieler-Spiel formuliert, wobei die genaue Verwendung von MCTS in dieser Arbeit recht unkonventionell ist.
Formal wird dieser Algorithmus für ein Einzelspieler-Spiel verwendet, welches formal wie folgt formalisiert werden kann[3, S.4].

- S: Die Menge aller Zustände.
- $S_T \subseteq S$: Die Menge aller Endzustände.
- A: Die Menge aller möglichen Aktionen.
- $f : S \times A \to S$: Die Übergangsfunktion.
- $R : S \to \mathbb{R}^k$: Die Bewertungsfunktion.

Ein Zustand kann zum Beispiel die Konfiguration eines Spielbretts sein. Eine Aktion kann das Bewegen einer Spielfigur sein. Klassische Werte für die Bewertungsfunktion wären $1, 0$ und -1 an für Gewinn, Unentschieden und Verlust [3, S.4].

Algorithm 1: Default MCTS [3, S.6]

```
function mcts_search(s_0):
    make root node v_0 with state s_0          // s_0 ist der initiale Zustand
    while break condition not met do
        v_l ← tree_policy(v_0)                 // Bester Kindknoten
        reward ← default_policy(state_of(v_l)) // Simulation
        propagate_reward(v_l, reward)
    end
    return action_to(best_child(v_0))
```

Abbildung 2.1: MCTS Suche

In der Tree-Policy wird ein noch nicht vollständig expandierter Knoten ausgewählt und expandiert mittels dem Ausführen einer Aktion. Der neue Knoten wird zurückgegeben und an Default-Policy weitergegeben.

Sollte der Knoten kein Endzustand sein, so wird eine Simulation, auch *Rollout* genannt, auf den Knoten ausgeführt. Diese Simulation wird normalerweise ausgeführt, bis ein Endzustand erreicht wird.
Dieser Endzustand wird mit der Bewertungsfunktion bewertet und der Wert wird zurückgegeben. Am Ende wird in *Propagate-Reward* die Bewertung über die Elternknoten bis zum Wurzelknoten hoch propagiert. Dabei werden normalerweise auch andere Daten verändert, wie zum Beispiel die Simulationsanzahl.
Die Abbruchbedingung für die Schleife ist generell ein Implementationsdetail und kann zum Beispiel an eine Anzahl von Durchläufen beschränkt sein oder auch bestimmten Laufzeitrestriktionen erliegen.

2.1.2 UCT

Der UCT beziehungsweise *Upper Confidence Bounds for Trees* ist eine sehr populäre Form des MCTS, für welchen es selber auch noch weitere Erweiterungen gibt. Das Ziel vom UCT ist es, eine Balance zwischen *Exploitation* und *Exploration* zu finden [3].
Exploration ist wichtig, da sich ansonsten ein MCTS zu stark auf Knoten konzentrieren kann, welche vielleicht anfangs gut bewertet wurden, jedoch man später keine guten Verbesserungen mehr bekommt. Auch kann ein Knoten anfangs einfach durch Zufall hinsichtlich der Aktionsauswahl gut bewertet worden sein. Gerade bei der Levelgenerierung kann es sein, dass ein Eingangslevel aufgrund seiner Struktur nicht das bilden komplizierterer Level erlaubt.
Exploitation ist wichtig, um mittels konsistent gut bewerteter Level bessere Knoten zu finden. Vor allem unter dem Aspekt, dass umso tiefer der Baum wird, umso länger kann es dauern, bis ein guter Knoten weiter expandiert wird.
Der Kern des UCT ist die UCB1 Selektionsmethode für die *Tree-Policy*. Dort wird bis zu einem Blattknoten iteriert und das Kindknoten welches bei jedem Schritt gewählt wird ist jenes, welches die folgende Funktion maximiert [3, S. 7]:

$$\overline{X_j} + 2C\sqrt{\frac{2ln(n)}{n_j}}$$

$\overline{X_j}$ ist der Erwartungswert vom Kindknoten und entspricht $\frac{\text{Summe aller Bewertungen}}{n_j}$. n entspricht wie oft der Knoten besucht wurde und n_j das selbige für den Kindknoten. Wurde ein Knoten noch nicht ausgewählt, so wird ∞ zurückgegeben.

C ist eine Konstante, wobei $C = \frac{1}{\sqrt{2}}$ generell empfohlen wird. Die Autoren erwähnen, dass dieser Wert genommen werden soll, wenn der Wertebereich in $[0, 1]$ liegt [3, S. 8]. Bei der Levelbewertung in dieser Arbeit können Werte weit darüber hinausgehen. In solch einem Fall empfehlen die Autoren verschiedene Werte für C zu betrachten [3, S. 8], weshalb es in den Experimenten betrachtet wird.
Der linke Term ist hier der Exploitation-Term, denn er bestimmt einen direkten Wert und der rechte Term ist der Exploration-Term [3, S.5]. Dank dem rechten Term kann ein Knoten, welcher Anfangs einen schlechteren Wert zurückgegeben hat, beim öfteren besuchen des Elternknoten ab einem bestimmten Punkt erneut ausgewählt werden.
Folgend wird der MCTS vervollständigt.

Algorithm 2: UCT Functions [3, S.9]

```
function tree_policy(v):
    while not is_terminal(v) do
        if can_expand(v) then
            return expand(v)
        else
            v ← best_child(v)                    // Kindknoten mit maximalem UCB1
        end
    end
    return action_to(best_child(v_0))
function default_policy(s):
    while not is_terminal(s) do
        a ← random action for s
        s ← f(s, a)
    end
    return reward_for(s)
function expand(v):
    a ← untried action for s(v)
    v' ← new_node()
    s(v') ← f(s(v), a)                           // Führe die Aktion aus
    v.children.add(v')
    return v'
function propagate_reward(v, reward):
    while not is_null(v) do
        v.rollout_count ← v.rollout_count + 1
        v.reward_sum ← v.reward_sum + reward
        v ← v.parent
    end
```

Abbildung 2.2: UCT Funktionen

2.2 MCTS Zur Levelgenerierung

2.2.1 Zustände

Alle Zustände entsprechen einem Sokoban-Level. Die Startposition vom Spieler wird anfangs für alle Level festgesetzt und der Wurzelknoten ist ein Level, wo alle Felder Blöcke sind außer die Startposition, dieses ist ein leeres Feld [7].
Nun zu ein paar technischen Details

Spieldarstellung

Beim MCTS werden sehr viele Knoten erstellt und jeder einzelne repräsentiert ein Sokoban-Spiel, deshalb ist es wichtig das Spielbrett möglichst kompakt zu halten.

In Sokoban kommt die Komplexität hinzu, dass ein Feld mehrere Objekte in sich haben kann zum Beispiel eine Box auf einer Zielposition. Jedes einzelne Feld wird daher von einem Byte repräsentiert. Bei diesem Byte repräsentieren die unteren 4 Bits die untere Schicht, also Leer und Zielposition und die oberen 4 Bits repräsentieren die obere Schicht bzw. die Kollisionsschicht und damit den Spieler, Boxen und die Blöcke.
Damit kann jedes Feld als ein Element aus $\{Leer, Spieler, Box, Block\} \times \{Leer, Zielposition\}$ verstanden werden.
Das Brett wird in 2D-Koordinaten dargestellt, jedoch aus Effizienzgründen als ein eindimensionales Array gespeichert. Ein jedes gültiges Feld (x, y) befindet sich in $(x, y) \in [0, n[\times[0, m[\subseteq \mathbb{N} \times \mathbb{N}$.
Für ein ungültiges Feld (x, y) gilt $(x, y) \notin [0, n[\times[0, m[$. Ein invalides Feld ist zum Beispiel $(-1, -1)$ und lässt sich als ein unsichtbares Feld erklären, welches das eigentliche Level umrandet.
Zwei Felder sind Nachbarn, falls die Manhattan-Distanz zwischen ihnen bei 1 liegt.
Das Feld $(0, 0)$ startet links oben. Standardmäßig hierfür geht nach rechts die positive x-Achse und nach unten geht die positive y-Achse.

Knoten

Alle Knoten besitzen grundlegend im UCT eine Referenz zu ihrem Elternknoten, ein Array mit Einträgen zu ihren Kindknoten, ihre aufsummierte Bewertung und die Simulationsanzahl für die Berechnung von dem Erwartungswert, welcher für den UCB benötigt wird.
Jeder Knoten besitzt ein eindimensionales Array, welches das Sokoban-Spiel darstellt. Zudem wird immer die momentane Position des Spielers abgespeichert.
Weitere Variablen werden noch vorgestellt.

2.2.2 Aktionen

Nun wird der grundlegende Algorithmus aus [7] vorgestellt. Die Standard-Aktionen in [7, S. 60] sind

- **Delete-Obstacle**: Löscht einen Block, welcher ein leeres Feld oder Box als Nachbarn hat.
- **Place-Box**: Platziert eine Box auf ein leeres Feld.
- **Freeze-Level**: Das momentane Brett wird als Startkonfiguration abgespeichert.
- **Move-Agent**: Bewegt den Spieler nach den Regeln von Sokoban.
- **Evaluate-Level**: Finale Aktion, welche in einen Endzustand übergeht.

Die Aktionen werden hierbei in zwei Mengen unterteilt. Zunächst dürfen bei einem jeden Zustand nur die ersten drei Aktionen angewandt werden. Nachdem die *Freeze-Level* Aktion verwendet wird, dürfen bei allen weiteren Zuständen nur die letzten beiden Aktionen angewandt werden. Die letzte Aktion führt immer in einen Endzustand und würde somit eine Simulation beenden.
Die *Delete-Obstacle* und *Place-Box* Aktionen bauen das grundlegende Level auf. Bei *Place-Box* darf zudem keine Box auf die Startposition gesetzt werden.

Für die Aktion *Freeze-Level* werden die Startpositionen als Integer in einem Array abgespeichert.
Nach dem Einfrieren des Levels findet über *Move-Agent* das Bewegen der Boxen statt. Die Boxen werden bewegt, indem der Spieler bewegt wird nach den Regeln von Sokoban.

Eine sehr starke Eigenschaft dieser Anwendungsweise ist, dass das Level somit stets lösbar sein muss und später nicht über einen teuren Algorithmus überprüft werden muss[7].

Bei *Evaluate-Level* wird der Spieler dann an den Startzustand gesetzt und alle momentanen Boxpositionen werden mit Zielpositionen ersetzt und die Boxen werden an ihre Startposition gesetzt.
Zudem findet vor der finalen Bewertung noch *Post-Processing* statt [7]. Dort werden alle Boxen, welche nicht bewegt wurden, mit einem Block ersetzt und alle Boxen, die nur einmal bewegt wurden, werden mit einem leeren Feld ersetzt [7]. Eine nicht bewegte Box funktioniert äquivalent zu einem Block, weshalb diese Umwandlung valide ist.
Für diesen Prozess wird in einem Array der *Move-Counter* beziehungsweise die Bewegungsanzahl der Boxen in einem Array abgespeichert.

Hinsichtlich dem Abspeichern von Leveln wird am Ende jeder *Default-Policy* die Bewertung vom generierten Endknoten überprüft. Sollte diese Bewertung ein neuer Bestwert sein, so wird dieses Level abgespeichert.
Es gilt anzumerken, dass die Knoten nach der *Default-Policy* verworfen werden und daher die abgespeicherten Level nicht im Baum selber präsent sind.

Zwei Methoden, welche für die Implementierung speziell sind ist *Delete-Obstacle* und *Move-Agent*. Auf *Move-Agent* wird später eingegangen. Hinsichtlich *Delete-Obstacle* gibt es zwei Methoden die Bedingung für das gesamte Spielfeld zu überprüfen. Hierbei ist $\neg Block$ ein Element, welches kein Block ist:

- Überprüfe für jeden Block, ob es einen $\neg Block$ als Nachbarn hat.
- Überprüfe für jeden $\neg Block$, ob es einen Block als Nachbarn hat und markiere diesen Block. Hierbei muss man auf Überschneidungen achten.

Die zweite Methode ist etwas schneller für sehr große Level, da diese mit $n \cdot m - 1$ vielen Blöcken anfangen. Im gesamten konnte die erste Methode jedoch bessere Ergebnisse erzielen.

2.2.3 Zustandsbewertung

Die in [7, S. 61-62] stark korrelierenden und verwendeten Metriken sind:

- n : **Box-Count**: Die Anzahl an Boxen auf dem Feld.
- P_b : **3x3-Block-Count**: Die Anzahl an 3×3 Feldern, bei denen mindestens ein Feld ein Block ist und mindestens ein Feld Leer ist.
- P_c : **Congestion-V2**: Hier wird für alle Boxen und deren Zielpositionen das Rechteck betrachten, in welchem sie sich befinden, wobei die Boxposition und Zielposition jeweils eine Ecke des Rechtecks sind. Nun sei b_i die Anzahl an Boxen im Rechteck, g_i die Anzahl an Zielpositionen im Rechteck, o_i die Anzahl an Blöcken im Rechteck und A_i der Flächeninhalt des Rechtecks. Zusammen mit Gewichten α, β, γ folgt

$$\sum_{i=1}^{n} \frac{\alpha b_i + \beta g_i}{\gamma(A_i - o_i)}$$

Congestion-V2 bewertet hierbei Überführungen sehr gut, bei denen sich möglichst viele andere Objekte zwischen der Box und dem Ziel befinden, in Abhängigkeit vom Flächeninhalt. Die Anzahl der Blöcke wirkt sich dabei sehr stark auf die Bewertung aus, denn wenn sich zwischen Box und Zielposition sich Blöcke befinden, dann führt dies grundlegend zu interessanten Situation.

In [7] wurde nicht genau erläutert, ob die Box und Zielposition, welche die beiden Eckpunkte des Rechtecks bilden, ebenfalls mitgewertet werden sollen oder nicht. Diese mitzubewerten würde gegen die grundlegende Idee dieser Metrik gehen und Boxen, welche tatsächlich Hindernisse sind, gleich bewerten mit der Startbox. Zudem würden Situationen wie in Abbildung 2.3 zu einer Bewertung von $\frac{1+1}{1} = 2$ führen, also einer eigentlich perfekten Bewertung für eine offensichtlich ungewollte Konfiguration.

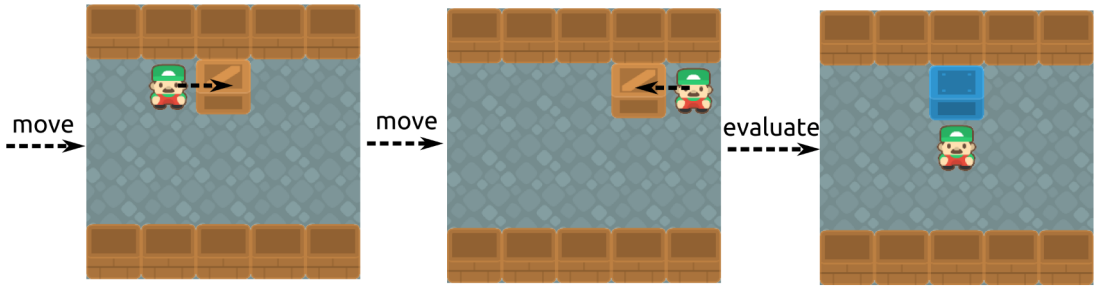

Abbildung 2.3: Zwei Bewegungen, bei welchen die Box wieder in der Eingangsposition endet

Texturen aus [9]

Auch soll *Box-Count* eine große Anzahl an Boxen belohnen und nicht Congestion-V2 mit leeren Wegen zwischen Box und Zielposition.
Demnach werden die Eckpunkte nicht mitgezählt und man betrachte Abbildung 2.4 für eine Visualisierung. In dem Rechteck eins ergibt sich eine Bewertung von $\frac{1+0}{6-1} = \frac{1}{5}$ und im Rechteck zwei die Bewertung $\frac{0+0}{4} = 0$. Das zweite Rechteck besitzt keine Hindernisse, weshalb dies nicht belohnt werden soll.

Ebenfalls gilt es zu erwähnen, auch wenn eine Kombination von Box und Zielposition zu null Punkten führt, so kann diese Kombination dennoch wichtig für die *Congestion-V2* Bewertung eines anderen Paars sein. Also muss man den Effekt dieser Metrik im gesamten betrachten.
Congestion-V2 ist eine verbesserte Form der ersten Bewertungsmethode *Congestion-V1* von [8]:

$$\sum_{i=1}^{n} (\alpha b_i + \beta g_i + \gamma o_i)$$

Hier wurden Blöcke gleich behandelt zu Boxen und Zielpositionen. Diese zweite Methode hat eine kleinere Korrelation mit dem empfundenen Schwierigkeitsgrad [7]. In dem Programmierprojekt wurden beide Methoden implementiert, jedoch wurde final *Congestion-V2* verwendet.

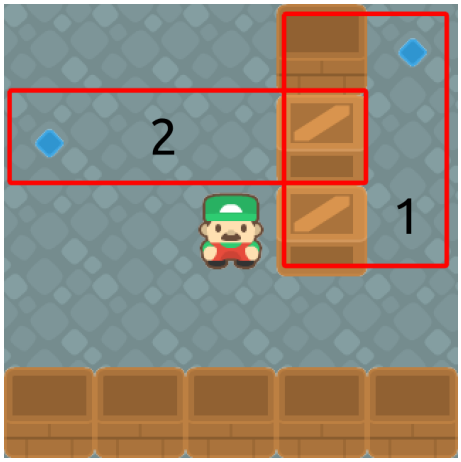

Abbildung 2.4: Congestion-Metrik

Texturen aus [9]

Der *3x3-Block-Count* soll dafür sorgen, dass die Topologie der Level selber interessant bleibt.
In [8] hat man hierfür die Terrain-Metrik verwendet, bei welcher alle Blöcke gezählt werden, die ein Nachbar eines leeren Feldes sind.
In [7] wurde nicht erläutert, ob man 3x3 Felder betrachten soll, welche invalide Felder beinhalten. In dieser Arbeit hier wird es nicht getan.

Die Bewertungen werden dann zusammengeführt mit Gewichten w_b, w_c, w_n und einem Normalisierungsfaktor k, welcher es möglichst in den Wertebereich von $[0, 1]$ bringen soll[7, S. 62]:

$$\frac{w_b P_b + w_c P_c + w_n n}{k}$$

Anzumerken gilt, dass man in [7] die Bewertung über $1,2$ hinweg hat laufen lassen, obwohl im MCTS normalerweise $[0, 1]$ angenommen wird, da dies auch in Beweisen angenommen wird [3, S. 7].
Außerdem wurde in [7] nicht erläutert, ob und wie die Bewertungsmethode für verschiedene Feldgrößen angepasst werden soll. Deswegen werden alle Bewertungen hier mit $\frac{n \cdot m}{25}$ skaliert. Dies dient dazu die Level unabhängig der Größe möglichst in den von [7] vorhergesehenen Zahlenbereich von $[0, 1.2]$ zu bringen. Die 25 kommt von der Standardfeldgröße von 5×5. Es sollte angemerkt werden, dass die Level dennoch weit über 1,2 hinausgehen können.
Hinsichtlich der Gesamtbewertung werden zudem Level mit 0 Boxen auch stets mit 0 bewertet.

2.2.4 Beispiel

Im folgenden Beispiel ist in Abbildung 2.5 der direkte Pfad in einem Baum bis zu einer Freeze-Aktion angeben und in Abbildung 2.6 die Move-Aktionen und die Evaluierung. Dieses Beispiel zeigt zudem die Mindestanzahl an Schritten die notwendig sind, damit in einem 5×5 Spielfeld ein gültiges Sokoban-Level erzeugt wird. Die Box muss zweimal bewegt werden, da sie sonst mit einem leeren Feld ersetzt wird. Insgesamt ergeben sich 12 Aktionen und damit auch eine Baumtiefe von 12.
Auch wird hiermit die grundlegende Struktur etwas ersichtlicher. Der Algorithmus baut in der ersten Phase die Levelstruktur auf und in der zweiten Phase wird das Level mit dynamischen Objekten erweitert.

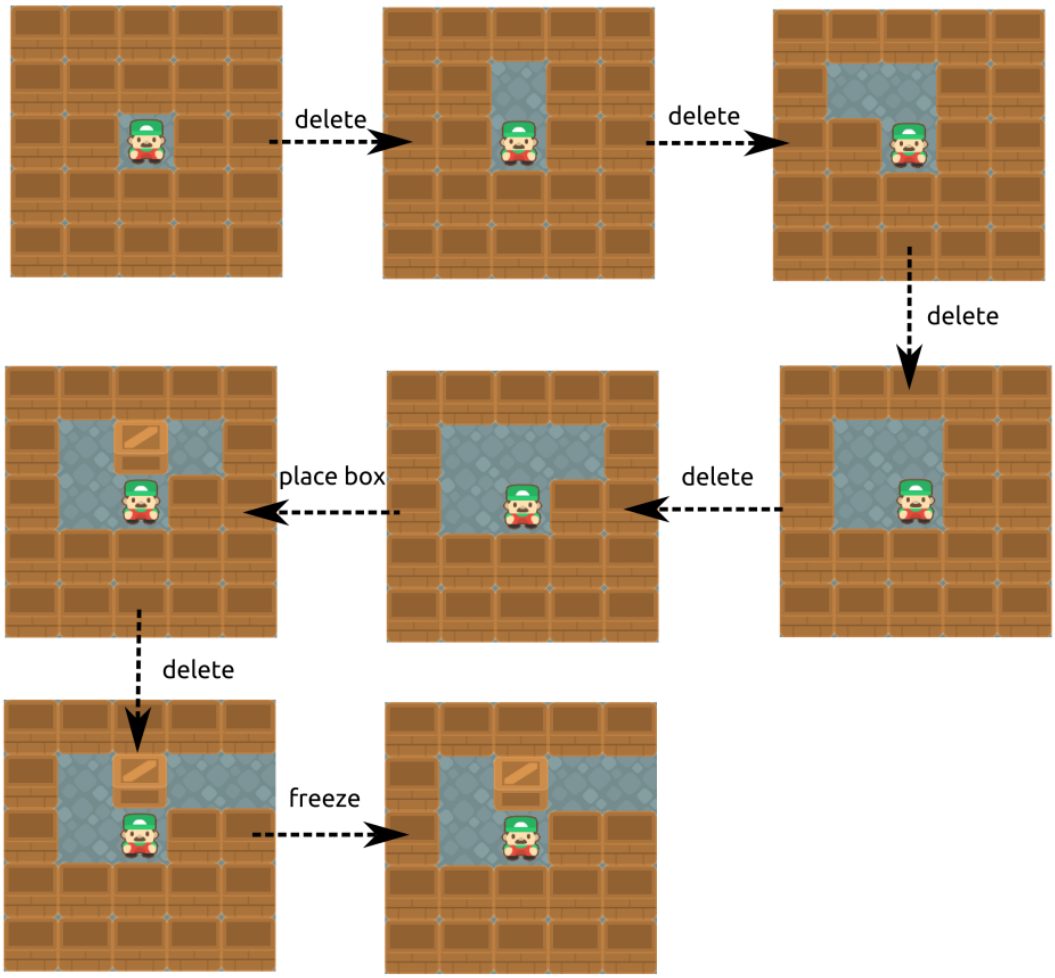

Abbildung 2.5: Beispiel: Ein Pfad bis zur Aktion Freeze

Texturen aus [9]

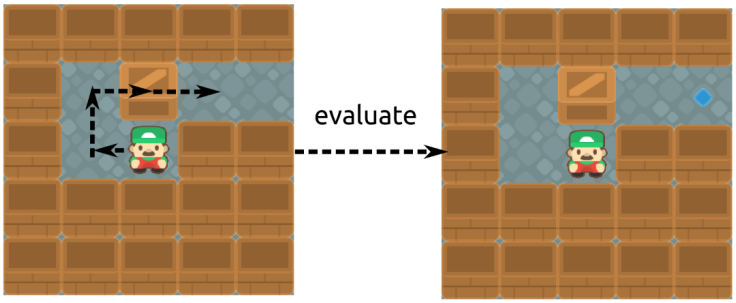

Abbildung 2.6: Moves und Evaluate

Texturen aus [9]

2.2.5 Zufallszahlengenerierung

Für einen MCTS muss man sehr oft zufällige Zahlen generieren und diese auch in bestimmten Reichweiten für die Aktionsauswahl. Hierfür benötigt man einen Algorithmus, welcher möglichst schnell ist und bei denen die Zahlen auch gleichverteilt sind.

Die naive Implementierung mit *rand* und Modulo, um eine Zufallszahl in einer bestimmten Reichweite zu bekommen ist ungenügend, aufgrund der mangelnden Qualität.
Neuere C++ Versionen bieten einen Zufallszahlengenerator, welcher auf dem *Mersenne Twister* Algorithmus beruht und damit hochqualitative Zufallszahlen anbietet [4].
Diese Variante ist langsamer, dennoch hat die Ausführung des Algorithmus mit diesen zufälligen Zahlen generell zu besseren Ergebnissen geführt.

2.3 Optimierungen

Ziel der Optimierungen im Monte-Carlo-Tree-Search ist es, die Anzahl an möglichen Aktionen zu verringern zugunstem dem schnelleren finden von guten Zuständen. Dies kann dazu führen, dass einzelne Schritte im Algorithmus länger brauchen, jedoch alles im gesamten effizienter wird. Gerade im *MCTS* mit *UCB1* kann es sehr viele ineffiziente Aktionen geben.
Die Optimierungen hinsichtlich der Aktionen finden generell statt mittels domänenspezifischen Wissen.
Ebenfalls spielt es eine Rolle, den Speicherverbrauch eines jeden einzelnen Knoten möglichst klein zu halten, da ein MCTS dazu tendiert sehr viel Speicherplatz zu verbrauchen.

2.3.1 Tree-Policy

Eines der größten Probleme anfangs war es, die *Tree-Policy* zu gestalten.
In der *Default-Policy* im MCTS ist es vorgesehen, immer stets die nächste Aktion zufällig zu wählen [3].
In der *Tree-Policy* ist es jedoch ein Implementationsdetail, wie die nächste Aktion gewählt wird.
Auch gibt es verschiedene Möglichkeiten bei der Expandierunsanzahl und zwar kann man nur einen Knoten nach dem anderen erstellen, oder man erstellt sofort alle. Der zweite Fall ist recht Speicherintensiv und vor allem bei einer höheren Explorationsrate eine schlechte Idee. Bei dem ersten Fall hat man jedoch das Problem, dass es je nach Spiel nicht trivial ist herauszufinden, welche weiteren Aktionen noch offenstehen. Zudem möchte man auch recht effizient alle übrigen Aktionen kennen, um eine zufällige Expansion zu ermöglichen.

Am Anfang im Projekt wurden alle Kindknoten erstellt und eines simuliert. Die Speichernutzung hiervon war viel zu hoch. Vor allem da es am Anfang im Projekt noch keine richtigen Optimierungen gab, wodurch die Iterationszahl sehr hoch war und damit auch die Knotenanzahl. Damit ergab sich, dass die Methode nur einen einzelnen Kindknoten erstellen darf.
Daraufhin ergab sich dann das Problem zu entscheiden, welche Aktionen noch offenstehen. Naiv müsse man durch alle Kindknoten iterieren und überprüfen, ob die Aktion im Feld (x, y) bereits ausgeführt worden ist. Der Baum besitzt ab einer bestimmten Tiefe jedoch sehr viele Aktionsmöglichkeiten, wodurch dies zu kostspielig wäre.

Daher wird ein Kompromiss zwischen Effizienz und Speicherplatz gewählt und damit wird die *Tree-Policy* wie folgt erweitert.

Wurde ein Knoten noch nie zuvor expandiert und wird dieser nun zur Expansion gewählt, so werden in einem Array alle Indices abgespeichert, bei welchen ein *Delete-Obstacle* ausgeführt werden kann. Selbiges für *Place-Box*.
Damit ist die Anzahl aller möglichen Aktionen bekannt und es kann jeder Aktion eine Zahl zugeordnet werden. Dahingehend kann so recht einfach zufällig expandiert werden und der Expansionsschritt selber kann schnell durchgeführt werden. Sollten im Array alle möglichen Aktionen erschöpft sein, so wird der Speicher für das Array wieder freigegeben.
Das hilfreiche an dieser Methode ist ebenfalls, dass die Indices als Integer abgespeichert werden und diese auch nur benötigt werden, bis man an einem *Freeze-Level* angelangt ist, da danach keine weiteren *Delete-Obstacle* und *Place-Box* Aktionen mehr stattfinden. Damit können die beiden Arrays für das Speichern der Box-Startpositionen und *Move-Counter* wiederverwendet werden. Mit dieser Methode ist es auch sehr einfach und effizient, Bedingungen an Aktionen zu stellen, die noch folgen. Diese Erweiterung wird als Bloom bezeichnet.

In Algorithmus 3 sieht man die grundlegende Struktur dieser Implementierung. Bloom wird bevor der Erstexpansion eines Knoten aufgerufen und *new_action_name* wird in der Expansionsfunktion aufgerufen und der Knoten wird zurückgegeben. Zudem sieht man ebenfalls die erweiterte *Tree-Policy*. Die spezifischen Implementierungen der Aktionen befinden sich im Projekt.

Hinsichtlich der Expansion für die *Tree-Policy* wurde zunächst der nächstmögliche Aktionstyp gewählt in der Reihenfolge *Delete*, *Place*, *Freeze*. Die Wahl der jeweiligen Aktionen vom Aktionstyp folgte dann zufällig. Der Grund hierfür war, dass die rein zufällige Wahl etwas kostenintensiver in der Implementierung ausfällt und dahingehend als ein Vergleich dienen sollte. Später mit allen Optimierungen konnte kein signifikanter Unterschied zwischen beiden Methoden festgestellt werden, weshalb die erstere final gewählt wurde.

Algorithm 3: Generating actions and expanding

```
function action_*(node):
    node.possible_actions ← generate_possible_actions(node)
function bloom(node):
    // Falls freeze verwendet wurde
    if node.flags contains MCTS_SECOND_ACTION then
        action_move_agent(node)
    else
        action_delete_obstacle(node)
        action_place_box(node)
        action_freeze(node)
    end
    node.flags ← node.flags ∪ {MCTS_BLOOMED}
// Die Funktion wird implementiert für alle 5 Aktionen
function new_action_*(node):
    action ← random_element(node.possible_actions)
    remove action from node.possible_actions
    child ← apply_action(node, action)
    return child
function new_tree_policy(node):
    while not is_terminal(node) do
        if can_expand(node) then
            if MCTS_BLOOMED ∉ node.flags then
                bloom(node)
            end
            return expand_next(node)                    // calls new_action_*
        else
            v ← best_child(node)
        end
    end
```

Abbildung 2.7: Algorithmusstruktur für die Aktionen und Bloom

2.3.2 Move-Agent

Das größte Problem an der naiven Standardimplementierung, ist die *Move-Agent* Aktion und zwar kann diese sehr schnell die Baumtiefe erhöhen, wie man bereits in Abbildung 2.6 sehen konnte. Man betrachte Abbildung 2.8. Um die erste Box zu erreichen erfordert es mindestens 3 Aktionen und um die oberste Box zu erreichen, erfordert es 11 Aktionen und dabei gilt es auch zu beachten, dass die Bewegung zufällig ist und sich der Spieler dadurch zum Beispiel im Kreis bewegen kann.

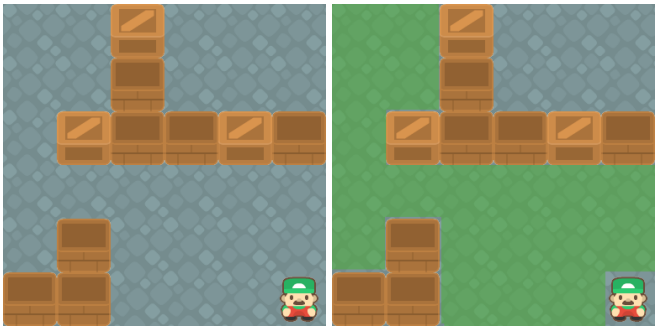

Abbildung 2.8: Markierung des erreichbaren Bereichs

Texturen aus [9]

Die erste Implementierungsidee war es, sich immer die letzte Bewegung abzuspeichern, um den Spieler daran zu hindern die letzte Bewegung rückgängig zu machen. Dies hat insgesamt relativ wenig gebracht und das eigentliche Problem nicht gelöst. Denn das Bewegen des Spielers selber birgt eigentlich keine Bedeutung für die Levelgenerierung, da das Level selber nicht verändert wird. Das eigentliche Wichtige ist das Bewegen der Boxen.
Deshalb wird die *Move-Agent* Aktion mit dem Algorithmus 4 erweitert. Dieser Algorithmus bewegt sich rekursiv von der Startposition aus in alle Richtungen und die Funktion wird für jedes Feld maximal einmal aufgerufen.

Sollte es von einem Feld aus möglich sein sich so zu bewegen, dass eine Box mitbewegt wird und auch das Feld ändert, dann wird sich der Eingangspunkt und die Bewegungsrichtung abgespeichert. Zudem wird beides jeweils als ein möglichst kleines Integer abgespeichert.
In jedem Schritt überprüft der Algorithmus, ob sich der Spieler auch tatsächlich bewegen kann und ruft erst dann die Funktion für die nächste Position auf. Dies garantiert, dass alle Schiebungen von Boxen auch tatsächlich erreichbar sind.

In Abbildung 2.8 ist der Bereich markiert, in welchem die Funktion jeweils aufgerufen wird.
In Abbildung 2.9 lassen sich die gespeicherten Bewegungsinformationen sehen.
Damit kann dann bei der Expansion einer der zufälligen Pushes ausgewählt werden.
Diese Methode hat zu einer drastischen Reduzierung der Laufzeit geführt. Ein zu beachtender Unterschied zur naiven Methode ist, dass das Bewegen einer jeden Box unabhängig von der Distanz zu der Box geschieht und damit zu einer Inflation der Bewegungsanzahl führen kann. Dies passiert jedoch nur, wenn die Boxpositionen voneinander abhängen und damit eine interessante Komplexität aufzeigt.

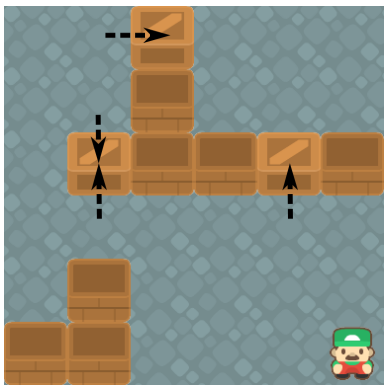

Abbildung 2.9: Markierung der abgespeicherten Bewegungen

Texturen aus [9]

Algorithm 4: Generating all possible moves

```
function get_all_possible_moves(pusher_position, grid):
    visited ← Array<bool> of size grid
    set all entries in visited to false
    visited[pusher_position] ← true
    moves ← Array<Move_Info>
    get_all_possible_moves_recursive(pusher, grid, visited, moves)
    return moves
function get_all_possible_moves_recursive(tile, grid, visited, moves):
    for direction in {Up, Right, Down, Left} do
        to ← tile + direction
        if (not grid.is_inside(to)) or visited[to] then
            continue
        end
        pawn ← top_layer(grid.get(to))                    // Box, Player oder Block
        if is_empty(pawn) then
            visited[to] ← true
            get_all_possible_moves_recursive(to, grid, visited, moves)
        else
            if is_box(pawn) and grid.can_push(tile, direction) then
                moves.add({tile, direction})
            end
        end
    end
```

Abbildung 2.10: Generierung aller Bewegungsmöglichkeiten

2.3.3 Tiefenlimits

Eine mögliche Abbruchbedingung in der *Default-Policy* kann die Tiefe des Baums sein, wie sie in *Minimax* auch verwendet wird [3].
In dieser Arbeit ist es nicht notwendig, da nach einem *Freeze* bei jeder zufälligen Expansion ein Endknoten recht schnell erreicht wird und das auch konsistent, wie in den Experimenten zu sehen ist. Ebenfalls hängt die tatsächliche Levelgenerierung von den Ergebnissen der *Default-Policy* ab.
Umgekehrt konnte man jedoch bei Abbildung 2.5 und 2.6 sehen, dass es sehr viele Aktionen erfordert, bis ein gültiges Level erstellt wird. Beziehungsweise ein Level, welches nicht mit null bewertet wird. Da die Aktionen zufällig gewählt werden, kann es so ohne weiteres vorkommen, dass man am Anfang sehr lange nur sehr niedrig bewertete Level findet.
Dies hat am Anfang vom Projekt zu dem Problem geführt, dass die Berechnung von *UCB1* nicht mehr richtig funktioniert hat, aufgrund von zu niedrigen Gleitkommazahlen.
Gerade wenn man bedenkt, dass der Wurzelknoten anfangs fünf Kinder besitzt, wovon vier *Delete-Obstacle* sind und eins *Freeze-Level* ist. *UCB1* führt dazu, dass *Freeze-Level* recht oft aufgerufen werden kann, obwohl immer mit null Punkten zu rechnen ist. Solche *Freeze-Level* mit sicheren null Punkte Bewertungen wird es in den ersten paar Ebenen im Baum sehr oft geben.
Auch erfordert es mehrere Bewegungsaktionen bis ein valides Level produziert wird.
Hier kann insgesamt ein Tiefenlimit aushelfen. Bei der Implementierung des Tiefelimits muss man jedoch beachten, dass manchmal Bewegungen nicht möglich sind, wodurch *Move-Agent* nicht anwendbar ist. Dahingehend kann ein hartes Tiefenlimit nicht auf *Evaluate-Level* angewandt werden.
Der Baum fängt mit einem voll besetzten Level an, wodurch das Erreichen eines bestimmten Tiefenlimit grundlegend garantiert ist für *Freeze-Level*.
Damit wird bei *Freeze-Level* eine weitere Bedingung hinzugefügt und zwar muss eine bestimmte Baumtiefe erreicht werden, bis diese Aktion ausgeführt werden darf.
Wie man in Abbildung 2.5 sehen kann, sollte solch ein Tiefenlimit bei mindestens 7 sein. Dieses minimale Tiefenlimit sollte generell für Level aller Größen nutzbar sein.
Die Idee für *Evaluate-Level* war es ein weiches Tiefenlimit zu wählen. Dieses würde dafür sorgen, dass stets *Move-Agent* ausgeführt wird, wenn es ausführbar ist unterhalb dieses Tiefenlimits. Dieses weiche Tiefenlimit konnte jedoch keine Verbesserung darstellen.

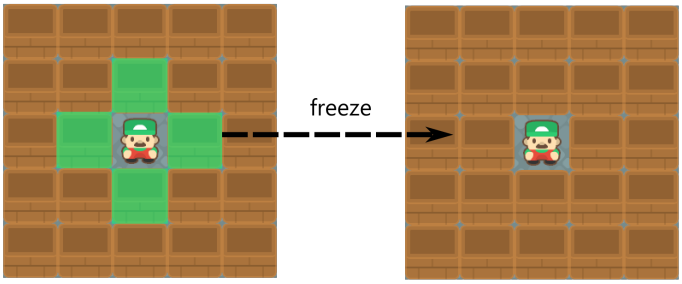

Abbildung 2.11: Eine ungewollte Wahl von *Freeze-Level*

Texturen aus [9]

2.3.4 Boxlimits

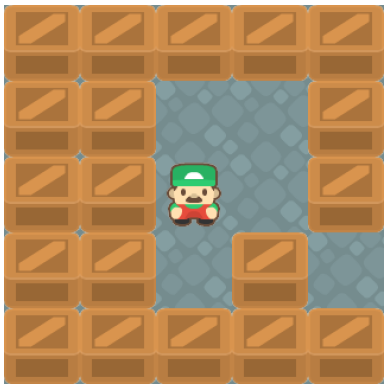

Abbildung 2.12: Ein Level, bei dem so viele Boxen generiert wurden, dass das Bewegen von Boxen eingeschränkt ist.

Texturen aus [9]

Die Autoren aus [7, S.63] erwähnen die Verwendung eines Boxlimits für die Levelgenerierung. Primär dient es für eine Variation hinsichtlich der Levelgenerierung [7, S.63].
Ohne solch ein Boxlimit kamen anfangs Situation wie in Abbildung 2.12 sehr oft zustande.
Solch ein Fall führt natürlich dazu, dass die meisten Boxen am Ende wieder mit einem Block ersetzt werden, also aktiv die Topologie des Levels bildet. Dies sollte jedoch bevor *Freeze-Level* geschehen und zudem erhöht es massiv die Anzahl an möglichen Aktionen im Baum.
Auch kann es passieren, dass ein Knoten *Freeze-Level* ohne eine Box ausführt.
Dahingehend werden zwei Vorbedingungen eingeführt.
Für *Freeze-Level* muss mindestens eine Box vorhanden sein.
Für *Place-Box* dürfen maximal h Boxen vorhanden sein. h muss hierbei je nach Levelgröße dynamisch gewählt werden.
Für h gibt es hierbei zwei sinnvolle Kandidaten.
Da eine jede Box zweimal bewegt werden muss damit sie gültig ist, ergibt sich $h = \lceil \frac{n \cdot m}{3} \rceil$ als eine logische obere Grenze.
Ebenfalls kann das Platzieren einer einzelnen Box, welche in ein Block umgewandelt wird einen sehr starken Einfluss auf die Metriken und Schwierigkeitsgrad haben, wodurch $h = \lceil \frac{n \cdot m}{2} \rceil$ auch eine sinnvolle Wahl ist. Denn damit können leichte Anpassungen bzw. Korrekturen durchgeführt werden. Die Bedeutung dieser Eigenschaft wird noch einmal in der nächsten Optimierung deutlich.

2.3.5 Konfigurationen

Die Idee für unmögliche Konfigurationen stammt aus dem Beweis aus [5]. In Sokoban gibt es unlösbare Konfigurationen hinsichtlich der Boxen. Und zwar können Boxen so positioniert sein, dass diese nicht auf Zielpositionen bewegt werden können. Sokoban liegt in *PSPACE* und deswegen wird dies nicht direkt betrachtet. Stattdessen werden alle Boxen betrachtet, welche nicht bewegt werden können.
Ein Beispiel hierfür kann man in Abbildung 2.13 sehen. Keiner der beiden Boxen kann bewegt werden. Dies führt dazu, dass bei *Evaluate-Level* beide Boxen mit einem Block ersetzt werden und das

Level mit 0 bewertet wird. Würde eine der beiden Boxen entfernt werden, so könne man die andere Box bewegen und das Level könne besser bewertet werden.
Das Entfernen der Boxen findet in der *Freeze-Aktion* statt.
Die erste offensichtliche Implementierungsidee war es für alle Boxen zu überprüfen, ob sie bewegt werden können. Diese Implementierung hat signifikant schlechter performt.
Der folgende Ansatz konnte eine Verbesserung darstellen und funktioniert ähnlich zum *3x3-Block-Count*. Es sollte jedoch angemerkt werden, dass dieser Ansatz nicht alle Boxen erkennt, die nicht bewegbar sind.
Folgend wird ein Feld als ein Hindernis bezeichnet, falls es invalide ist oder ein Block ist.
Viele unmöglichen Konfiguration lassen sich wie folgt erkennen:

- Eine 2×2 Fläche in welcher es nur eine Box gibt und die anderen Felder Hindernisse sind. Zum Beispiel eine Box auf Feld $(0, 0)$.
- Eine 2×2 Fläche bei welchem eine Spalte aus Boxen besteht und die andere aus Hindernissen wie in Abbildung 2.13. Anstatt Spalten können es auch Reihen sein.
- Eine 2×2 Fläche befüllt mit Boxen.

Die Ausnahme hinsichtlich der Boxen besteht, wenn sich die Boxen auf einer Zielposition befinden. Das Entfernen findet jedoch statt bevor Zielpositionen existieren und deshalb muss dies hier nicht betrachtet werden.
Die aufgelisteten Bedingungen dienen dazu, das Problem in einem möglichst *Divide-and-conquer* Ansatz zu betrachten.
Ähnlich zu dem *3x3 Block Count* wird hier über das Feld iteriert und bei jedem Schritt ein $2x2$ Rechteck betrachtet. Jedoch ist der Startpunkt hierbei $(-1, -1)$. Dann werden die notwendigen Angaben gezählt.
Sollten sich im Rechteck vier Boxen oder nur eine Box und drei Hindernisse befinden, so wird die erste Box entfernt.
Sollten sich im Rechteck 2 Boxen befinden und 2 Hindernisse, so wird die Manhattan-Distanz zwischen den ersten beiden Boxen berechnet. Sollte diese Distanz bei eins liegen, so befinden sich beide Boxen in einer Reihe oder einer Spalte und damit kann einer der beiden Boxen entfernt werden.
Diese Implementierung ist zum einen schneller als die naive Implementierung und außerdem entfernt sie nicht alle Boxen. Nicht alle Boxen zu entfernen könnte der Grund für das bessere Ergebnis sein, da nicht bewegte Boxen zu Blöcken werden und diese nach *Congestion-V2* sehr gut bewertet werden.

Außerdem führt im Gegensatz zu den anderen Optimierungen diese tatsächlich dazu, dass die Aktionsanzahl wächst, da die Boxen wieder bewegt werden können.

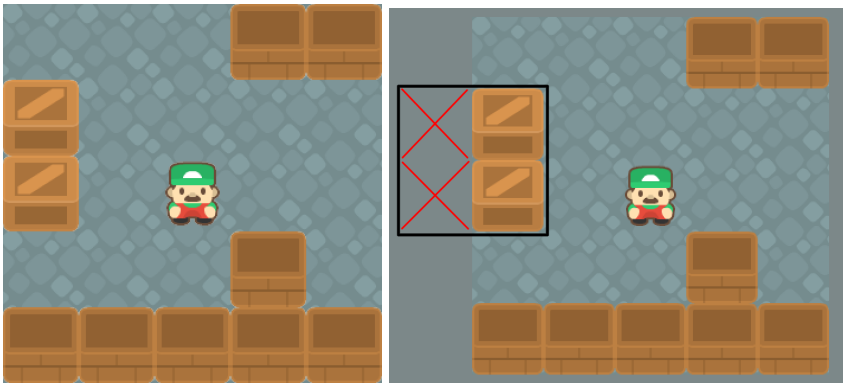

Abbildung 2.13: Die beiden Boxen sind so positioniert, dass keine der beiden bewegt werden kann.

Texturen aus [9]

2.3.6 UCB Alternativen

Die folgenden Erweiterungen sind verschiedene Ansätze und Versuche für bestimmte Situationen bessere Funktionen als UCB1 zu sein. Alle Ansätze konvergieren generell stets gegen den Erwartungswert μ, da dies optimal ist.

UCB1-TUNED

UCB1-TUNED nach [2, S. 245] maximiert

$$\overline{X_j} + C \cdot \sqrt{\frac{ln(n)}{n_j} min(\frac{1}{4}, V_j(s))}$$

$$V_j(s) = \frac{\sum_{\tau=1}^{s} X_{j,\tau}^2}{s} - \overline{X}_{j,s}^2 + \sqrt{\frac{2ln(t)}{s}}$$

Diese Funktion erlaubt es UCB1 noch genauer zu kontrollieren und in den Experimenten der Autoren liefert UCB1-TUNED stets bessere Ergebnisse, jedoch konnten sie nicht den *Regret-Bound* beweisen [2, S. 245].
Diese Methode erfordert, dass die Bewertung in einer separaten Variable stets quadriert hinzu addiert wird.

UCB-V

Bei UCB-V nach [1, Abschnitt 2.1] soll folgendes maximiert werden

$$B_{k,s,t} = \overline{X_{k,s}} + \sqrt{\frac{2V_{k,s}\mathcal{E}_{s,t}}{s}} + C\frac{3B\mathcal{E}_{s,t}}{s}$$

$\mathcal{E}_{s,t}$ soll hierbei eine monoton steigende Funktion sein. Die Standardwahl ist $\mathcal{E}_{s,t} = \xi log(t)$. $\overline{X_{k,s}}$ ist der Erwartungswert und $V_{k,s}$ ist die Varianz. Die Bewertungen sind so verteilt, dass sie sich in $[0, B]$ befinden. Das generelle Ziel von UCB-V ist es, den dritten Term dominieren zu lassen, falls die Explorationsfunktion $\mathcal{E}$ es zulässt und s nicht zu groß wird [1, Abschnitt 2.1].

Da die Bewertungen bereits quadriert mit abgespeichert werden, kann die kann die Varianz einfach wie folgt berechnet werden [14]:

$$\sigma^2 = \frac{squared\_score - \frac{score^2}{n}}{n-1}$$

Single-Player MCTS

Bei SP MCTS nach [3, S.11] wird folgendes maximiert

$$\overline{X_j} + 2C\sqrt{\frac{2ln(n)}{n_j}} + \sqrt{\sigma^2 + \frac{D}{n_j}}$$

Hier wird zur UCB1 Formel noch ein dritter Term hinzugefügt mit einer einstellbaren Konstante D. Für hohe n_j konvergiert der dritte Term gegen σ und addiert damit einfach die Standardabweichung. Der Zusatz von $\frac{D}{n_j}$ soll für wenig besuchte Knoten die Unsicherheit des Werts darstellen [3, S.11], also wird der Wert erhöht für eine wahrscheinlichere Wiederwahl.

2.3.7 Default-Policy Allokation

Bei der Default-Policy wird der zu simulierende Knoten dupliziert und expandiert bis ein Endzustand erreicht wird. Danach werden alle Knoten, welche während der Simulation gebildet werden, gelöscht. Das Problem ist, dass in diesem repetitiven Ablauf sehr viel allokiert wird und gleich wieder gelöscht wird. Dies ist sehr ineffizient, gerade weil der Speicherverbrauch in jeder *Default-Policy* eigentlich recht gleich ist beziehungsweise nie ein bestimmtes Limit überreicht. Außerdem kann es zur Speicherfragmentierung führen und damit Speicher unnötig verbrauchen.

Der im Programm verwendete Allokator ist **malloc**. Dieser ist ein generischer Allokator, also nicht unbedingt sehr performant für spezifische Situationen. Deswegen wird hier ein Arena-Allokator verwendet.
Der Arena-Allokator soll ein einfacher Ersatz für *malloc* während der Default-Policy sein und deswegen folgende drei Funktionen anbieten wie *malloc* [11]:

- **malloc(size)**: Alloziere Speicher der Größe *size*.
- **realloc(ptr, size)**: Alloziere einen neuen Speicher der größe *size*, welche den gleichen Inhalt wie *ptr* hat.
- **free(ptr)**: Gibt den Speicher von *ptr* wieder frei.

Beim Arena-Allokator wird zu Beginn ein großer Block an Speicher allokiert und zwar 10MB über *malloc*.
Zudem besitzt der Allokator einen Zeiger auf die nächste Speicherstelle, welche vergeben werden kann.
Das Äquivalent von *malloc*, *realloc* und *free* für den Arena-Allokator funktioniert wie folgt:

- **malloc:** Speicher die Allokationsgröße in den ersten 8 Bytes und gebe den Zeiger um 8 Bytes verschoben zurück. Erhöhe den Zeiger intern um die gesamte Allokationsgröße.

- **realloc:** Allokiere einfach die neue Größe mit dem Arena malloc und kopiere die Bytes.
- **free:** Diese Operation führt nichts aus.

Der Grund für das Abspeichern der Allokationsgröße ist das Ermöglichen von *realloc*, da in *realloc* selber die Größe von *ptr* nicht weitergeben wird, sondern nur die neue Größe.
Der Allokator wird zurückgesetzt, indem der Pointer einfach auf den Anfang des Blocks zeigt. Ebenfalls werden bei allen Allokationen die angefragte Größe *aligned*. Zum Beispiel anstatt 17 Bytes werden 24 Bytes allokiert. Dies sorgt dafür, dass alle Speicheradressen stets 8 Byte *aligned* sind.
Die 10MB sollten stets ausreichen. Sollte dies jedoch nicht der Fall sein , so wird ein weiterer Block allokiert.
Zudem erfordert diese Methode keine Iteration durch alle generierten Knoten, um sie zu löschen oder zurückzusetzen. Das Zurücksetzen des Blocks ist $O(1)$ und skaliert damit auch viel besser für höhere Tiefenlimits und Levelgrößen.
Da beide Allokatoren dasselbe Interface verwenden, kann der aktive Allokator in einer globalen Variable abgespeichert werden und zudem muss kein Code selber etwas abändern.
Algorithmus 5 zeigt den Übergang von der alten *Default-Policy* zu der neuen.
Diese Optimierung hat keinen direkten Einfluss, also läuft der Algorithmus grundlegend gleich ab. Der Algorithmus wird mit dieser Optimierung lediglich schneller.

Algorithm 5: default policy adjustment

```
function old_default_policy(node):
    branch ← clone(node)
    basic_default_policy(branch)
    score ← branch.score
    delete_node_branch(branch)
    return score
function new_default_policy(node):
    activate_arena_allocator()
    branch ← clone(node)
    basic_default_policy(branch)
    reset_arena_allocator()
    activate_default_allocator()
    return branch.score
```

Abbildung 2.14: Default-Policy Anpassung

2.3.8 Bootstrapping

In Generierungsmethoden wie [12] hat man Level-Templates verwendet und mit diesen Level generiert. Im Moment findet nichts dergleichen im Algorithmus statt.
Im Moment muss eine recht hohe Tiefe erreicht werden, bis überhaupt ein verwendbares Level generiert wird. Sollte demnach ab einer hohen Tiefe ein Knoten generiert werden, welcher bei der Simulation ein sehr gutes Level erstellt hat, so kann es sehr viele Iterationen erfordern, bis das Level mehrere Male expandiert wird.

Die Idee ist, dass sich auf das gut bewertete Level fokussiert wird, um das Level möglichst zu optimieren hinsichtlich der Bewertung und damit auch Schwierigkeit.
Der Ablauf hierfür ist wie folgt:

- Führe Iterationen im MCTS durch bis zu einem Timeout aus
- Wähle die letzten n neu abgespeicherten Level
- Erzeuge einen neuen MCTS und füge diese Level als Kindknoten vom Wurzelknoten hinzu
- Führe Iterationen im neuen MCTS bis zu einem Timeout aus

Bei diesem Ablauf wird in der zweiten Iterationsabfolge das Tiefenlimit deaktiviert, da bereits ein fast fertiges Level vorliegt.
Ebenfalls werden alle Zielpositionen der Level gelöscht. Also werden Level in einen Zustand vor *Freeze* zurückgesetzt.
Der Wurzelknoten wird hierfür auch als vollständig expandiert markiert, da dieser eigentlich einen nicht existenten Zustand repräsentiert.
Zudem kann der zweite Timeout sehr niedrig gewählt werden, da sich sehr schnell verbesserte Level generieren. In dem Programmierprojekt wurde hierfür ein δ Wert hinzugefügt, welcher dem *Bootstrapping* $\delta \cdot Timeout$ viel Zeit zur Verfügung stellt.

2.4 Sekundäre Eigenschaften

Die generierten Level folgen generell keinem levelweitem Muster. Bestimmte Muster lassen sich jedoch mittels Erweiterungen der Bewertungsmethode hinzufügen. In Abbildung 2.15 sieht man ein Beispiel für ein diagonales Level.

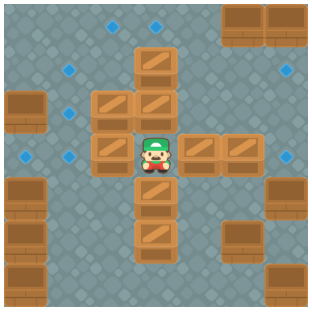

Abbildung 2.15: Levelbewertung mit Begünstigung von Diagonalität.

Texturen aus [9]

Das Problem an solchen Bewertungen ist, dass der Schwierigkeitsgrad dazu tendiert um einiges niedriger zu sein. Etwas ähnliches könnte hierfür mittels *Bootstrapping* erzielt werden. Die Implementierung dessen erlaubt es beliebig viele und beliebig aussehende Level dem Baum hinzuzufügen. Demnach könnte ein Level mit einer bestimmten Grundstruktur hinzugefügt werden und dem Algorithmus zur Optimierung beziehungsweise weiteren Befüllung überlassen werden.

Implementierungen solcher Bewertungserweiterungen lassen sich im Programmierprojekt sehen. Es konnten keine Erweiterungen erstellt werden, welche nicht die Levelqualität zu stark Beeinträchtigen.

3 Ergebnisse

Das Projekt wurde in C++ verfasst. Der Compiler für die Experimente ist g++ 11.4.0 in Ubuntu 22.04. Die C++ Version ist c++17. Für die Experimente wurde die höchste Optimierungsstufe gewählt und zwar -O3.
Der Prozessor für die Experimente ist ein AMD Ryzen(TM) 5 3600 mit 3.6GHz.
Der Arbeitsspeicher ist $2 \cdot 8$ GB mit $2133\frac{MT}{s}$
Als eine Iteration vom MCTS wird hierbei eine Ausführung von $mcts\_search$ aus Algorithmus 1 verstanden.
Bei den Experimenten wird generell ein Timeout gewählt. Die Alternative wäre, eine maximale Iterationszahl zu wählen. Jedoch können bestimmte Optimierungen, wie zum Beispiel das Tiefenlimit dazu führen, dass eine Iteration länger braucht mit einem höheren Wert. Ebenfalls ist ein Timeout interessanter hinsichtlich der praktischen Verwendung.
Deshalb wird ein Timeout von 30s pro Iteration für die Experimente gewählt.
Bei den Experimenten werden 10 Iterationen durchgeführt. In jeder Iteration wird ein neuer *seed* gewählt und für alle MCTS verwendet.
Gemessen wird die Iterationszahl, Bestbewertung und die Dauer bis zur Bestbewertung. Angeben werden diese Datenpunkte dann als Durchschnitt über die Iterationen in der Tabelle.
Die generierten Level haben eine Größe von 5×5 und die mittlere Position $(2, 2)$ ist die Startposition. Anfangs sind alle Optimierungen, außer die *Tree-Policy* deaktiviert. Iterativ werden die Optimierungen hinzugefügt. Sollte sich eine Optimierung als gut erweisen, so wird diese in das nächste Experiment übertragen.
Zudem wird die Standardabweichung σ hinsichtlich der Bestbewertung gemessen, welches bei allen Experimenten unter $0,1$ liegt.
Es wird anfangs UCB1 verwendet mit der Konstante $\frac{1}{\sqrt{2}}$.
Es wurde ebenfalls sichergestellt, dass die GUI für die Experimente nicht initialisiert wird.

3.1 Parameterwahl

In [7] wurden keine Gewichte für Congestion-V2 angegeben, wodurch die Verwendung von den angegebenen Gewichten für w_b, w_c, w_n unklar ist.
Jedoch haben die Autoren fünf Level angegeben mit aufsteigenden Bewertungen $0.4, 0.6, 0.8, 1.0, 1.2$ angegeben [7, S.62].
Diese fünf Level und die Markierung von Boxposition und Zielposition für die Metriken wurde repliziert.
Es gilt anzumerken, dass die besser bewerteten Level mehrere Lösungen besitzen und damit je nach Lösungsweg Congestion-V2 verschiedene Werte annehmen kann.
Die beiden bestbewerteten Level wurden nach über drei Minuten in [7] generiert.

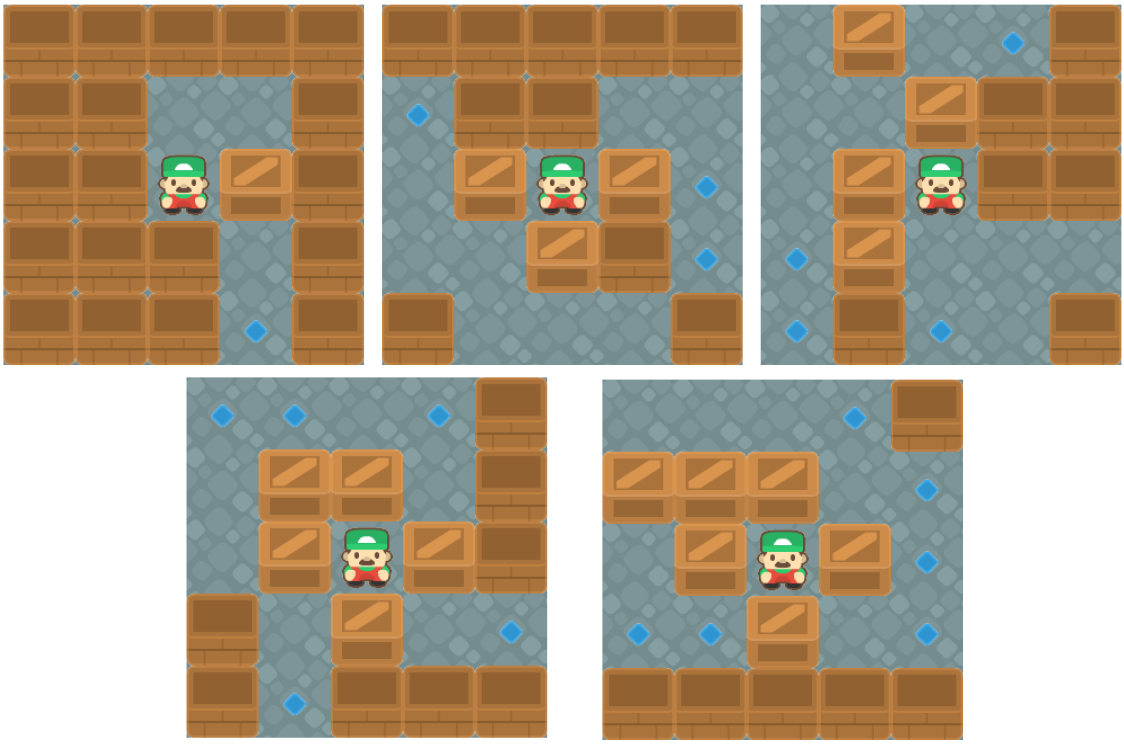

Abbildung 3.1: Die fünf Beispiellevel

Texturen aus [9]

Diese fünf Beispiellevel aus [7, S.62] wurden in Abbildung 3.1 nacherstellt. Diese Level dienen als Grundlage für die Wahl der Parameter.
Zunächst wurden die Gewichte analytisch gewählt.
k ist ein Normalisierungsfaktor und konnte von daher einfach auf einen festen Wert gesetzt werden $k = 50$. Dann wurden über die Gewichte iteriert für α, β, γ über die Werte $[0.1, 2.0]$ in $0,1$ schritten. Für w_b, w_c, w_n über die Werte $[1, 20] \subseteq \mathbb{N}$ in 1 schritten.
Für alle 6-Tuple wurde die Standardabweichung zu den Bewertungen des Beispiellevels berechnet. Das 6-Tuple mit der kleinsten Standardabweichung wurde gewählt. Die generierten Level für dieses Tuple waren generell ganz gut, jedoch war $w_b = 1$ und damit besaßen die besten Level keine Blöcke. Blocklose Level sind generell sehr einfach zu lösen und bieten keine Tiefe. Gut gesetze Blöcke können massiv die Levelschwierigkeit erhöhen, wie an Congestion-V2 in [7] zu sehen ist.
Aufgrund dessen wurde $w_b = 3$ gesetzt und $k = 55$ gesetzt. Damit ergibt sich im gesamten.

- $\alpha = 1,9$
- $\beta = 0,1$
- $\gamma = 1,3$
- $w_b = 3$
- $w_c = 7$
- $w_n = 8$

$\beta = 0,1$ geht gleich auf mit persönlichen Beobachtungen und zwar besitzt die Boxanzahl eine größere Wichtigkeit für die Schwierigkeit.

3.2 Move-Agent

In diesem Experiment wird die Optimierung für *Move-Agent* mit der naiven Variante verglichen. In der naiven Variante werden von der Spielerposition aus alle vier benachbarten Felder betrachtet. Für jedes Feld, zu dem sich der Spieler bewegen kann, wird die Bewegung abgespeichert ähnlich zu der Optimierung.

Move-Agent		
	Optimiert	Standard
Iterationszahl	4498609	4972319
Bewertung	1,1480	0,8199
Dauer	11,7857	12,4690
σ	0,0457	0,0245

Tabelle 3.1: Move-Agent Experiment

Bei der optimierten Variante kommt es zu weniger Iterationen, jedoch erreicht es besser bewertete Level.

3.3 Boxlimits

Die *Move-Agent* Optimierung ist hier aktiv.
Bei diesem Experiment wird das untere Boxlimit auf 1 gestellt. Betrachtet werden die verschiedenen oberen Boxlimits. ∞ ist hierbei der Wert ohne die Optimierung.
Im Methoden Teil wurden zwei obere Limits empfohlen und zwar $\lceil \frac{5 \cdot 5}{3} \rceil = 9$ und $\lceil \frac{5 \cdot 5}{2} \rceil = 13$.

Oberes Boxlimit					
	∞	20	15	10	5
Interationszahl	3600105	3632097	3650202	3709194	3950324
Bewertung	1,1507	1,1433	1,1363	1,2997	1,3518
Dauer	15,8943	16,0630	12,3663	17,4361	14,7178
σ	0,0826	0,0521	0,0612	0,0193	0,0525

Tabelle 3.2: Oberes Boxlimit

3.4 Tiefenlimit

Hier werden verschiedene Werte für das untere Tiefenlimit betrachtet. Ein Tiefenlimit von 0 bedeutet hierbei, dass die Optimierung deaktiviert ist. Ein unteres Boxlimit von 1 und ein oberes Boxlimit von 9 sind hier aktiv.
Ebenfalls noch einmal anzumerken gilt, dass grundlegend ein Tiefenlimit von 7 begründet wurde.

Tiefenlimit					
	0	5	7	10	15
Iterationszahl	3527152	3562598	3474843	3108058	2722091
Bewertung	1,3678	1,3710	1,3605	1,3931	1,3654
Dauer	10,8055	15,1329	14,5628	17,0468	9,1510
σ	0,0635	0,0825	0,0415	0,0403	0,0260

Tabelle 3.3: Tiefenlimit

Die Bewertungen sind alle fast gleich, jedoch verringert sich die Iterationszahl und damit auch der Speicherverbrauch.

3.5 Konfigurationen

Ein Tiefenlimit von 10 ist hier aktiv.

Konfigurationen		
	Standard	Optimiert
Iterationszahl	3135409	3091629
Bewertung	1,3557	1,4439
Dauer	14,5781	10,6731
σ	0,0516	0,0614

Tabelle 3.4: Konfiguration

3.6 Allokator

Hier werden die verschiedenen Allokationsmethoden verglichen. Das Entfernen der Konfigurationen wird hinzugefügt.

Allokator		
	Standard	Optimiert
Iterationszahl	3069198	3897098
Bewertung	1,4589	1,4611
Dauer	14,6182	13,5235
σ	0,0462	0,0445

Tabelle 3.5: Allokator

Die optimierte Variante schafft in etwa 27% mehr Iterationen.

3.7 Bootstrapping

Die Allokator-Optimierung wird hinzugefügt. Hierfür wurde n=4 verwendet. Es werden verschiedene δ Werte betrachtet. Zudem wird der Start angegeben, welcher bei $30 \cdot (1 - \delta)$ liegt. $\delta = 0$ bedeutet kein Bootstrapping.
Die Iterationszahl für $\delta > 0$ ist die Iterationszahl seit *Start*.

δ					
	0	0,05	0,10	0,15	0,20
Iterationszahl	3924687	1170963	2425516	3625749	4948784
Bewertung	1,4350	1,4705	1,4516	1,4583	1,4769
Dauer	15,0072	28,5884	27,0664	25,6074	24,0890
Start	-	28,50	27	25,50	24
σ	0,0506	0,0780	0,0725	0,0749	0,0685

Tabelle 3.6: Bootstrap

3.8 Explorationsfaktor

Folgende Betrachtungen sind ohne *Bootstrapping*.
In UCB1 gibt es den Explorationsfaktor, welcher standardmäßig $2C = \frac{2}{\sqrt{2}} = \sqrt{2}$ ist.
Es wird untersucht, ob ein höherer Explorationsfaktor, zu besseren Ergebnissen führt mit $\xi \cdot C = \frac{\xi}{\sqrt{2}}$.
$\xi = 1$ ist hierbei der Standardwert.
Hierfür werden 30 Iterationen betrachtet.

ξ						
	1	2	3	4	5	6
Iterationszahl	3939814	3807184	3760231	3737064	3722411	3715273
Bewertung	1,4231	1,4582	1,4252	1,4307	1,4302	1,4315
Dauer	17,7743	16,4087	14,1706	13,9001	12,8775	16,4749
σ	0,0456	0,0563	0,0451	0,0565	0,0522	0,0583

Tabelle 3.7: Exploration

Die Bewertungen sind alle fast gleich, jedoch sinkt die Iterationszahl mit höherem ξ.

3.9 UCB Alternativen

Für dieses Experiment wird $C = \frac{2}{\sqrt{2}}$ verwendet für *UCB1* und *SP MCTS*. $C = \frac{16}{\sqrt{2}}$ für *UCB-V* und $C = \frac{8}{\sqrt{2}}$ für *UCB1 Tuned*.
Hierfür werden 30 Iterationen betrachtet.

UCT				
	UCB1	UCB1 Tuned	UCB-V	SP MCTS
Iterationszahl	3773864	3764409	3790436	3774020
Bewertung	1,4322	1,4407	1,4344	1,4371
Dauer	18,1368	15,3107	13,8132	15,0853
σ	0,0552	0,0762	0,0564	0,0409

Tabelle 3.8: Erweiterungen

3.10 Gesamt

Nun wird der Algorithmus noch einmal im gesamten betrachtet mit allen Optimierungen und verschiedenen Feldgrößen. Zunächst eine grundlegende Ausführung ohne *Bootstrapping*. Hier wurden über 800 000 Iterationen für jede Iteration folgendes gemessen:

- (Rot) Die Dauer der Iteration
- (Blau) Die gesamte Allokation zu dem Zeitpunkt
- (Gelb) Die Tiefe in welcher der *Rollout* stattgefunden hat.
- (Grün) Die momentane Bestbewertung

Die Allokationsgröße wurde mit der Linux exklusiven Funktion *mallinfo2* gemessen siehe [10]. Für die Graphen wurden uniform 10000 Datenpunkte gewählt, da es ansonsten zu viele Datenpunkte wären.

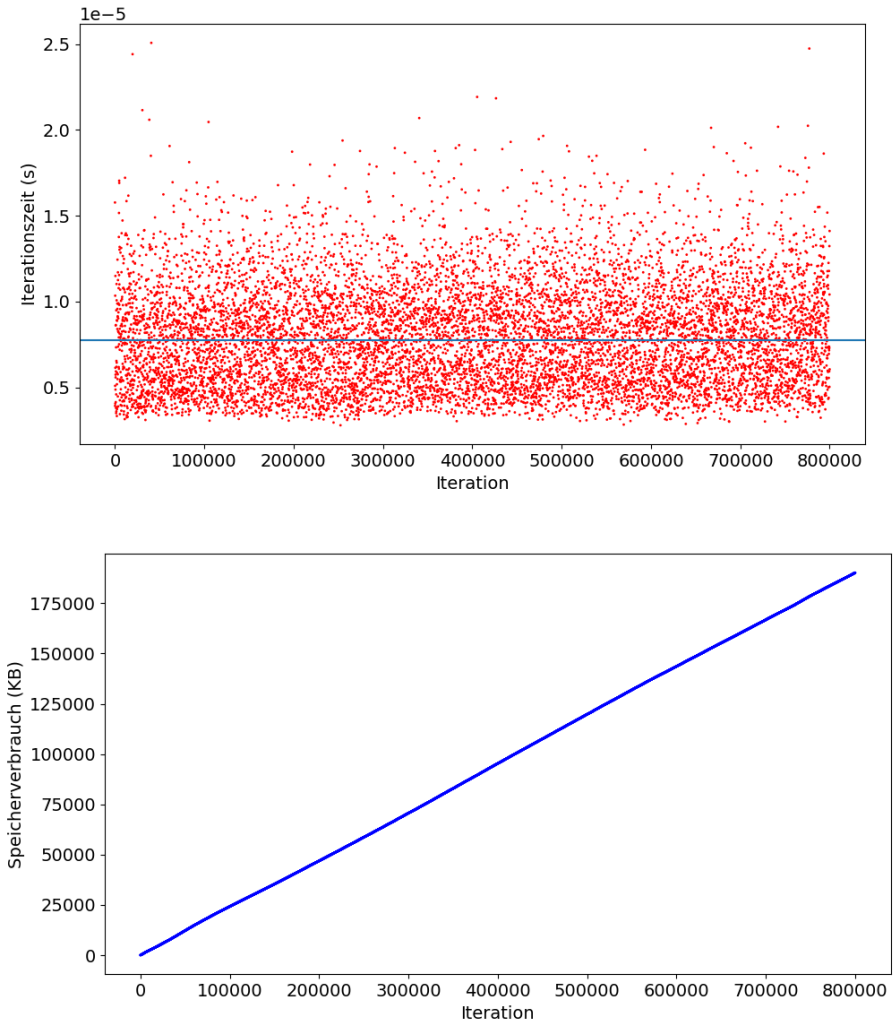

Abbildung 3.2: Iterationsdauer und Gesamtallokation
Graph erstellt mit [6]

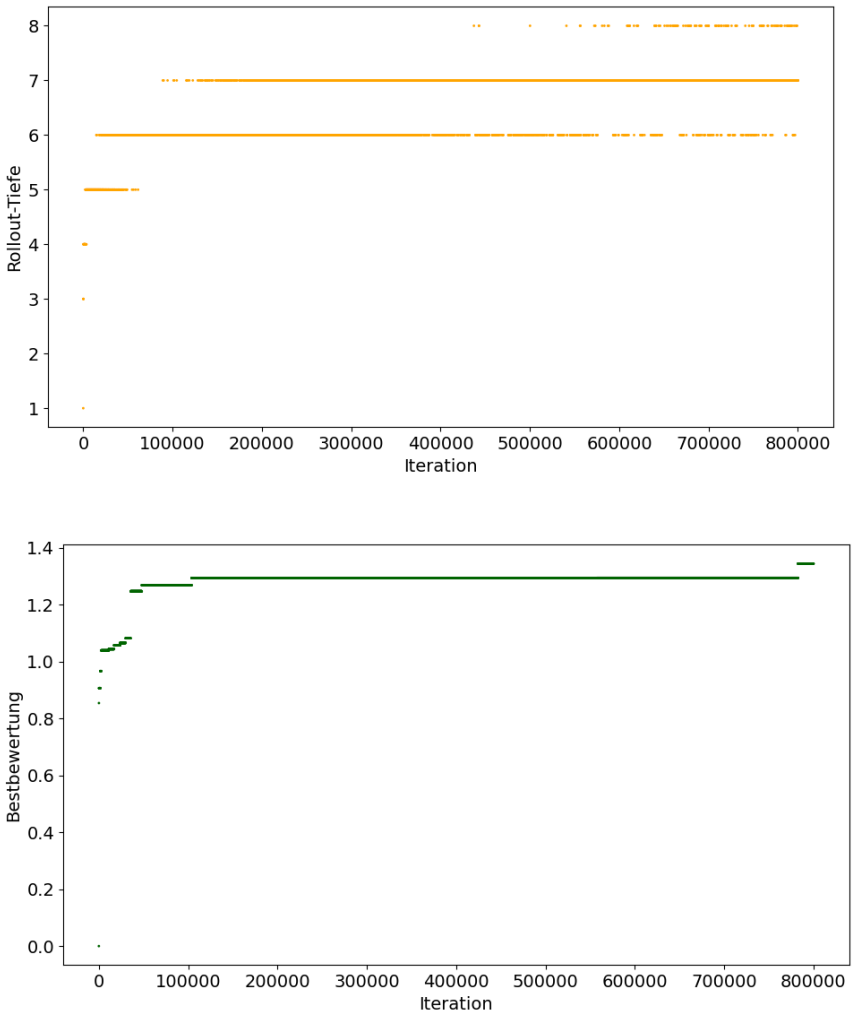

Abbildung 3.3: Rollout-Tiefe und Bestbewertung
Graph erstellt mit [6]

Der Graph für die gesamte Allokation steigt linear. Hohe Bewertungen werden sehr früh erreicht und die Tiefe findet in der Nähe von 7 statt. Bei der Iterationszeit ist zudem die Regressionsgerade zu sehen, welche einen sehr kleinen Anstieg besitzt.

Folgend werden Beispielgenerierungen betrachtet. Links sieht man das bestbewertete Level mit dem Graphen, welcher den Generierungsverlauf zeigt hinsichtlich der Bestbewertungen. Rechts das selbe nach dem Bootstrapping, weshalb rechts im Graphen der Zeitverlauf auch deutlich geringer ist.

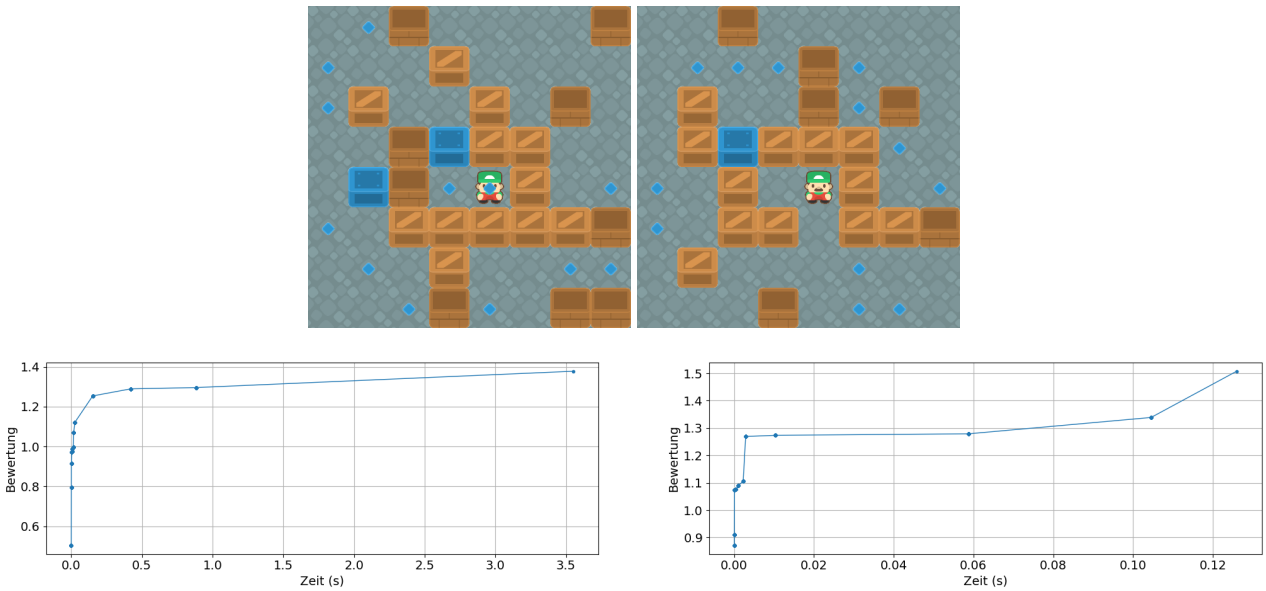

Abbildung 3.4: Beispielgenerierung 8×8
Graph erstellt mit [6]

Texturen aus [9]

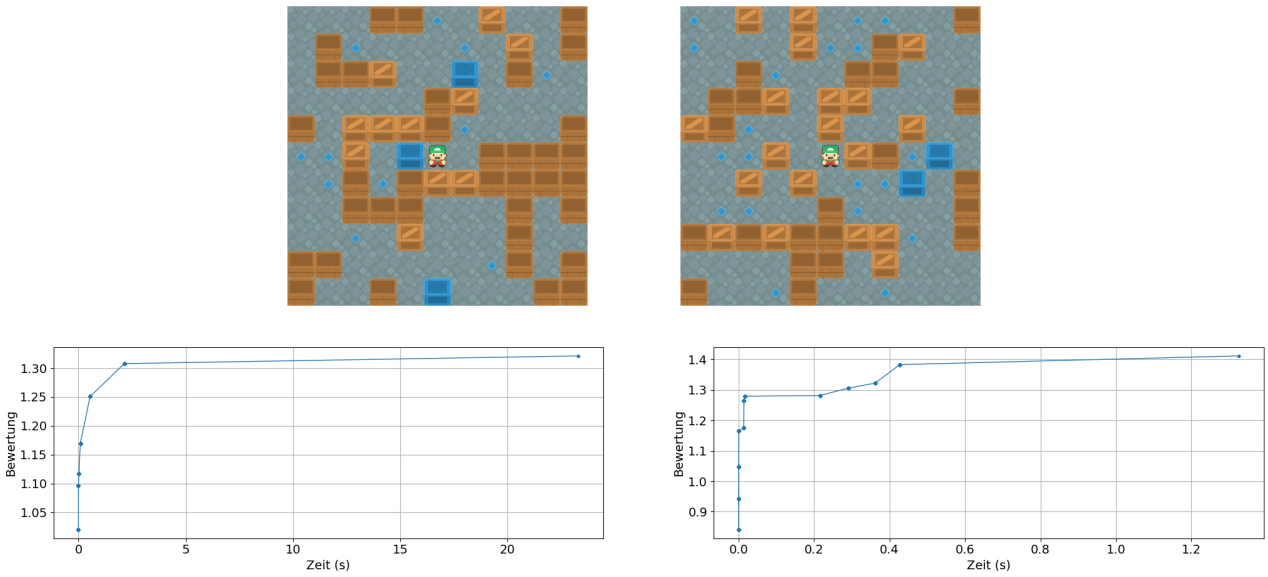

Abbildung 3.5: Beispielgenerierung 11×11
Graph erstellt mit [6]

Texturen aus [9]

Noch ein letzter Ablauf, bei welchem alle Level der Menge hinzugefügt werden, welche entweder neue Bestbewertungen sind oder eine Bewertung von über 1,3 besitzen. In Abbildung 3.7 sind die sechs besten Level.

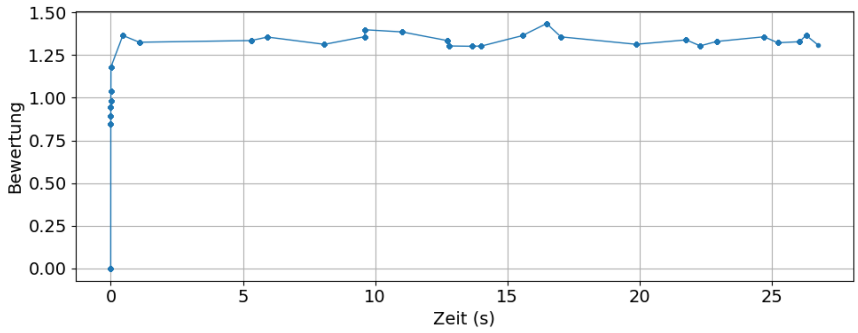

Abbildung 3.6: Generierung von Leveln über 1,3
Graph erstellt mit [6]

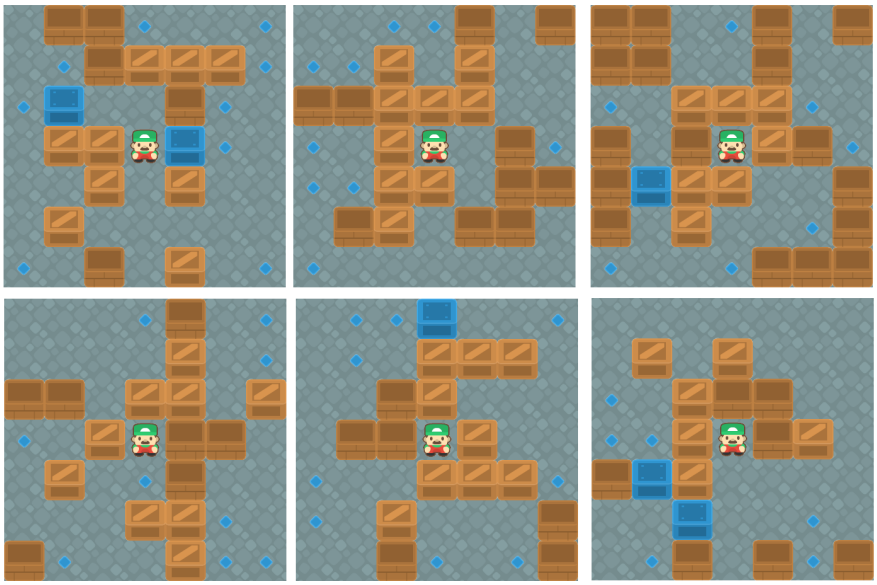

Abbildung 3.7: 7x7 Beispiel

Texturen aus [9]

4 Diskussion

4.1 Experimente

4.1.1 Move-Agent

Bei *Move-Agent* kann man eine klare Verbesserung gegenüber der Standardvariante feststellen. Die Iterationsanzahl bei der naiven Variante ist höher, was zu erwarten ist, da die naive Variante um einiges schneller bei der Bestimmung der nächsten Bewegungen ist, als die optimierte Variante.
Die optimierte Variante erreicht um einiges höhere Levelbewertungen. Es gilt hierbei auch zu beachten, dass der Sprung von schlecht bewerteten Leveln zu mittleren Leveln um einiges niedriger ist, als der von mittleren zu sehr gut bewerteten Level. Ebenfalls ist eine niedrige Iterationsanzahl in diesem Fall besser als eine höhere, da der Speicherverbrauch um einiges niedriger ausfällt in Abhängigkeit zur Bewertungsverbesserung.

4.1.2 Tiefenlimit

Je höher das Tiefenlimit, umso kleiner ist die Iterationszahl. Dies ist zu erwarten, da das Programm die meiste Zeit in der *Default-Policy* verbringt und diese mit einem erhöhten Tiefenlimit noch länger braucht im Durchschnitt.
Mit dem Tiefenlimit kann der Speicherverbrauch verringert werden, da man gleiche Bewertungen mit weniger Iterationen schafft. Man könnte das Tiefenlimit noch höher setzen für einen geringeren Speicherverbrauch, jedoch würden sich dann wahrscheinlich keine einfacheren Level mehr bilden.
Das Tiefenlimit war eines der ersten Optimierungen, welches durch die später optimierte *Move-Agent* Aktion und generelle Strukturverbesserung weniger wirksam geworden ist.

4.1.3 Boxlimit

Dass die Iterationszahl mit sinkendem Boxlimit steigt war zu erwarten, da die Anzahl an möglichen Aktionen reduziert wird und damit schneller ein Endknoten erreicht werden kann. Während die Iterationszahl recht langsam steigt, steigt die Bestbewertung sehr stark an. Die lässt sich damit begründen, dass dieses Limit eine Übersättigung von schlechten Boxen verhindert.
Das ein Limit von 5 hier am besten abschneidet ist sehr überraschend, da sich auf einem 5×5 Feld durchaus mehr Boxen befinden können.

4.1.4 Konfigurationen

Die optimierte Variante konnte eine Verbesserung darstellen. Interessant ist der kleine Einfluss auf die Iterationszahl, denn der Filter selber hat eine Laufzeit von $O(4nm)$ und er erhöht die Aktionsanzahl. Dies sollte die Wahrscheinlichkeit für das Erreichen eines Endzustands verringern.

4.1.5 Allokator

Es war recht überraschend um wie viel schneller diese Optimierung ist. Diese Optimierung hat keinen direkten Einfluss auf den Algorithmus und erlaubt es lediglich mehr Iterationen zu vollführen.

4.1.6 Bootstrapping

Beim *Bootstrapping* erkennt man, dass sehr schnell das Maximum erreicht wird und damit die Zeit hierfür möglichst klein gesetzt werden sollte. *Bootstrapping* schafft es in sehr kurzer Zeit selbst bei einem sehr gut bewertetem Level die Bewertung noch höher zu bekommen. Dies ist mit der sehr hohen *Exploitation* zu begründen.
Ebenfalls kann bei Bootstrapping der alte Baum verworfen werden, um damit Speicherplatz frei zu machen. Die Level welche dort erstellt werden ähneln sich alle sehr stark, da diese vom selben Elternknoten abstammen. Würde man diesen Algorithmus mehrere Male wiederholen wollen, so müsste man einen extra Filter einbauen, welche sehr ähnliche Level entfernt.
Etwas verwunderlich ist ebenfalls die extrem hohe Iterationszahl. Begründen ließe sie sich damit, dass viele Limits erreicht werden. Das Tiefenlimit zum Beispiel ist vollkommen deaktiviert. Das Boxlimit ist höchstwahrscheinlich ebenfalls bereits erschöpft. Zudem bestehen kaum noch löschbare Blöcke. Damit ist die Anzahl an Aktionen drastisch reduziert und ein Endzustand kann schneller erreicht werden. Zudem führt eine Vergrößerung von C für UCB1 hier zu einer drastischen Reduzierung der Iterationszahl.

Ein Problem dieser Methode, welche angemerkt werden sollte ist, dass bei kleineren Levelgrößen mit den vorhandenen Limits die Levelmetriken zu stark optimiert werden können. Dies führt dazu, dass die Level merkbar schlechter werden können. Bei größeren Levelgrößen ist dies generell nicht der Fall. Ebenfalls kann die Hinzunahme von allen Knoten, die eine bestimmte Mindesbewertung besitzen, einen starken Einfluss auf diese Methode haben.

4.1.7 UCB Alternativen

Die Ergebnisse für den Explorationsfaktor und die UCB1-Alternativen waren recht inkonsistent hinsichlich der Bestbewertungen zueinander. Grundlegend kann *UCB1-TUNED* sich als eine Alternative herausstellen.

4.1.8 Gesamt

Es war etwas verwunderlich, dass die Expansionstiefe sich um die Zahl 7 festigt, deshalb eine Betrachtung zum Wachstum vom Baum.
Bei einer jeden *Delete-Obstacle* können bis zu drei weitere *Delete-Obstacle* und eine *Place-Box* Aktionen im nächsten Knoten möglich sein. Außerdem fängt der Baum mit vier möglichen *Delete* Aktionen an. Damit wächst der Baum grundlegend im Bereich $\frac{(4+n)!}{3!}$ bis zu einem *Freeze* beziehungsweise gerade anfangs durchaus darüber. Dahingehend kann bereits bei einer geringen Baumtiefe und auch mit den Limits mehrere Millionen Knoten existieren.
Nach einem *Freeze* kann ein Knoten maximal $4 \cdot b + 1$ viele Aktionen besitzen mit b als Boxanzahl. Da eine Box in vier Richtungen bewegt werden kann und es noch *Evaluate Level* gibt. Damit ist es

ein *Worst Case* Wachstum von $(4 \cdot b + 1)^n$.
Dass die Iterationszeit fast konstant bleibt, ist grundlegend zu erwarten mit allen Limits. Die Erhöhung der Baumtiefe sollte bei der geringen Größe ebenfalls keinen signifikanten Effekt haben.
Dem Graphen für die Allokationsgröße kann man entnehmen, dass die Allokation in jeder Iteration fast gleich ist. Dies ist zu erwarten, da die Feldgröße konstant bleibt und lediglich ein Knoten pro Iteration dem Baum hinzugefügt wird. Alle Allokationen für Berechnungen und der *Default-Policy* sind nicht persistent.

4.2 Weiteres

Ein Aspekt der Levelgenerierung, welcher ebenfalls wichtig ist die Inklusion neuer nicht Maximalwerte. Hierfür gibt es im Programmierprojekt Einstellungen, bei welchem dem *Level-Set* alle Level hinzugefügt werden, welche einen bestimmten Wert erreichen. Es kommt sehr oft vor, dass wenn der Maximalwert bei zum Beispiel 1,31 liegt sehr viele unterschiedliche Level erzeugt werden die nur etwas darunter liegen. Beziehungsweise insgesamt ist die Anzahl an diversen Level mit einer guten Bewertung ziemlich hoch und mit diesen Einstellungen können diese inkludiert werden. Dies konnte man im letzten Experiment sehen. Im Programmierprojekt kann diese Hinzunahme ebenfalls eingestellt werden in *settings.h*.
Generell ist der Speicherverbrauch vom Programm sehr hoch mit 1GB pro Minute. Jedoch erreicht das Programm recht schnell ein bestimmtes Maximum mit vielen gut bewerteten Leveln. 30s scheint hierbei generell ein sinnvoller Timeout zu sein, wobei meist nur wenige Sekunden ausreichen. Ein größerer Timeout ist hierbei eher sinnvoll, um weitere Level bestimmter Mindestbewertungen hinzuzufügen.

4.3 Reproduzierbarkeit und Restriktionen

In dieser Arbeit konnte kein direkter Vergleich zu [7] erstellt werden, dennoch konnten klare MCTS Verbesserungen vorgestellt werden für den Ablauf des simulierten Spielens. Gerade die präsentierten Levelmetriken aus [7] erforderten mehrere Iterationen in der Implementierung, aufgrund eines Mangels an Details. Dennoch konnte ein Algorithmus basierend auf [7] erstellt werden, welcher ähnlich interessante Level erstellen kann und dies in vergleichsweise sehr kurzer Zeit.

Hinsichtlich der Wahl der Optimierungen und Parameter der Experimente gibt es sehr viele verschiedene Kombinationsmöglichkeiten. Die Anzahl an Möglichkeiten ist viel zu hoch, um sie hier vernünftig darzustellen. Gerade hinsichtlich des hohen Timeouts und der Notwendigkeit möglichst viele Iterationen zu haben, da die gesamte Levelgenerierung auf Zufallszahlen beruht.
Deshalb ist die Wahl der Optimierungen und deren Parameter höchstwahrscheinlich nicht die optimale Kombination. Ein erwähnter Parameter, welcher ebenfalls einen Einfluss auf die Metriken besitzt, ist die Hinzunahme aller Level einer bestimmten Mindestbewertung.

Die Levelgröße in dieser Arbeit ist auf einen Flächeninhalt von 254 beschränkt beziehungsweise einer Feldgröße von etwa $(16, 15)$. Diese Limitierung wurde gewählt zugunsten vom Speicherverbrauch und da man in [7] nur kleinere Feldgrößen betrachtet hat. Diese Maximalgröße scheint auch der Punkt zu sein, ab welchem nicht mehr sinnvolle Level erstellt werden können.

Die Levelmetriken sind weder eine notwendige Bedingung noch eine hinreichende Bedingung für gute Level. Jedoch existiert nach [7] eine Korrelation zur Schwierigkeit eines Levels. Diese Korrelation besteht ebenfalls nach persönlichem Empfinden.
Die größte Restriktion der Levelgenerierung lässt sich in *Congestion-V2* sehen. Während in *Congestion-V1* der Flächeninhalt keine Rolle gespielt können sich Level bilden, in welchen es eine große Distanz zwischen Box und Zielposition gibt. In *Congestion-V2* ist es jedoch unwahrscheinlicher. Also weisen Level die generelle Tendenz auf, dass sich die Level sehr lokal lösen lassen. Dies ist vor allem bei größeren Leveln zu beobachten, wo es dazu vorkommen kann, dass man klare Lösungen sehr nahe zu Boxen erkennen kann.
Dennoch gilt es anzumerken, dass die generierten Level eine arbiträr hohe Anzahl an möglichen Lösungen haben können. Dahingehend wird in *Congestion-V2* wahrscheinlicher eine optimale Lösung in Betracht gezogen, da bei einer willkürlich großen Distanz es wahrscheinlicher ist, dass es andere Lösungen gibt.

Insgesamt können oft Level erstellt werden, welche einen bestimmten Kniff besitzen und nicht direkt lösbar sind. Jedoch fallen einem beim Lösen der Level oftmals bestimmte Lösungsmuster auf und vor allem der Bias, dass die vorhergesehene Zielposition für eine Box oftmals eine nahe Zielposition ist. Jedoch sind die generierten Level vom Aufbau her recht einzigartig, gerade bei größeren Leveln. Ebenfalls erstellt dieser Levelgenerator von sich aus keine ästhetisch interessanten Level, jedoch könnte wie bereits besprochen mit *Bootstrapping* etwas ausgeholfen werden. Der Grund für den Mangel an Ästhetik ist der *3x3-Block-Count*, denn dieser belohnt nur Heterogenität in einem kleinen Bereich.

5 Fazit

5.1 Thema und Ergebnisse

In dieser Arbeit konnten bestimmte Schlüsse gezogen werden hinsichtlich der Anwendung und Optimierung für die Levelgenerierung mittels simulierten Spielen in einem MCTS.

- Für jedes dynamische Spielobjekt sollte eine Mindest - und Maximalanzahl gewählt werden in Relation zu der Levelgröße
- Man sollte entscheiden, wo das Mindestlimit für ein valides Level liegt.
- Es sollten nicht direkt Spielbewegungen betrachtet werden. Stattdessen sollten nur alle Spielbewegungen betrachtet werden, welche auch tatsächlich einen direkten Einfluss auf das Level besitzen.
- Level können nicht interagierbare dynamische Objekte besitzen und es kann lohnenswert sein diese partiell zu entfernen.
- *Bootstrapping* kann zur Erzeugung besserer Level führen und für bestimmte Topologien verwendet werden.

Optimierungen, welche generell für alle MCTS interessant sind ist hierbei vor allem die Speicherverwaltung in *Default-Policy*. Insbesondere die Form der Implementierung in dieser Arbeit erlaubt es standardgemäß die *Default-Policy* durchzuführen ohne, dass irgendwelche Änderungen in der *Default-Policy* selber notwendig sind.

Die vorgestellten Bewertungsmetriken korrelieren grundlegend mit dem Schwierigkeitsgrad. Zudem lassen sich die Metriken auch auf andere Spiele übertragen und sie sind schnell zu berechnen. Die generierten Level können aufgrund der Metriken nicht ästhetisch interessant sein, jedoch besitzen sie oftmals einen interessanten Schwierigkeitsgrad.

Insgesamt können die hier vorgestellten Optimierungen hilfreich sein für weitere Implementierungen solcher Algorithmen. Auch zeigt die Implementation, dass solch ein Algorithmus sehr schnell gut bewertete Ergebnisse produzieren kann.

5.2 Weiteres

Die Levelmetriken in dieser Arbeit können sich eigentlich auf sehr viele Spiele anwenden lassen. Wichtig wäre es dennoch für zukünftliche Anwendungen weitere Levelmetriken zu formen. Vor allem eine für die Bildung einer interessantere Topologie. Außerdem könnten Baumpfade in Betracht

gezogen werden für solche Metriken, gerade da eine Iteration über den Baumpfad in der Bewertungspropagierung sowieso stattfindet. Auch könnte man den Baumpfad in Betracht ziehen, um Repetion im Spielvorgang zu vermeiden.

In [13] werden verschiedene grundlegende Formen der Bewertungsmöglichkeiten von Leveln besprochen, welche sich jedoch nicht ganz auf Sokoban anwenden lassen.
Die hier vorgestellte Repräsentation eines Zustands ist als eine einfaches Brett mit ein paar extra Variablen. In [13] stellen die Autoren die Idee vor Level minimaler und eigenschaftsbasierter darzustellen. Man könne Level betrachten mittels dem Zählen bestimmter Eigenschaften wie Objektanzahl, Pfadlänge und Zweigungsfaktor [13, S.176]. Auch könne man alle Objekte in einer separaten Liste halten zusammen mit Eigenschaften [13, S.175].

Ebenfalls müsste man bei Optimierungen hinsichtlich der Bewegungen wie für *Move-Agent* Metriken in Betracht ziehen, welche die Aktionsüberführungen gewichten zum Beispiel mittels der Distanz.
Für die Implementierung vom Algorithmus müssen während dem simuliertem Spielen Metadaten für die Boxen abgespeichert werden und zwar Startposition und *Move-Counter*. Diese Herangehensweise könnte generell erweitert werden und weitere Metadaten könnten in Betracht gezogen werden.
Eine wichtige generische Optimierung, welche nicht erwähnt wurde, ist das *Multithreading*. Auch währen bestimmte Formen von *pruning* und Serialisierung für den Speicherverbrauch wichtig.
Weitere Methoden hinsichtlich der Levelbildung währen Verkettungen von Leveln und eine dynamische Vergrößerung der Levelgröße mit der Baumtiefe oder Schiebeaktionen, welche gegen eine Wand gehen würden.

Literaturverzeichnis

[1] Jean-Yves Audibert, Rémi Munos, and Csaba Szepesvári. Tuning bandit algorithms in stochastic environments. In *International conference on algorithmic learning theory*, pages 150–165. Springer, 2007. https://inria.hal.science/inria-00203487.

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

[3] 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.

[4] Mersenne twister 19937 generator. https://cplusplus.com/reference/random/mt19937/. (besucht am 10.11.2023).

[5] Joseph Culberson. Sokoban is pspace-complete. Technical Report 97-02, University of Alberta, 1997. doi: https://doi.org/10.7939/R3JM23K33.

[6] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. doi: https://doi.org/10.5281/zenodo.592536.

[7] Bilal Kartal, Nick Sohre, and Stephen Guy. Data driven sokoban puzzle generation with monte carlo tree search. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 12, pages 58–64, 2016.

[8] Bilal Kartal, Nick Sohre, and Stephen Guy. Generating sokoban puzzle game levels with monte carlo tree search. Technical Report 16-005, University of Minnesota, 2016. https://hdl.handle.net/11299/215990.

[9] Sokoban. https://www.kenney.nl/assets/sokoban. (besucht am 10.11.2023).

[10] mallinfo(3) — linux manual page. https://man7.org/linux/man-pages/man3/mallinfo.3.html. (besucht am 10.11.2023).

[11] malloc(3) — linux manual page. https://man7.org/linux/man-pages/man3/malloc.3.html. (besucht am 10.11.2023).

[12] Joshua Taylor and Ian Parberry. Procedural generation of sokoban levels. In *Proceedings of the International North American Conference on Intelligent Games and Simulation*, pages 5–12, 2011.

[13] Julian Togelius, Georgios N Yannakakis, Kenneth O Stanley, and Cameron Browne. Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):172–186, 2011.

[14] Algorithms for calculating variance — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance. (besucht am 10.11.2023).