



ulm university universität  
**u**ulm

Fakultät für  
Informatik  
Institut für künstliche In-  
telligenz

# Automatisierung von Turtles in Minecraft mithilfe eines Numerischen Planners

Projektarbeit an der Universität Ulm

**Vorgelegt von:**

Michael Staněk  
Alexander Fischer  
[michael.stanek@uni-ulm.de](mailto:michael.stanek@uni-ulm.de)  
[alexander.fischer@uni-ulm.de](mailto:alexander.fischer@uni-ulm.de)

**Betreuer:**

M.Sc. Cornelia Olz

2022

Fassung 4. April 2022

© 2022 Michael Staněk  
Alexander Fischer

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/3.0/de/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.  
Satz: PDF-L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub>

# Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>1</b>
1.1 Problembeschreibung . . . . .	1
1.1.1 Minecraft . . . . .	2
1.1.2 Turtles . . . . .	4
1.1.3 Zielsetzung . . . . .	6
1.2 Verwandte Arbeiten . . . . .	8
<b>2 Grundlagen AI Planning</b>	<b>9</b>
2.1 Planning . . . . .	9
2.2 Modellierung . . . . .	10
2.3 Solving . . . . .	10
<b>3 Implementierung</b>	<b>12</b>
3.1 Architektur . . . . .	12
3.2 API Beschreibung . . . . .	12
3.3 Strategien . . . . .	15
3.3.1 Navigation . . . . .	16
3.3.2 Lager- und Inventarverwaltung . . . . .	16
3.3.3 Crafting . . . . .	17
3.3.4 Mining . . . . .	18
3.3.5 Sammeln . . . . .	18
<b>4 Das Planungsproblem</b>	<b>23</b>
4.1 Überblick . . . . .	23
4.2 Aktionen . . . . .	24
4.2.1 Aktionen zum Sammeln von Ressourcen . . . . .	24
4.2.2 Aktionen zum Craften . . . . .	26
4.2.3 Aktionen zum Schmelzen . . . . .	27
4.2.4 Andere Aktionen . . . . .	28
4.2.5 Probleminstanz . . . . .	29
4.2.6 Resultierender Plan . . . . .	31

*Inhaltsverzeichnis*

---

<b>5 Diskussion</b>	<b>33</b>
5.1 Planner Auswahl . . . . .	33
5.2 Fazit des Planungsproblems . . . . .	33
5.3 Wieso verwendet der Planner nicht einfach die Turtle-API? . . . . .	34
<b>A Anhang</b>	<b>36</b>
A.1 Die Domain . . . . .	36
<b>Literaturverzeichnis</b>	<b>43</b>

# 1 Einleitung

## 1.1 Problembeschreibung

Minecraft<sup>1</sup> ist ein 2009 erschienenes Computerspiel, in dem ein Spieler sich in einer dreidimensionalen Welt bewegen und diese verändern kann. Dabei gibt das Spiel keine fixen Ziele vor - was man genau erreichen will, bleibt vielmehr dem Spieler selbst überlassen. Der Spieler kann zwar durch diverse Aktionen sogenannte „Errungenschaften“ erreichen, z. B. indem er gewisse Gegenstände herstellt, Monster tötet, reist oder etwas baut. Viele Spieler vernachlässigen diese Errungenschaften jedoch und versuchen stattdessen selbst gesteckte Ziele zu erreichen. Solche Ziele könnten z. B. das Erkunden der Spielwelt, das Sammeln von Gegenständen<sup>2</sup>, das Herstellen von (oft teuren oder komplizierten) Gegenständen, das Bauen von subjektiv als schön befundenen Häusern und Strukturen (zum Beispiel einer Stadt auf dem Mars, siehe [3]) oder das Bauen von Strukturen zur Vereinfachung oder Automatisierung der Beschaffung von Rohstoffen, sogenannten „Farms“, sein.

Eine inoffizielle Modifikation namens „CC:Tweaked“<sup>3</sup> fügt dem Spiel sogenannte Turtles hinzu. Diese sind prinzipiell frei programmierbare Roboter und theoretisch bei passender Programmierung in der Lage, viele der genannten möglichen Ziele komplett autonom zu erreichen. Unser Ziel ist es nun in diesem Projekt, eine Programmstruktur zu schaffen, die es Turtles erlaubt, vorgegebene, komplexe Ziele selbstständig zu erreichen. Dabei soll die Programmstruktur nicht komplett problemspezifisch sein, sondern sich prinzipiell ohne große Änderungen mit unterschiedlichen Problemstellungen verwenden lassen.

Dabei wollen wir einen Planner verwenden, der die grobe Planung übernehmen soll, während kleinere Teilschritte in Minecraft direkt implementiert, also auf den Turtles programmiert werden. Somit dient das Projekt uns der Erfahrungssammlung in der praktischen Anwendung von Plannern. Darin findet sich auch eine Grundmotivation

---

<sup>1</sup><https://de.wikipedia.org/wiki/Minecraft#Spielwelt>.

<sup>2</sup><https://www.youtube.com/watch?v=dgha9S39Y6M>.

<sup>3</sup><https://tweaked.cc/>.



Abbildung 1.1: Eine Spielwelt in Minecraft. Man sieht einen Erdblock, der an einem Baum hängt, im Hintergrund einen aus Wasserblöcken bestehenden Fluss und Hügel, die aus diversen unterschiedlichen Blöcken bestehen.

für dieses Projekt: Minecraft bietet einerseits hinreichend viel Realitätsähnlichkeit, um anschaulich und verständlich zu sein, andererseits von Haus aus bereits einen hohen Abstraktionsgrad, sodass die Minecraftwelt selbst wiederum leicht modellierbar ist. Außerdem erlaubt die Existenz von Turtles einfach eine reale Umsetzung des Planes im Sinne einer Umsetzung des Planes in Minecraft.

Wir wollen zunächst die Spielwelt selbst beschreiben, bevor wir uns unserem Modell derselben widmen. Die Details unterscheiden sich in verschiedenen Versionen des Spiels leicht. Wir verwenden und beschreiben Version 1.16.5.

### 1.1.1 Minecraft

In Minecraft besteht die Welt aus würfelförmigen Blöcken, ist 256 Blöcke hoch und in die Horizontale für unsere Zwecke unendlich<sup>4</sup> groß. Diese Blöcke sind in einem fixen, ganzzahlbasiertem Koordinatensystem angeordnet - ihre Position ist also immer in 3 ganzzahligen Koordinaten gegeben, von denen die y-Koordinate zwischen 0 und 255 liegt. Eine Beispielumgebung findet sich in Abbildung 1.1. Der Spie-

<sup>4</sup>Bei normaler Laufgeschwindigkeit dauert es fast zwei Monate Echtzeit, den Rand der Welt zu erreichen.



Abbildung 1.2: Das Inventar eines Spielers. Die meisten der 9x4 Inventarslots sind leer, in einem befinden sich ein Stack mit 64 Cobblestone (Bruchstein), in einem anderen ein Stack mit 36 Dreckblöcken.

Ier kann sich in dieser Welt frei bewegen und Blöcke abbauen oder setzen. Wenn er einen Block abbaut, bekommt er ihn in der Regel als „Item“ in seinem Inventar (siehe Abbildung 1.2). Items in seinem Inventar kann er dann auf mehrere Arten verwenden, insbesondere kann er:

- Blöcke in die Spielwelt setzen.
- Items mit dem sogenannten Crafting-System kombinieren, um andere Items zu erhalten. Dafür muss er Items in einem 3x3 Gitter einem „Rezept“ genannten Muster entsprechend anordnen und verliert dabei in der Regel die verwendeten Items. Diesen Vorgang nennt man „Craften“.

Das Inventar eines Spielers besteht aus 36 sogenannten (Inventar-)slots. Ein Slot ist dabei ein Speicherplatz für einen sogenannten Stack von Items. Ein Stack von Items wiederum entspricht einem Item mit einer Quantität - zum Beispiel könnte ein Stack aus 5 Bruchsteinen oder 42 Goldäpfeln bestehen. Je nach Typ des Items haben unterschiedlich viele Items in einem Stack Platz - so haben zum Beispiel 64 Erdblocke in einem Stack Platz, jedoch nur drei Schilder. Hat ein Spieler mehr als 64 Erdblocke im Inventar, so verbrauchen diese entsprechend mindestens zwei Inventarslots. Manche Blöcke haben selbst wieder Inventare, mit denen der Spieler per Rechtsklick auf den Block interagieren kann. Ein Beispiel für solche Blöcke sind Kisten, die verwendet werden, um mehr Items als im Inventar Platz haben zu



Abbildung 1.3: Eine Turtle

speichern.

Das Spiel hat noch weitere Aspekte, die für unser Problem jedoch verhältnismäßig uninteressant sind und auf die wir aufgrund dessen nicht weiter eingehen werden.

### 1.1.2 Turtles

Turtles<sup>5</sup> sind prinzipiell Roboter, die eine inoffizielle Modifikation namens „CC:Tweaked<sup>6</sup>“ dem Spiel hinzufügt. Turtles sind im Spiel Blöcke (siehe Abbildung 1.3), die selbst ein 16 Slots großes Inventar besitzen. Der Spieler kann mit Turtles interagieren (siehe Abbildung 1.4) und Items aus seinem Inventar in das Inventar einer Turtle geben oder aus dem Inventar einer Turtle nehmen und in sein eigenes legen. Für den Zustand einer Turtle sind somit für uns relevant:

- Die Position, gegeben in 3 ganzzahligen Koordinaten (x, y und z) sowie die Rotation der Turtle, der sogenannten Blickrichtung. Diese kann die Werte Norden, Süden, Osten oder Westen, jedoch nicht oben oder unten haben.
- Das Inventar der Turtle, gegeben durch Itemtyp und Itemanzahl in jedem ihrer 16 Slots.

---

<sup>5</sup><https://www.computercraft.info/wiki/Turtle>.

<sup>6</sup><https://tweaked.cc/>.



Abbildung 1.4: Das Interface einer Turtle. Mithilfe einer Kommandozeile kann man Programme ausführen oder editieren. Man sieht links die 4x9 Inventarslots des Spielers und rechts davon die 4x4 Inventarslots der Turtle.

Außerdem kann der Spieler Turtles in der Programmiersprache Lua programmieren und auf Turtles gespeicherte Programme starten.

Zum Programmieren der Turtle stehen dem Spieler dabei diverse APIs zur Verfügung. Für uns ist vor allem die Turtle-API<sup>7</sup> interessant. Diese beinhaltet Methoden:

- Zur Bewegung: Werden `turtle.forward()`, `turtle.back()`, `turtle.up()` oder `turtle.down()` aufgerufen, versucht die Turtle, sich einen Block nach vorne, hinten, oben bzw. unten zu bewegen. Ist ein Block, ein Spieler oder Ähnliches im Weg und behindert die Turtle, bewegt sich die Turtle nicht und die Methode gibt `false` zurück. Ist sie hingegen nicht behindert, so bewegt sich die Turtle und die Methode gibt `true` zurück. Für Bewegung nach vorne bzw. hinten ist dabei die Blickrichtung der Turtle relevant - je nach Blickrichtung wird entweder die x- oder z-Koordinate um 1 inkrementiert oder dekrementiert. Bei der Bewegung nach oben bzw. unten wird immer die y-Koordinate um 1 inkrementiert bzw. dekrementiert.

Des Weiteren können die Methoden `turtle.turnLeft()` und `turtle.turnRight()` verwendet werden, um die Blickrichtung der Turtle zu verändern. Diese Rotieren jeweils die Turtle um 90 Grad gegen bzw. im Uhr-

---

<sup>7</sup>[https://www.computercraft.info/wiki/Turtle\\_\(API\)](https://www.computercraft.info/wiki/Turtle_(API)).

zeigersinn.

- Werden die Methoden `turtle.digDown()`, `turtle.dig()` oder `turtle.digUp()` aufgerufen, so baut die Turtle den Block unter, vor bzw. über sich ab und bekommt ihn als Item in ihr Inventar, insofern sie ihn dort aufnehmen kann. Kann sie ihn nicht aufnehmen, weil alle Inventarslots voll oder mit Items anderen Typs besetzt sind, so bekommt sie ihn nicht.
- Die Methoden `turtle.inspectDown()`, `turtle.inspect()` und `turtle.inspectUp()` untersuchen den Block unter, vor bzw. über der Turtle und geben eine Datenstruktur zurück, aus der man auslesen kann, was für ein Block an dieser Stelle steht.
- Methoden zur Manipulation des Inventars: Mithilfe von `turtle.select(number slotNum)`, `turtle.getSelectedSlot()`, `turtle.getItemCount()` und `turtle.transferTo(number slot, number quantity)` kann das Turtleeigene Inventar manipuliert werden. Außerdem kann die Turtle mithilfe der Methoden `turtle.dropDown()`, `turtle.drop()`, `turtle.dropUp()`, `turtle.suckDown()`, `turtle.suck()` und `turtle.suckUp()` mit benachbarten Kisten interagieren und Items aus dem eigenen Inventar mithilfe der drop-Methoden in den ersten freien Slot in deren Inventar verschieben oder mithilfe der suck-Methoden Items aus dem ersten besetzten Slot der Kisten in den ersten freien Slot des eigenen Inventars verschieben.
- Die Methode `turtle.craft()`, um zu craften. Die 16 Slots des Inventars der Turtle sind in einem 4x4 Feld angeordnet. Die oberen linken 3x3 Slots dienen dabei zugleich als Gitter zum Craften. Sind in diesen 9 Slots Items einem Rezept entsprechen angeordnet und wird `turtle.craft()` aufgerufen, so werden diese dem Rezept entsprechend in andere Items verwandelt.

Ein kleines Beispielprogramm, das mehrere dieser Methoden benutzt, findet sich in Abbildung 1.5.

### 1.1.3 Zielsetzung

Unter Verwendung dieser Methoden können Turtles prinzipiell auch sehr komplexe Probleme lösen - insbesondere kann eine Turtle selbstständig alle nötigen Schritte unternehmen, um eine zweite Turtle zu craften. Dafür muss sie:

- Rohstoffe sammeln: Sie muss Holz, Sand, Bruchstein, Eisenerz, Kohle, Diamanten und sogenannten Redstone sammeln. Während sie Eisenerz, Koh-



Abbildung 1.5: Ein Beispiel für ein einfaches Programm, das eine Turtle ausführen kann. Sie läuft 16 Blöcke vorwärts und baut dabei jeweils den Block vor sich und unter sich ab, wodurch ein 2 Blöcke hoher Tunnel entsteht.

Ie, Bruchstein, Diamanten und Redstone durch Suche unter der Erde finden kann, können Sand und Holz auf der Oberfläche gefunden werden.

- Aus Bruchstein einen Ofen craften und in der Welt platziieren.
- Unter Verbrennung von Kohle im Ofen Eisenerz zu Eisen, Bruchstein zu Stein und Sand zu Glas schmelzen.
- Durch wiederholtes Craften die Items zu höherwertigen Items kombinieren, bis sie schließlich eine zweite Turtle erhält.

Unser Ziel ist es nun, ein Programm zu schreiben, dem man ein Ziel übergibt und welches dieses Ziel dann erreicht. Dafür soll der Planner die grobe Planung übernehmen, während wir die einzelnen Schritte (craften, Rohstoffe suchen, schmelzen, Navigation) direkt in Lua implementieren. Ein konkretes Ziel, das wir so erreichen wollen, ist die Herstellung einer zweiten Turtle.

## **1.2 Verwandte Arbeiten**

Unserem Wissen nach gibt es bisher keine Arbeiten, die einen Planner nutzen, um beliebige Items von einer Turtle craften zu lassen.

In dem Paper [5] wird ein Planner genutzt, der einem menschlichen Nutzer einen Plan für die Konstruktion einer Struktur generieren soll. Sie benutzen hierfür unter anderem HTN planner, da die Strukturen wiederum aus kleineren Teilstrukturen bestehen (z. B. ein Haus mit mehreren identischen Wänden).

# 2 Grundlagen AI Planning

In diesem Kapitel wollen wir kurz einführen, was Planning grundsätzlich ist, wie man Planning-Probleme modellieren kann und wie sie gelöst werden.

## 2.1 Planning

Das Ziel von Planning ist, von einem Startzustand mittels einer Sequenz von Aktionen in einen Endzustand zu kommen. Das Planning-Problem besteht dann aus der Aufgabe, einen Plan zu finden, der genau dieses Ziel erreicht. Dabei bezeichnet der Plan ebenjene Sequenz von Aktionen. Die Aktionen, aus denen dieser Plan besteht, haben hierbei typischerweise einen Einfluss auf die Umwelt. Dabei können Aktionen andere Aktionen und deren Auswirkungen auf die Umwelt voraussetzen.

Aktionen bestehen hierbei aus den folgenden Komponenten:

1. **Vorbedingungen (auch Preconditions):** Diese müssen erfüllt sein, damit diese Aktion getätigigt werden kann.
2. **Effekte:** Die Auswirkungen dieser Aktion auf die Umwelt.

Praktisch besteht die Umwelt hierbei aus Variablen, die durch Effekte verschiedene Werte zugewiesen bekommen. Aktionen bedingen dann gewisse Werte dieser. Ein weiterer Aspekt von Planning kann es außerdem sein, einen gewisse Variablen minimierenden Plan zu finden. So könnte z. B. jede Aktion einen Effekt haben, der eine „Dauer“- Variable erhöht. Ziel wäre es dann, einen Plan zu finden, der so schnell wie möglich den Endzustand erreicht, in dessen Verlauf die „Dauer“-Variable also möglichst wenig erhöht wird.

## 2.2 Modellierung

Typischerweise werden die Domain und das konkrete Problem separat modelliert. Die Domain beschreibt hierbei die Umwelt, während das Problem Start- und Endzustand beschreibt.

Es gibt hierbei viele Sprachen, mit denen man diese Probleme modellieren kann. Eine dieser Sprachen ist PDDL [2] („Planning Domain Definition Language“). Seit deren Einführung ist sie zur Standardsprache für die Repräsentierung und den Austausch von Planning-Domains geworden [1].

PDDL wurde seit der Einführung viel erweitert. Für unsere Domain (siehe Kapitel 4) sind insbesondere sogenannte „numeric fluents“ relevant, die der Sprache mit PDDL 2.1 hinzugefügt wurden [1]. Mit diesen kann man auch Effekte auf numerischen Werten definieren und es möglich machen, dass Aktionen gewisse Werte dieser voraussetzen.

## 2.3 Solving

Wenn man nun die Planning Domain und das Problem definiert hat, ist es von Interesse, einen Plan zu generieren, der dieses Problem löst. Das ist die Aufgabe eines Planners. Dieser kann hierbei nach verschiedenen Algorithmen vorgehen, die wiederum unterschiedliche Eigenschaften haben (Optimalität, Laufzeit, Speicherverbrauch, etc.).

Ein einfacher Lösungsweg, der Optimalität garantiert, wäre zum Beispiel eine Breitensuche. Dabei wird ein Graph in Form eines (nicht zwangsläufig endlichen) Baumes generiert, bei dem jeder Knoten für einen Zustand steht sowie jede Kante für eine mögliche Aktion, die von einem Zustand in einen anderen führt. Die Blätter markieren dabei Zielzustände oder Zustände ohne mögliche weitere Aktionen, als Wurzel bezeichnet man den Startzustand. Mithilfe klassischer Breitensuche auf Bäumen kann man nun, sofern existent, einen Weg vom Startzustand zu einem Zielzustand finden. Dieser markiert dann einen optimalen Plan in jener Hinsicht, dass er die nötige Schrittanzahl minimiert.

In der Praxis ist die Verwendung klassische Breitensuche leider meist nicht möglich, da in vielen Domains die Laufzeit dieses Verfahrens zu hoch sein würde. Das ist zum Beispiel der Fall, wenn viele Aktionen zur Verfügung stehen und nur eine

sehr spezifische und lange Folge an Aktionen zum Endzustand führt. Dann müssen andere, ausgefeilte Suchverfahren, oft unter Verwendung von Metriken zur Kostenabschätzung verwendet werden.

Wir haben uns für unser Planungsproblem für die Verwendung des „Expressive Numeric Heuristic Search Planner“ (ENHSP) entschieden. Die von ihm verwendeten Algorithmen sind unter<sup>1</sup> zu finden. Eine Diskussion der Auswahl des Planners findet sich in Abschnitt 5.2.

---

<sup>1</sup><https://sites.google.com/view/enhsp/home/how-to-use-it>.

# 3 Implementierung

## 3.1 Architektur

Unser Projekt besteht aus mehreren Komponenten. Zum einen benutzen wir den ENHSP Planner<sup>1</sup> [4]. Dieser erstellt einen Plan für die von uns definierte Domain und das geschilderte Problem (siehe Kapitel 4). Dieser Plan listet alle Schritte auf, die von einer Turtle durchgeführt werden müssen, um das gegebene Problem zu lösen. Da dieser Plan noch nicht die Form von Lua-Methodenaufrufen hat, gibt es außerdem ein kleines, „Translator“ genanntes Programm. Dieses Programm übersetzt<sup>3</sup> den Output des Planners in ausführbaren Lua-Code. Dieser verwendet ausschließlich Methoden, die von unserer API bereitgestellt werden (siehe 3.2). Der Prozess ist abgebildet in Abbildung 3.1.

Auf der anderen Seite gibt es noch das Lua-Projekt. Dieses stellt Methoden sowie eine diese Methoden verwendende API bereit. Diese API kann dann vom übersetzten Plan aufgerufen werden. So kann der vom Planner erstellte Plan durch Aufrufen des übersetzten Programms in Minecraft ausgeführt werden.

Zusätzlich gibt es noch den sogenannten Program Spreader. Dieser verteilt Tests und Lua-Skripte auf die einzelnen Turtles. Das ist notwendig, da jede einzelne Turtle einen Ordner für sich selbst anlegt. Sie kann nur Programme ausführen, die in diesem Ordner sind. Auch der vom Translator generierte Plan wird mithilfe des Program Spreaders auf die Turtles verteilt.

## 3.2 API Beschreibung

Im Rest dieses Kapitels gehen wir auf die Implementierung der API ein, die der übersetzte Plan verwendet, um sich in Minecraft umzusetzen. Die Basis für die Im-

---

<sup>1</sup>[https://en.wikipedia.org/wiki/RAS\\_syndrome](https://en.wikipedia.org/wiki/RAS_syndrome).

<sup>2</sup><https://sites.google.com/view/enhsp/>.

<sup>3</sup>Wer hätte es gedacht.

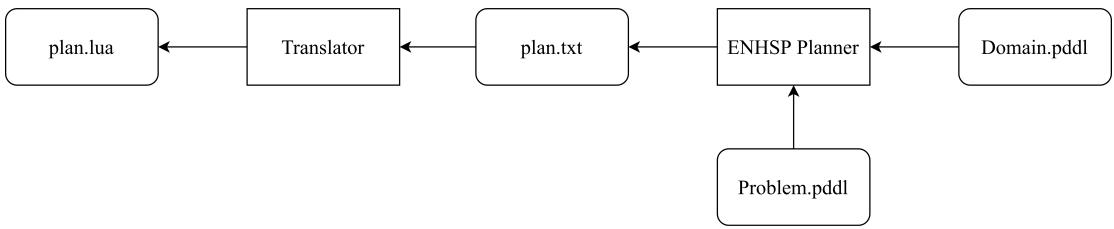


Abbildung 3.1: Schritte zur Generierung eines Plans

plementierung liefert die bereits in der Einführung beschriebene Turtle API<sup>4</sup>. Dort sind Methoden gelistet, mit denen man das Verhalten einer Turtle auf niedriger Ebene steuern kann. Diese Methoden sind jedoch viel zu feingliedrig für unseren Planner, der sich um die Beschaffung von komplexen Items und nicht um die Bewegung der Turtle kümmern soll. Deshalb (und aufgrund von weiteren, im Diskussionskapitel 5.3 beschriebenen Problemen mit nicht deterministischen Aktionen) stellen wir eine ausdrucksstärkere API bereit, die unser Planner nutzen kann. Diese ist komplett in Lua geschrieben und kann somit auf den Turtles ausgeführt werden.

Im Folgenden werden die einzelnen Methoden vorgestellt, die wir als API bereitstellen. Dies sind Methoden, die meist eine oder mehrere der in Abschnitt 3.3 definierten Strategien verwenden, um eine spezifische Funktion bereitzustellen. „Minen“ bedeutet im Folgenden das Sammeln von Ressourcen, die vorwiegend unterirdisch verfügbar sind. Hierfür muss eine andere Strategie verwendet werden als beim oberirdischen Sammeln, weshalb wir hier unterscheiden müssen.

#### **Initiate()**

`Initiate(args)` muss immer aufgerufen werden, sobald die Turtle startet. Die Methode initialisiert interne Konstanten und Variablen, die relevant für die Ausführung der verschiedenen Methoden auf der Turtle sind. Man kann `Initiate(args)` außerdem auch die Startkoordinaten und die Richtung der Turtle übergeben. Das ist besonders für die Höhe relevant, damit die Turtle weiß, wie sie auf die ideale Mininghöhe kommt. Übergibt man keine Argumente, dann basiert die Navigation nur auf der Entfernung zum Ursprung. In der Domäne und somit im unübersetzten Plan kommt `Initiate()` nicht vor; stattdessen beginnt der Übersetzer das Übersetzte Programm mit `Initiate()`. Die Höhe muss vor der Ausführung des Plans manuell in den Übersetzten Plan eingetragen werden. Alternativen dazu würden existieren, sind jedoch nicht allzu interessant, weshalb wir diese hier auslassen.

<sup>4</sup>[https://computercraft.info/wiki/Turtle\\_\(API\).](https://computercraft.info/wiki/Turtle_(API).)

### **InitiateChests()**

Diese Methode muss nach `Initiate()` aufgerufen werden. Sie initialisiert die interne Datenstruktur der Inventarverwaltung (In Abschnitt 3.3.2 beschrieben) und baut das zunächst nur aus drei Kisten bestehende Lager. `InitiateChests()` sucht hierbei nach 6 Holz der gleichen Art<sup>5</sup> und geht zum Ursprung zurück. Dann entfernt sie die restlichen Items aus dem Inventar und Craftet erst 24 Bretter und daraus dann 3 Kisten. Diese platziert sie dann in einem Halbkreis eine Block hoch über dem Ursprung. Analog zu `Initiate()` wird `InitiateChests()` vom Übersetzer an den Beginn des übersetzten Programms gestellt und wird in der Planungsdomäne nicht modelliert.

### **Mine(item,quantity)**

`Mine(item,quantity)` schickt die Turtle zum Minen (siehe Abschnitt 3.3.4) nach den festgelegten Ressourcen. Dabei werden auch anderweitig verwendbaren Rohstoffe abgebaut (Redstone, Eisenerz, Diamanten, Kohle und Cobblestone). Da der Planner nur statisch festlegen kann, dass er eine fixe Menge einer festgelegten Ressource abbaut, kann er nicht mit diesem Überschuss umgehen. Auch darum kümmert sich die `Mine(item,quantity)` Methode. Bei jedem Mining-Durchlauf werden die überschüssigen Ressourcen in einer Table gespeichert<sup>6</sup>. Geht die Turtle nun erneut minen, schaut sie erst in dieser Table, ob die angefragten Ressourcen bereits da sind. Wenn dies der Fall ist, werden diese aus der Table gelöscht und die Turtle geht nicht minen.

### **Gather(item, quantity)**

`Gather` schickt die Turtle zum Sammeln nach der angegebenen Menge von Holz oder Sand. Überschüssiger Sand und Bäume werden auch abgebaut und werden nach der gleichen Strategie behandelt wie bei `Mine(item,quantity)`.

---

<sup>5</sup>In Minecraft existieren unterschiedliche Holzarten. Man kann jede davon verwenden, um Kisten zu craften, und sie normalerweise auch beliebig kombinieren. Dies ist jedoch aufgrund der konkreten Implementierung des Craftingsystems bei Turtles nicht ohne die Verwendung von Kisten zum Zwischenspeichern von Blöcken möglich. Da jedoch per Definition noch keine Kisten existieren, wenn die Turtle die ersten Kisten baut, kann sie diese nicht verwenden und muss stattdessen 6 Holzblöcke derselben Holzart verwenden.

<sup>6</sup>Eine Table in Lua ist ein Key Value Store (Ähnlich einer Hashtable). Key ist in obigen Fall der Itemname als String. Der Value ist die Anzahl des entsprechenden Objekts.

### **PlaceFurnace()**

Holt sich einen Ofen aus dem Lager und platziert diesen an einer speziell für ihn reservierten Stelle.

### **Craft(itemname, itemcount)**

Craftet itemname itemcount mal. Dazu wird vorher das benötigte Material aus dem Lager geholt. Der Planner sorgt hierbei immer dafür, dass die Materialien vorher abgebaut wurden und im Lager bereitstehen. Dann ordnet die Turtle die Materialien entsprechend im Inventar an und craftet das Item. Die gecrafteten Items werden anschließend wieder in den Kisten gelagert.

### **Smelt(itemname, itemcount, fuelname, fuelcount)**

Smelt erlaubt das Schmelzen von Objekten. Dies wird benötigt, um aus Eisenerz Eisen, aus Sand Glas und aus Cobblestone Stein zu gewinnen. Dazu wird außerdem noch Treibstoff gebraucht. Das zu schmelzende Item und die Anzahl wird mit itemname und itemcount übergeben. Mit fuelname und fuelcount wird der Treibstoff übergeben, der für das Schmelzen verwendet werden sollen. Wenn die Methode aufgerufen wird, holt sich die Turtle das benötigte Material und läuft zum Ofen. Anschließend gibt sie die Items in den Ofen und wartet so lange vor dem Ofen, bis das Schmelzen abgeschlossen ist. Schließlich entnimmt sie das fertiggeschmolzene Item dem Ofen und lagert es ein.

## **3.3 Strategien**

In diesem Kapitel schauen wir uns ein paar ausgewählte Funktionalitäten näher im Detail an. Es wird vor allem auf Strategien eingegangen, die die Turtle verwendet, um konkrete Aufgaben zu erledigen. Diese Aufgaben werden als Teil der Aufrufe der Turtle API ausgeführt. Wir werden uns zum Beispiel anschauen, was für eine konkrete Strategie zum minen die Turtle verwendet.

### 3.3.1 Navigation

Die Turtle speichert intern ihre Koordinaten und ihre Blickrichtung basierend auf ihrer ursprünglichen Platzierung. Man kann ihr entweder die genauen Koordinaten und Richtung per Argument beim Start übergeben oder sie den gesetzten Defaultwert übernehmen lassen. Lediglich die Höhe ist hierbei wirklich relevant, damit die Turtle beim Mining auf die richtige Höhe findet. Bei Bewegung und Drehung werden diese internen Koordinaten und die Blickrichtung entsprechend geupdated. So kann die Turtle immer zu ihrem Ursprung zurückfinden. Dafür werden von dieser Klasse Methoden für die Bewegung bereitgestellt, die denen der Turtle API entsprechen, aber zusätzlich Standortinformationen updaten. Diese werden im restlichen Projekt für die meisten Bewegungen benutzt.

Methoden für die Navigation stellt die Klasse `movement.lua` bereit. Mit `go_towards(position)` lässt man die Turtle zu der angegebenen Position navigieren. Der Parameter `position` ist hierbei eine Table, die ganzzahlige Einträge für die x, die y und die z-Koordinate enthält.

Außerdem wird noch die Methode `navigate(position)` bereitgestellt, diese macht im Wesentlichen das gleiche wie die Methode `go_towards`, respektiert allerdings, dass im Startbereich Strukturen gebaut werden und sorgt so insbesondere dafür, dass Turtles bei der Fortbewegung niemals ihre Kisten zerstören.

API Methoden benutzen die Navigation so, dass die Turtle nach ihrem Aufruf wieder am Ursprungspunkt steht und Richtung Osten schaut.

### 3.3.2 Lager- und Inventarverwaltung

Eine kleine Schwierigkeit liegt in der Lagerung der Items. Da die Turtle beim Craften nichts außer den Items, die sie zum craften benötigt, im Inventar haben darf, muss sie ihre restlichen Items zwischenspeichern. Dafür eignen sich Chests (Kisten). Mithilfe von `turtle.drop()` kann sie Items in die Kiste geben, die dort im ersten freien Slot gespeichert werden. Mit `turtle.suck()` kann sie den ersten Stack Items aus der Kiste holen. Will die Turtle also Items aus dem  $n$ -ten Slot holen, so muss sie erst die ersten  $n$  Itemstacks aufsammeln und dann  $n-1$  davon in die Kiste zurückgeben. Insbesondere bedeutet das, dass sie für diese Operation mindestens  $n$  freie Inventarslots braucht.

Da ein Craftingrezept bis zu 9 unterschiedliche Items verwenden kann, bedeutet

dies, dass im Worst-Case-Szenario - das da wäre, dass die Turtle bereits 8 unterschiedliche, für das craften benötigte Items im Inventar hat und das letzte Item, das sie braucht, in einer Kiste einen Slot weit hinten belegt - die Turtle nur die ersten 8 Slots der Kisten sicher ohne weitere, größere Komplikationen verwenden kann. Deshalb haben wir uns entschieden, nur die ersten 8 Slots einer Kiste zu verwenden. Benötigt die Turtle mehr Platz, so muss sie weitere Kisten verwenden.

Um den Planner nicht zu überfordern, haben wir die Lagerverwaltung über Kisten direkt in Lua einprogrammiert - die Klasse `cheststoragesystem.lua` stellt entsprechende Methoden bereit, um Items in den Kisten zu Lagern, Items aus den Kisten zu holen und eine Inventur zu machen (alle Items in den Kisten zu zählen). Dabei speichert sie immer den aktuellen Zustand der Kisten in einer Datei, der wir anhand des Akronyms „**M**inecraft **I**nventory **C**heststoragesystem **H**elping **I**nterface“ den Namen „Chests.michi“ gegeben haben.

#### 3.3.3 Crafting

Mithilfe der Lager- und Inventarverwaltung kann die Turtle sich beliebige Items aus den Kisten holen. Wenn sie nun etwas craften will, schaut die Turtle sich das Rezept des Items an. Alle Crafting-Rezepte, die wir für das Projekt brauchen, sind in der Datei `recipes.lua` gespeichert. Wir beschränken uns auf jene Rezepte, die nötig für das Craften einer zweiten Turtle sind. Hat sie nun das Rezept nachgeschaut, holt sie die benötigten Items aus dem Lagersystem. Diese ordnet sie dann passend im Inventar an und craftet das entsprechende Item.

Hier wollen wir auch kurz darauf eingehen, dass wir die Crafting Rezepte etwas anpassen mussten, um die in Minecraft vorkommenden unterschiedlichen Holzarten zu unterstützen. In Minecraft gibt es viele verschiedene Holzarten. Relevant für das Projekt sind Eiche, Birke, Fichte, Tropenbaum, Akazie und Schwarzeiche. Das sind genau die Holzarten, die die Turtle beim Sammeln in der Welt finden kann. Die unterschiedlichen Stämme und Bretter, die sich daraus craften lassen, können in Crafting Rezepten, die Holz verwenden, beliebig kombiniert werden. Wenn man alle diese Kombinationen aus Rezepten einzeln einspeichert, benötigt man jedoch allein für Kisten  $6^8 = 1679616$  unterschiedliche Rezepte. Daher benutzen wir in Rezepten den Platzhalter `planks` für alle Bretterarten und den Platzhalter `woods` für alle Holzarten. Wird nun etwas mit den Materialien gecraftet, gibt es zusätzliche Logik, die auswählt, welche Holzarten dafür benutzt werden sollen. Diese werden dann wie üblich im zweiten Schritt aus dem Lager geholt. Außerdem gibt es noch das Crafting Rezept „`merged:planks`“. Dieses kann verwendet werden, um der Turt-

Ie zu sagen, dass sie einfach irgendwelche Bretter craften soll. Intern wird dann wieder geschaut, welche Stämme dafür konkret verwendet werden können.

### 3.3.4 Mining

Abbildung 3.2 zeigt die Verteilung der Rohstoffe auf die verschiedenen Höhen. Wir brauchen Diamanten, Redstone, Eisen, Kohle und Cobblestone (Bruchstein). Wir sehen, dass Redstone und Diamanten erst sehr weit unten zu finden sind, während Kohle und Eisen weniger restriktiv verteilt sind. Cobblestone ist auf jeder Ebene auffindbar. Daher sollten wir nur auf den Höhen mineen, auf denen auch Diamanten und Redstone zu finden ist. Diamanten und Redstone tauchen am häufigsten zwischen Ebene 5 - 12 auf<sup>7</sup>. Daher mineen wir auf diesen Ebenen.

Als Strategie könnte man nun einfach nach und nach die ganze Höhe aushöhlen und dabei alle Ressourcen mitnehmen. Wir haben uns aber für eine andere Strategie entschieden, bei der die Turtle deutlich weniger Blöcke abbauen muss, aber trotzdem jeden einzelnen Block überprüft.

Die Turtle kann mit `Turtle.inspect()` und verwandten Methoden jeweils die Blöcke vor sich (und damit mithilfe von Rotation auch links und rechts von sich), unter sich und über sich anschauen. Dies ist in Abbildung 3.3 mit den farbigen Blöcken gekennzeichnet. Das heißt, sie kann, wenn sie einen Tunnel gräbt, die Rohstoffe um sich in einem Kreuz Muster sehen. Wenn wir sie dann in einem Springermuster (siehe Position der Löcher in Abbildung 3.3) kleine Tunnel mit fixer Länge graben lassen, kann sie jeden Stein überprüfen. Wenn ein gesuchter Rohstoff dabei ist, wird er abgebaut.

Die Methode `function mine(goal)` implementiert dieses Verhalten. Zusätzlich überprüft sie, sobald sie zwei Tunnel gegraben hat, ob sie das definierte Ziel erreicht hat und bricht das Minen ab, wenn dem so ist. Sie speichert auch den Fortschritt, damit die Turtle beim nächsten Mineen einen neuen Tunnel gräbt.

### 3.3.5 Sammeln

Sammeln an der Oberfläche funktioniert etwas anders. Für unsere Zielstellung brauchen wir Sand und Bäume. Sand ist meist direkt Bestandteil der Oberfläche

<sup>7</sup>[https://minecraft.fandom.com/wiki/Ore/Pre-1.17\\_distribution](https://minecraft.fandom.com/wiki/Ore/Pre-1.17_distribution).

<sup>8</sup>[https://static.wikia.nocookie.net/minecraft\\_de\\_gamepedia/images/c/c0/1.18\\_Erzverteilung.jpg](https://static.wikia.nocookie.net/minecraft_de_gamepedia/images/c/c0/1.18_Erzverteilung.jpg).

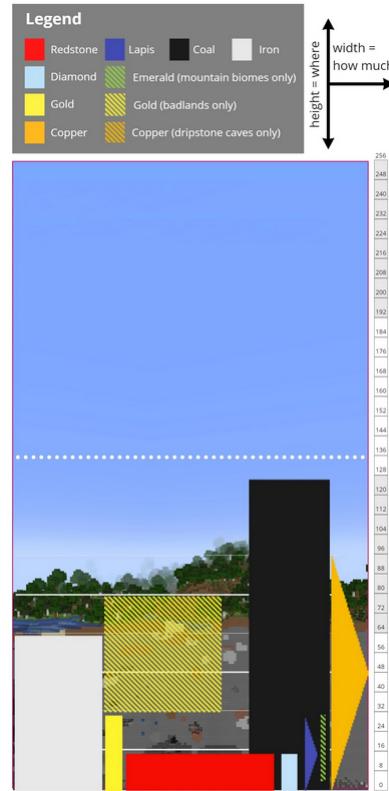


Abbildung 3.2: Minecraft Erzverteilung<sup>8</sup>

(siehe Abbildung 3.4), während Bäume auf der Oberfläche wachsen und meist einen Stamm haben, der eine Dicke von einem Block und eine Höhe von ungefähr sechs Blöcken hat (siehe Abbildung 3.5).

Wir brauchen also einen Algorithmus, der direkt an der Oberfläche entlang läuft und eine große Fläche absucht. Dabei ist die Form einer Spirale sinnvoll, da bei ihr wenig Zeit genutzt werden muss, um den letzten Endpunkt zu erreichen und die Fläche um die Basis gleichmäßig in alle Richtungen erkundet wird. Dieses Verhalten implementieren die Methoden `gather_wood(quantity, startup)` und `gather_ring(quantity, spiral, startup)`. Sobald unter einer Turtle Sand ist, wird so tief gegraben, bis kein Sand mehr gefunden wird. Trifft sie auf einen Baum,



Abbildung 3.3: Die farbigen Blöcke verdeutlichen das Sichtfeld einer Turtle. Die Löcher sind so angeordnet, dass alle Blöcke geprüft werden.

wird der Stamm angegraben<sup>910</sup> . Dann buddelt<sup>16</sup> die Turtle so weit nach oben, bis

<sup>9</sup>Das Wort „Fällen“ mag dem Minecraftunbewandten Leser hier zunächst adäquat vorkommen. Jedoch passt es an dieser Stelle nicht. Laut Definition des Dudens<sup>11</sup> bezeichnet Fällen „durch Hauen, Sägen o. Ä. zum Fallnen bringen; umschlagen; umhauen“. Jedoch kommt ein Baum in Minecraft nicht zu Fall, wenn man einen beliebigen, insbesondere auch den untersten Block abbaut. Stattdessen bleibt der Rest des Baumes einfach stehen und schwebt dann möglicherweise in der Luft; dies kann nicht als Fallverhalten angesehen werden. Viel passender ist hier somit die Verwendung des Wortes „Graben“. Dieses bezeichnet nach Definition des Dudens<sup>12</sup> „bohrend in etwas eindringen, sich in etwas bohren, wühlen, [hin]eingraben“ und beschreibt somit viel besser das von uns beobachtete Verhalten der Turtle. Hierbei kann zwar nicht von „bohrendem“ Verhalten gesprochen werden, da „bohren“ laut verschiedenen Definitionen des Dudens<sup>13</sup> „drehende Bewegungen“, einen „Bohrer“, „kreisförmige Bewegungen“ oder eine „Bohrmaschine“ voraussetzen würde. Jedoch kann durchaus von „wühlen“ gesprochen werden<sup>14</sup>, was im Duden<sup>15</sup> als „mit etwas (besonders den Händen, Pfoten, der Schnauze) in eine weiche, lockere Masse o. Ä. hineingreifen, eindringen und sie mit [kräftigen] schaufelnden Bewegungen aufwerfen, umwenden“ definiert ist. Die Ansicht, dass Holz in Minecraft „weich“ ist, ist weit verbreitet, da der Spieler Holz problemlos mit seinen Händen abbauen kann. Die Bewegung der Turtle wirkte auf uns subjektiv betrachtet durchaus als schaufelnd, die Animation des verschwindenden Holzblockes kann hingegen leicht als ein Aufwerfen von Holzstückchen gedeutet werden.

<sup>10</sup>Für die Fußnoten 9 und 11-17 übernimmt Michael Staněk die alleinige Verantwortung

<sup>11</sup><https://www.duden.de/rechtschreibung/faellen>, Definition 1.

<sup>12</sup><https://www.duden.de/rechtschreibung/graben>, Definition 4 b).

<sup>13</sup><https://www.duden.de/rechtschreibung/bohren>, Definitionen 1 a), 1 c), 1 d), 1 e), 2.

<sup>14</sup>Der aufmerksame Leser mag sich fragen, warum wir nicht einfach direkt das Wort „wühlen“ verwenden, wenn wir doch die Definition von „graben“ als „wühlen“ verwenden. Tatsächlich wenden wir jedoch den Teil „[hin]eingraben“ von Definition 4 b) an, um dann erst in der Definition von „graben“ im Wort „[hin]eingraben“ das Wort „wühlen“ durch erneute Anwendung von Definition

### 3 Implementierung

---

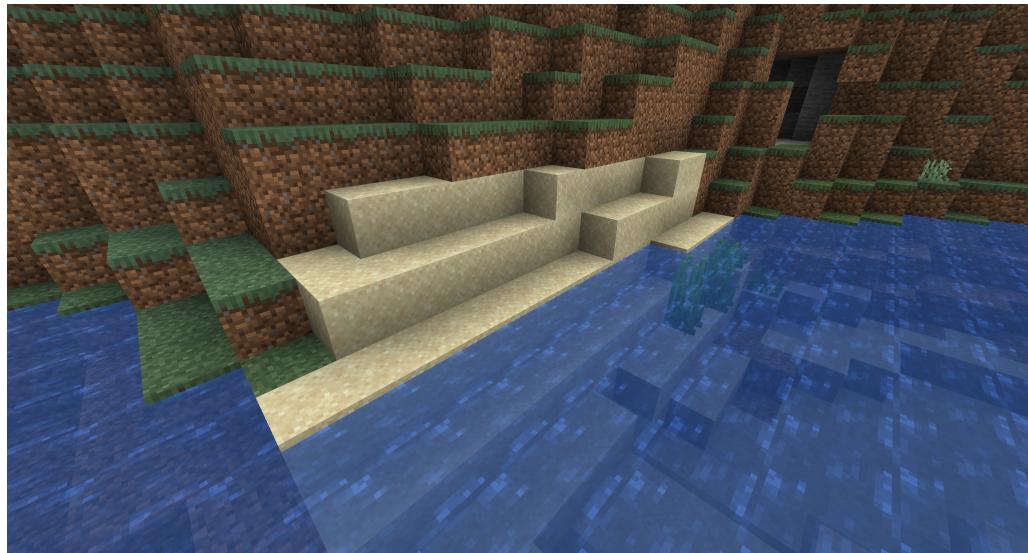


Abbildung 3.4: Sand auf der Oberfläche

kein Stamm mehr über ihr ist. Erreicht sie das in `quantity` definierte Ziel, speichert sie ihren Fortschritt und bricht das Sammeln ab. Beim nächsten Mal wird an dieser Stelle fortgesetzt.

---

4 b) einzusetzen. „hineinwühlen“ wäre zwar ebenfalls ein passender Wortstamm, jedoch würde damit der Satzbau deutlich komplizierter und länger werden. Solch unnötige und inhaltsleere Lesearbeit wollen wir Korrektoren und interessierten Lesern ersparen.

<sup>15</sup><https://www.duden.de/rechtschreibung/wuehlen>, Definitionen 1 a), 3 a).

<sup>16</sup>„graben; Erdarbeiten machen“, nach Definition des Dudens<sup>17</sup>, somit ebenfalls ein passender Begriff. Für eine Diskussion, warum hier der Begriff „graben“ passend ist, siehe Fußnote<sup>9</sup>.

<sup>17</sup><https://www.duden.de/rechtschreibung/buddeln>, Definition 1 a).



Abbildung 3.5: Ein Baum auf der Oberfläche

# 4 Das Planungsproblem

## 4.1 Überblick

In diesem Kapitel wollen wir die Domain definieren, die die Problemstellung „Verwende die in Kapitel 3 beschriebene API, um mithilfe einer Turtle eine weitere Turtle herzustellen“ konkretisiert. Dafür verwenden wir den PDDL-Standard, da dieser auch von dem von uns verwendeten Planner interpretiert werden kann.

Da Domäne und Problem separat gespeichert werden, kann dieselbe Domäne dann auch für andere konkrete Problemstellungen, z.B. „Baue 42 Turtles“ oder „Sammle 1234 Eisen, 2345 Diamanten und bau 3456 Computer“ unverändert verwendet werden. Nachdem wir verschiedene Alternativen ausprobiert haben, haben wir uns am Ende für folgende Modellierung entschieden:

- Typen: Es gibt den Typ `item` und davon den Untertypen `fuel`. Für jeden Typ von Item in Minecraft gibt es auch jeweils einen diesen Typ repräsentierenden Typ in der Domain. Zwecks sprachlicher Klarheit nennen wir Typen in Minecraft in diesem Kapitel Minecrafttypen und Typen in PDDL Problemtypen und assoziieren jeweils einen Minecrafttyp mit dem ihn repräsentierendem Problemtyp. Unterscheiden wir nicht zwischen diesen, nennen wir sie einfach Typen. Kann ein Minecrafttyp im Ofen verbrannt werden, so ist der entsprechende Problemtyp ein Untertyp von `fuel`, ansonsten von `item`.
- Funktionen:
  - Die Funktion (`fuelvalue ?f - fuel`) gibt bei einem Typ `?f`, der im Ofen als Brennstoff verwendet werden kann, an, wie viele Items dort geschmolzen werden können, wenn 1 Item vom Typ `?f` als Brennstoff verwendet wird.
  - Die Funktion (`count ?i - item`) gibt an, wie viele Items vom Typ `?i` die Turtle besitzt. Da die Lagerverwaltung über die Kisten auf Seiten der API implementiert wurde, spielt es dabei für den Planner keine Rolle,

ob die Items im Inventar der Turtle oder in einer Kiste liegen und ob sie auf mehrere Stacks aufgeteilt sind.

- Die Funktion (`turtleCount`) gibt an, wie viele Turtles momentan in der Welt existieren. Für die Problemstellung „Baue eine zweite Turtle“ ist diese Funktion irrelevant und konstant gleich 1. Sie kann jedoch bei anderen konkreten Problemstellungen verwendet werden, um dem Planner die Option zu geben, die Dauer der Ausführung des Plans in Minecraft zu verringern. Dafür können bei einer großen Aufgabe, zum Beispiel dem Sammeln einer sehr großen Anzahl von Ressourcen, erst weitere Turtles gebaut werden, die dann zusammen mit der ersten Turtle Ressourcen sammeln und so die erwartete Ausführungszeit des Plans zu verringern.
  - Die Funktion (`time`) dient als Heuristik für die Dauer der Ausführung des Plans und wird entsprechend bei Aktionen, deren Ausführung voraussichtlich in Minecraft verhältnismäßig lange dauert, erhöht.
  - Die Funktion (`furnaces`) gibt an, wie viele Öfen existieren. Sie muss mindestens 1 sein, damit Schmelzaktionen durchgeführt werden können. Ähnlich wie die Funktion (`turtleCount`) kann sie (bei leichter Veränderung der Schmelzaktionen) verwendet werden, um dem Planner die Option zu geben, mehrere Öfen zu bauen, dadurch Schmelzvorgänge zu beschleunigen und so eine schnellere erwartete Ausführungszeit des Plans in Minecraft zu erreichen.
- Aktionen, um die verschiedenen Aktionen, die die Turtle ausführen kann, zu modellieren. Es gibt:
    - Aktionen zum Sammeln von Ressourcen
    - Aktionen zum Craften
    - Aktionen zum Schmelzen

## 4.2 Aktionen

### 4.2.1 Aktionen zum Sammeln von Ressourcen

Für jeden Rohstoff, der in Minecraft in der Welt aufgesammelt werden muss, gibt es jeweils eine Aktion. Diese Sammelaktionen haben folgende Form:

```

1 (:action mine_diamond
2   :parameters ( ?d - diamond)
3   :precondition ()
4   :effect (and
5     (increase (time) 1)
6     (increase (count ?d) (turtleCount)))
7   )
8 )

```

Da die Ausführung von Rohstoffsammelaktionen in Minecraft viel länger braucht als die Ausführung von Schmelz- oder Craftaktionen, wird nur bei Rohstoffsammelaktionen der Zeitzähler in Zeile 5 erhöht. Indem der Rohstoffzähler nicht einfach um 1, sondern um turtleCount erhöht wird, hat der Planner theoretisch die Möglichkeit, erst mehr Turtles zu bauen und in die Welt zu setzen, um schneller eine große Rohstoffanzahl zu sammeln. Dabei modelliert Zeile 6 sehr gut den Fall, dass alle Turtles gleichzeitig auf der Suche nach dem Rohstoff, in diesem Beispiel einem Diamanten, sind.

Würde man die Zeilen 5 und 6 durch

```

1   (increase (time) (/ 1 (turtleCount)))
2   (increase (count ?d) 1)

```

ersetzen, so wäre der resultierende Plan etwas flexibler in der Ausführung auf Minecraftseite - sobald mehrere Turtles existieren, müssten diese nicht gleichzeitig dieselben Rohstoffe suchen, sondern könnten unterschiedliche Aktionen implementieren. Leider stürzte der von uns verwendete Planner bei dieser Variante jedoch ab. Unsere Vermutung ist, dass er Division mit einem sich während des Planens veränderndem Nenner nicht unterstützt.

Alternative Ideen zu der Verwendung der Funktion time waren die Verwendung von sogenannten Processes und Durative Actions. Jedoch garantiert der Planner bei der Verwendung von Processes keine Optimalität und erzeugte tatsächlich auch bei verhältnismäßig einfachen Problemstellungen in unserer Domain offensichtlich suboptimale Ergebnisse. Durative Actions andererseits werden (wie sich nach Kommunikation mit dem Entwickler ergeben hat) zwar geparsed, jedoch vom Planner selbst nicht implementiert, weshalb dieser abstürzt, wenn in der Domain Durative Actions vorkommen.

### 4.2.2 Aktionen zum Craften

Für jedes Rezept, das dem Planner zur Verfügung stehen soll, existiert in der Domain eine Aktion. Diese Craftingaktionen sind schematisch immer gleich aufgebaut. Für jeden Minecrafttyp, der für ein Craftingrezept benötigt wird, benötigt die dieses Craftingrezept repräsentierende Craftingaktion ein Objekt des entsprechenden Problemtyps als Parameter. Außerdem benötigt es ein Objekt des dem Produkt des Rezeptes entsprechendem Problemtyps.

In welchem Muster die Items angeordnet werden müssen, ist hier irrelevant, da wir diese Berechnung in der API bereits implementiert haben. Die Aktion muss also nur modellieren, dass die zum Craften benötigten Items verbraucht werden und dafür ein oder mehrere neue Items erzeugt werden.

Zum Beispiel benötigt das Rezept für das Craften eines Computers in Minecraft 7 Stein, 1 Redstone und 1 Glasscheibe (Glass Pane) und erzeugt dafür im Gegenzug einen Computer. Die entsprechende Craftingaktion sieht dann wie folgt aus:

```
1 (:action craft_computer
2   :parameters ( ?i1 - stone
3                 ?i2 - glass_pane
4                 ?i3 - redstone
5                 ?o - computer
6               )
7   :precondition (and
8     (>= (count ?i1) 7)
9     (>= (count ?i2) 1)
10    (>= (count ?i3) 1)
11  )
12   :effect (and
13     (decrease (count ?i1) 7)
14     (decrease (count ?i2) 1)
15     (decrease (count ?i3) 1)
16     (increase (count ?o) 1)
17   )
18 )
```

### 4.2.3 Aktionen zum Schmelzen

Schmelzaktionen funktionieren sehr ähnlich wie Craftingaktionen. Ein Unterschied ist jedoch, dass beim Schmelzen in Minecraft Treibstoff benötigt wird. Je nachdem, welcher Gegenstand als Treibstoff verwendet wird, können unterschiedlich viele Gegenstände geschmolzen werden – zum Beispiel kann ein Kohleitem verwendet werden, um bis zu 8 Items zu schmelzen. Theoretisch kann auch ein Kohleitem verwendet werden, um z. B. erst 4 Items von Typ a und dann 4 Items von Typ b zu schmelzen. Der Einfachheit halber haben wir diese Möglichkeit vernachlässigt. Die Precondition ( $\geq (\text{count } ?i1) (\text{fuelvalue } ?i2)$ ) sorgt dafür, dass so viele Items, wie es der Treibstoff erlaubt, geschmolzen werden. Die Precondition ( $\geq (\text{furnaces}) 1$ ) modelliert, dass nur dann geschmolzen werden kann, wenn tatsächlich auch bereits ein Ofen gebaut wurde. So sieht z. B. die Aktion zum Schmelzen von Eisen wie folgt aus:

```

1 (:action smelt_iron
2   :parameters ( ?i2 - fuel
3                 ?i1 - iron_ore
4                 ?o - iron_ingot
5               )
6   :precondition (and  (>= (count ?i1) (fuelvalue ?i2))
7                     (>= (count ?i2) 1)
8                     (>= (furnaces) 1)
9                   )
10  :effect (and
11      (decrease (count ?i1) (fuelvalue ?i2))
12      (decrease (count ?i2) 1)
13      (increase (count ?o) (fuelvalue ?i2))
14    )
15 )

```

Ähnlich dazu, wie (`turtleCount`) verwendet wird, um dem Planner die Option zu geben, mithilfe mehrere Turtles Sammelabläufe zu beschleunigen, könnte man auch hier mit kleinen Änderungen (`furnaces`) verwenden, um dem Planner zu erlauben, mithilfe mehrerer Öfen Schmelzabläufe zu beschleunigen. Dazu könnte man zum einen den Effekt

```

1 (increase (time) (/ 1 (furnaces)))

```

hinzugefügt werden, was jedoch der von uns verwendete Planner nicht akzeptieren würde (analog zu den Problemen bei den alternativen Implementierungen der Sammelaktionen). Alternativ könnte man ähnlich zu den Sammelaktionen verfahren, die resultierenden Schmelzaktionen hätten dann folgende Form:

```

1 (:action smelt_iron
2   :parameters ( ?i2 - fuel
3                 ?i1 - iron_ore
4                 ?o - iron_ingot
5                 )
6   :precondition (and (>= (count ?i1) (* (furnaces) (fuelvalue ?i2)))
7                   (>= (count ?i2) (furnaces))
8                   (>= (furnaces) 1)
9                   )
10  :effect (and
11      (decrease (count ?i1) (* (furnaces) (fuelvalue ?i2)))
12      (decrease (count ?i2) (furnaces))
13      (increase (count ?o) (* (furnaces) (fuelvalue ?i2)))
14      (increase (time) 1)
15      )
16  )

```

Dies machen wir jedoch nicht, da Schmelzaktionen viel schneller verlaufen als Sammelaktionen. Sinnvoll würde dies erst bei sehr großen Problemen werden, wenn z. B. Turtlecount über 100 wäre und entsprechend schnell Rohstoffe gesammelt werden würden. Da der Planner jedoch ohnehin nicht in der Lage ist, so große Probleme in absehbarer Zeit zu lösen, lassen wir diese Erweiterung weg.

#### 4.2.4 Andere Aktionen

Die Domain beinhaltet noch zwei weitere Aktionen: Eine, die das In-Die-Welt-Setzen einer bereits gecrafteten Turtle modelliert und eine, die das Bauen eines bereits gecrafteten Ofens modelliert. Da wir die Zähler (`turtleCount`) und (`furnaces`) für die Anzahl an Öfen bzw. Turtles in der Welt haben, können wir in den Aktionen diese jeweils einfach um 1 Inkrementieren. Die Aktion zum Ofen bauen beschreiben wir also durch

```

1 (:action build_furnace
2   :parameters ( ?t - furnace)
3   :precondition (>= (count ?t) 1)
4   :effect (and
5           (decrease (count ?t) 1)
6           (increase (furnaces) 1)
7         )

```

8

)

Die Aktion zum In-Die-Welt-Setzen einer Turtle funktioniert analog. Die ganze Domain befindet sich im Anhang A.1.

### 4.2.5 Probleminstanz

Mithilfe der bisher beschriebenen Infrastruktur (bestehend aus der Planungsdomäne, einem Planner, dem Übersetzer, der Minecraft-API und schlussendlich Turtles in Minecraft) können wir nun diverse Ziele automatisiert erreichen. Ist ein Gegenstand in der Spielwelt sammelbar im Sinne eines abbaubaren Blockes, so können wir ihn in prinzipiell beliebiger Vielfachheit sammeln (wobei bei größerer Anzahl natürlich die Laufzeit steigt). Alle Gegenstände, die sich durch wiederholtes Craften und Schmelzen aus sammelbaren Gegenständen im genannten Sinne herstellen lassen, können wir ebenfalls in beliebiger Vielfachheit herstellen. Die Bedingung dafür ist, dass alle Rezepte sowohl in der Minecraft-API als auch in der Planungsdomäne modelliert sind - wobei diese Modellierung auf beiden Seiten jeweils einem einfachen Muster folgt und bei allen Rezepten ohne größere Schwierigkeiten zu implementieren ist.

Wir haben uns das konkrete Ziel gesetzt, dass die Turtle eine zweite Turtle herstellen soll<sup>1</sup>. Wir beschreiben nun die dafür verwendete Probleminstanz. Probleminstanzen für andere Aufgaben lassen sich analog schreiben. In der PDDL-Problemdatei definieren wir zunächst für jeden Minecrafttyp ein Objekt vom entsprechenden Problemtyp. Dann initialisieren wir die Funktionen. Dabei setzen wir (`turtleCount`) auf 1, (`time`) und (`furnaces`) auf 0 und für jeden Objekttyp `?t` setzen wir (`count ?t`) auf 0 - somit steht zu Beginn eine Turtle zur Verfügung, die keine Items besitzt und auch noch keinen Ofen gebaut hat. (`fuelvalue coal`) setzen wir auf 8, da ein Kohleitem in Minecraft 8 Items schmelzen kann.<sup>4</sup>

---

<sup>1</sup>Womit die Grundvoraussetzungen für ein digitales Graue-Schmiere-Apokalypseszenario erfüllt sind.<sup>23</sup>

<sup>2</sup>[https://en.wikipedia.org/wiki/Gray\\_goo](https://en.wikipedia.org/wiki/Gray_goo).

<sup>3</sup><https://xkcd.com/1208/>.

<sup>4</sup>Wir könnten auch noch weitere Brennstoffe zum Schmelzen verwenden, z.B. Bretter, Türen oder Boote. Jedoch schmelzen viele davon keine ganzzahlige Anzahl von Items (z.B. schmelzen Bretter 1.5 Items), während wir für Boote oder Türen extra die Rezepte hinzufügen müssten, um diese craften zu können, obwohl wir Türen und Boote dann nur zum Verbrennen craften würden und sonst nirgends benötigen würden. Da der Planner sich schlussendlich ohnehin für Kohle entscheiden würde, aber seine Laufzeit durch die zusätzlichen, unnötigen Möglichkeiten steigen würde, haben wir diese weggelassen.

Als Ziel setzen wir, dass eine zweite Turtle gebaut werden soll, also dass `(count crafty_turtle) >= 1` gelten soll. Dabei soll die zu erwartende Zeit, die die Ausführung des Plans braucht, in Form der Funktion `(time)` minimiert werden. Wir erhalten so folgende Problemdatei:

```

1 (define (problem make_turtle)
2   (:domain minecraft)
3   (:objects
4     wood - wood
5     planks - planks
6     glass - glass
7     glass_pane - glass_pane
8     chest - chest
9     computer - computer
10    iron_ingot - iron_ingot
11    iron_ore - iron_ore
12    redstone - redstone
13    sand - sand
14    cobblestone - cobblestone
15    stone - stone
16    turtle - turtle
17    coal - coal
18    diamond - diamond
19    mining_turtle - mining_turtle
20    stick - stick
21    diamond_pickaxe - diamond_pickaxe
22    furnace - furnace
23    crafting_table - crafting_table
24    crafty_turtle - crafty_turtle
25  )
26
27  (:init
28    (= (furnaces) 0)
29    (= (time) 0)
30    (= (turtleCount) 1)
31    (= (count planks) 0)
32    (= (count wood) 0)
33    (= (count glass) 0)
34    (= (count glass_pane) 0)
35    (= (count chest) 0)
36    (= (count computer) 0)
37    (= (count iron_ingot) 0)
38    (= (count iron_ore) 0)
39    (= (count redstone) 0)
40    (= (count sand) 0)
41    (= (count cobblestone) 0)
42    (= (count stone) 0)
43    (= (count coal) 0)
44    (= (count turtle) 0)
45    (= (count diamond) 0)
46    (= (count mining_turtle) 0)
47    (= (count stick) 0)
48    (= (count diamond_pickaxe) 0)
49    (= (count furnace) 0)
50    (= (count crafty_turtle) 0)
51    (= (count crafting_table) 0)

```

```

52 (= (fuelvalue coal) 8)
53 )
54
55 (:goal
56   ( >= (count crafty_turtle) 1
57   )
58
59   (:metric minimize (time))
60 )

```

## 4.2.6 Resultierender Plan

Führen wir nun den Planner mit unserer Problemdatei und Domaindatei aus, so generiert er in etwa in einer halben Minute<sup>5</sup> folgenden Plan:

```

1 (mine_iron iron_ore)
2 (farm_sand sand)
3 (mine_cobblestone cobblestone)
4 (mine_iron iron_ore)
5 (farm_sand sand)
6 (mine_cobblestone cobblestone)
7 (mine_iron iron_ore)
8 (farm_sand sand)
9 (mine_cobblestone cobblestone)
10 (mine_iron iron_ore)
11 (farm_sand sand)
12 (mine_cobblestone cobblestone)
13 (mine_iron iron_ore)
14 (farm_sand sand)
15 (mine_cobblestone cobblestone)
16 (mine_iron iron_ore)
17 (farm_sand sand)
18 (mine_diamond diamond)
19 (mine_cobblestone cobblestone)
20 (farm_wood wood)
21 (farm_wood wood)
22 (craft_planks wood planks)
23 (craft_planks wood planks)
24 (craft_chest planks chest)
25 (mine_iron iron_ore)
26 (farm_sand sand)
27 (mine_diamond diamond)
28 (mine_cobblestone cobblestone)
29 (mine_cobblestone cobblestone)
30 (mine_cobblestone cobblestone)
31 (mine_cobblestone cobblestone)
32 (mine_cobblestone cobblestone)
33 (mine_cobblestone cobblestone)
34 (mine_cobblestone cobblestone)
35 (mine_cobblestone cobblestone)
36 (mine_cobblestone cobblestone)

```

---

<sup>5</sup>30 Sekunden bei Ausführung mit einem Intel®Core™i7-1185G7 - Prozessor.

#### 4 Das Planungsproblem

---

```
37 | (craft_furnace cobblestone furnace)
38 | (build_furnace furnace)
39 | (mine_cobblestone cobblestone)
40 | (mine_coal coal)
41 | (smelt_stone coal cobblestone stone)
42 | (mine_coal coal)
43 | (farm_sand sand)
44 | (smelt_glass coal sand glass)
45 | (craft_glass_pane glass glass_pane)
46 | (mine_redstone redstone)
47 | (craft_computer stone glass_pane redstone computer)
48 | (mine_coal coal)
49 | (mine_iron iron_ore)
50 | (smelt_iron coal iron_ore iron_ingot)
51 | (craft_turtle iron_ingot computer chest turtle)
52 | (farm_wood wood)
53 | (craft_planks wood planks)
54 | (craft_stick planks stick)
55 | (mine_diamond diamond)
56 | (craft_diamond_pickaxe diamond stick diamond_pickaxe)
57 | (craft_mining_turtle turtle diamond_pickaxe mining_turtle)
58 | (farm_wood wood)
59 | (craft_planks wood planks)
60 | (craft_crafting_table planks crafting_table)
61 | (craft_crafty_turtle mining_turtle crafting_table crafty_turtle)
```

Dieser kann nun verwendet werden, um den Lua-Code für die Erstellung einer Turtle zu generieren.

# 5 Diskussion

## 5.1 Planner Auswahl

Wir haben uns für den ENHSP Planner entschieden. Wir konnten nur drei Planner finden, welche numerical fluents unterstützen, was für unser Problem jedoch notwendig ist<sup>1</sup>. Von den dreien war der ENHSP für uns am geeignetsten, da er in der Java Virtual Machine läuft und somit plattformunabhängig ist. Die anderen beiden Planner gab es leider nur in unkomplizierter Form (C++ Code). Der Kompilierungsprozess schlug leider trotz sorgfältigem Folgen der bereitgestellten Dokumentation fehl. Die resultierenden Fehler ließen sich leider trotz Aufwendung einiger Anstrengungen nicht beheben. Da wir mit dem ENHSP einen funktionierenden Planner haben, der auch die von uns benötigten Features implementiert hat, entschieden wir uns einfach diesen zu benutzen.

## 5.2 Fazit des Planungsproblems

Das Ziel, mithilfe des Planners die Turtle der Vervielfachung zu bemächtigen, wurde erreicht. Außerdem wird das dazu gehörende Planungsproblem auf gängiger<sup>2</sup> Hardware in vernünftiger Zeit<sup>3</sup> gelöst. Die Idee, dem Planner zu ermöglichen, durch die Konstruktion zusätzlicher Turtles die Möglichkeit zu geben, die erwartete Ausführungszeit des Plans zu verringern, wurde semierfolgreich implementiert. Die Domäne erlaubt dies dem Planer zwar, und aufgrund der Verwendung von (time) als Metrik wird auch versucht, die erwartete Ausführungszeit zu minimieren. Jedoch haben einfache Versuche ergeben, dass der Planner mit Standardoptionen suboptimale Resultate liefert (s. u.). Wurden hingegen Optionen verwendet, die laut der Website<sup>4</sup> des Planers Optimalität Garantieren, so terminierten diese auf unseren

---

<sup>1</sup><https://planning.wiki/ref/planners/tags/numeric>.

<sup>2</sup>unserer.

<sup>3</sup>In diesem Kontext: Unter einer Minute.

<sup>4</sup><https://sites.google.com/view/enhsp/home/how-to-use-it>.

Rechnern nicht erfolgreich, sondern stürzten mit einer OutOfMemoryException ab - sie benötigten also zu viel Speicher.

Für einen hier beispielhaft genannten Versuch definierten wir Domäne und Problemstellung wie in Kapitel 4 beschrieben, bis darauf, dass wir das Ziel in der Problemdatei ersetzen durch

```
1 (:goal
2   (and
3     (>= (turtleCount) 2)
4     (>= (count diamond) 100)
5   )
6 )
```

Eine mögliche Lösung wäre es, zuerst eine zweite Turtle zu konstruieren, was (mit dem Plan aus Kapitel 4) 43 Zeiteinheiten dauert. Anschließend könnte man mit beiden Turtles in 50 Zeiteinheiten 100 Diamanten sammeln. Für diesen Plan würde somit die als Metrik verwendete Zählvariable (`time`) bei 93 liegen.

Der ausgegebene Plan des Planners sah jedoch stattdessen vor, zuerst mit einer Turtle in 98 Zeiteinheiten 98 Diamanten zu sammeln, dann in 43 Zeiteinheiten eine zweite Turtle zu konstruieren und anschließen in einer Zeiteinheit die letzten beiden Diamanten zu sammeln - womit die (`time`) für diesen Plan bei 142 liegt. Dies zeigt<sup>5</sup>, das der Planner bei Standardoptionen bei unserer Domain bereits suboptimale Resultate liefert.

## 5.3 Wieso verwendet der Planner nicht einfach die Turtle-API?

Prinzipiell hätten wir anstatt eine API in Minecraft zu schreiben, auch einfach die bereits vorhandenen Methoden der Turtle-API direkt verwenden können. Der offensichtliche Nachteil davon wäre, dass damit die Planungsdomäne um ein Vielfaches komplexer werden würde und wohl kaum noch ein Planner in der Lage wäre, das Problem auf gängiger<sup>6</sup> Hardware in vernünftiger Zeit<sup>7</sup> zu lösen. Eine weitere, etwas weniger offensichtliche Problematik liegt in der Determiniertheit von Aktionen.

---

<sup>5</sup>Beweis durch Beispiel.

<sup>6</sup>unserer.

<sup>7</sup>In diesem Kontext: Unter einer Minute.

## 5 Diskussion

---

Prinzipiell ist Ressourcenbeschaffung in Minecraft eine undeterministische Angelegenheit: Rohstoffe sind nach einer bekannten Verteilung zufällig unterirdisch oder auf der Oberfläche in der Spielwelt verteilt. Somit gibt es ohne weitere Kenntnis über die konkrete Umgebung einer Turtle zunächst keine vordefinierte Sequenz von Aktionen, die das Auffinden eines Rohstoffes garantiert. Aufgrunddessen ist eine Modellierung von Problemen der Art „Sammle Rohstoffe“ nur mithilfe der von der Turtle-API bereitgestellten Methoden sehr schwierig zu bewerkstelligen, da man hierfür undeterministische Aktionen benötigen würde. Unsere Methoden zur Rohstoffgewinnung folgen jedoch alle dem Muster

```
1 WHILE nicht alle Rohstoffe gesammelt DO
2     verwende eine Sinnvolle Strategie,
3     um Rohstoffe zu sammeln
4 END
```

Aufgrund der WHILE-Schleife ist diese Aktion nun deterministisch: Dass sie irgendwann terminiert ist durch die konkreten Regeln zur Rohstoffverteilung in Minecraft gegeben, und die Wahrscheinlichkeit, dass dies nicht in vernünftiger Zeit<sup>8</sup> passiert, ist bei allen implementierten Sammelaktionen vernachlässigbar gering.

Die von uns implementierten Sammelaktionen sind somit deterministisch in der Hinsicht, dass sie garantiert die erwünschten Rohstoffe beschaffen. Ihre Ausführungszeit ist zwar nicht deterministisch, jedoch reicht es für uns aus, sie mithilfe der statistischen Ausführungszeit oder auch nur einfach mit einer Konstante abzuschätzen.

---

<sup>8</sup>In diesem Kontext: Unter einem Tag.

# A Anhang

## A.1 Die Domain

```
1 (define
2   (domain minecraft)
3
4   (:types
5     item - object
6     wood - item
7     planks - item
8     glass - item
9     glass_pane - item
10    chest - item
11    computer - item
12    iron_ingot - item
13    iron_ore - item
14    redstone - item
15    sand - item
16    cobblestone - item
17    stone - item
18    turtle - item
19    fuel - item
20    diamond - item
21    mining_turtle - item
22    stick - item
23    diamond_pickaxe - item
24    furnace - item
25    coal - fuel
26    crafty_turtle - item
27    crafting_table - item)
28
29
30   (:functions
31     (fuelvalue ?f - fuel)
32     (count ?i - item)
33     (turtleCount)
34     (time)
35     (furnaces)
36   )
37
38   (:constraint positive_amounts
39     :parameters(? i - item)
40     :condition (>= (count ?i) 0)
41   )
42 )
```

```

43
44
45
46
47    ;; GATHERING / MINING / FARMING ACTIONS ;;;;;;;;;;;;;
48
49    (:action farm_wood
50        :parameters (    ?w - wood)
51        :precondition ()
52        :effect (and
53            (increase (time) 1); (/ 1 (turtleCount)))
54            (increase (count ?w) (turtleCount)); (* 1 (turtleCount)))
55        )
56    )
57
58
59    (:action farm_sand
60        :parameters (    ?s - sand)
61        :precondition ()
62        :effect (and
63            (increase (time) 1); (/ 1 (turtleCount)))
64            (increase (count ?s) (turtleCount)); (* 1 (turtleCount)))
65        )
66    )
67
68    (:action mine_coal
69        :parameters (    ?cl - coal)
70        :precondition ()
71        :effect (and
72            (increase (time) 1); (/ 1 (turtleCount)))
73            (increase (count ?cl) (turtleCount)); (* 1 (turtleCount)))
74        )
75    )
76
77    (:action mine_cobblestone
78        :parameters (    ?cs - cobblestone)
79        :precondition ()
80        :effect (and
81            (increase (time) 1); (/ 1 (turtleCount)))
82            (increase (count ?cs) (turtleCount)); (* 1 (turtleCount)))
83        )
84    )
85
86    (:action mine_redstone
87        :parameters (    ?r - redstone)
88        :precondition ()
89        :effect (and
90            (increase (time) 1); (/ 1 (turtleCount)))
91            (increase (count ?r) (turtleCount)); (* 1 (turtleCount)))
92        )
93    )
94
95    (:action mine_diamond
96        :parameters (    ?d - diamond)
97        :precondition ()
98        :effect (and
99            (increase (time) 1); (/ 1 (turtleCount)))

```

```

100           (increase (count ?d) 1); (turtleCount)); (* 1 (turtleCount)))
101         )
102       )
103
104
105   (:action mine_iron
106     :parameters ( ?i - iron_ore)
107     :precondition ()
108     :effect (and
109       (increase (time) 1); (/ 1 (turtleCount)))
110       (increase (count ?i) (turtleCount)); (* 1 (turtleCount)))
111     )
112   )
113
114
115
116
117   ;;OTHER;;;;;;;;;;;;;;;
118
119   (:action deploy_turtle
120     :parameters ( ?t - mining_turtle)
121     :precondition (>= (count ?t) 1)
122     :effect (and
123       (decrease (count ?t) 1)
124       (increase (turtleCount) 1)
125     )
126   )
127
128
129   (:action build_furnace
130     :parameters ( ?t - furnace)
131     :precondition (>= (count ?t) 1)
132     :effect (and
133       (decrease (count ?t) 1)
134       (increase (furnaces) 1)
135     )
136   )
137
138
139   ;;CRAFTING RECIPES ;;;;;;;;;;;;;;;;;
140
141
142   (:action craft_planks
143     :parameters ( ?w - wood
144                   ?p - planks
145                   )
146     :precondition (>= (count ?w) 1)
147     :effect (and
148       (decrease (count ?w) 1)
149       (increase (count ?p) 4)
150     )
151   )
152
153   (:action craft_chest
154     :parameters ( ?i1 - planks
155                   ?o - chest
156                   )

```

```

157      :precondition (>= (count ?i1) 8)
158      :effect (and
159          (decrease (count ?i1) 8)
160          (increase (count ?o) 1)
161      )
162  )
163
164  (:action craft_glass_pane
165      :parameters (    ?i1 - glass
166                      ?o - glass_pane
167                      )
168      :precondition (>= (count ?i1) 6)
169      :effect (and
170          (decrease (count ?i1) 6)
171          (increase (count ?o) 16)
172      )
173  )
174
175  (:action craft_computer
176      :parameters (    ?i1 - stone
177                      ?i2 - glass_pane
178                      ?i3 - redstone
179                      ?o - computer
180                      )
181      :precondition (and
182          (>= (count ?i1) 7)
183          (>= (count ?i2) 1)
184          (>= (count ?i3) 1)
185          )
186      :effect (and
187          (decrease (count ?i1) 7)
188          (decrease (count ?i2) 1)
189          (decrease (count ?i3) 1)
190          (increase (count ?o) 1)
191      )
192  )
193
194
195  (:action craft_turtle
196      :parameters (    ?i1 - iron_ingot
197                      ?i2 - computer
198                      ?i3 - chest
199                      ?o - turtle
200                      )
201      :precondition (and
202          (>= (count ?i1) 7)
203          (>= (count ?i2) 1)
204          (>= (count ?i3) 1)
205          )
206      :effect (and
207          (decrease (count ?i1) 7)
208          (decrease (count ?i2) 1)
209          (decrease (count ?i3) 1)
210          (increase (count ?o) 1)
211      )
212  )
213

```

```

214
215
216 (:action craft_mining_turtle
217   :parameters ( ?i1 - turtle
218                 ?i2 - diamond_pickaxe
219                 ?o - mining_turtle
220                 )
221   :precondition (and
222                 (>= (count ?i1) 1)
223                 (>= (count ?i2) 1)
224                 )
225   :effect (and
226             (decrease (count ?i1) 1)
227             (decrease (count ?i2) 1)
228             (increase (count ?o) 1)
229             )
230   )
231
232
233
234 (:action craft_diamond_pickaxe
235   :parameters ( ?i1 - diamond
236                 ?i2 - stick
237                 ?o - diamond_pickaxe
238                 )
239   :precondition (and
240             (>= (count ?i1) 3)
241             (>= (count ?i2) 2)
242             )
243   :effect (and
244             (decrease (count ?i1) 3)
245             (decrease (count ?i2) 2)
246             (increase (count ?o) 1)
247             )
248   )
249
250 (:action craft_crafty_turtle
251   :parameters ( ?i1 - mining_turtle
252                 ?i2 - crafting_table
253                 ?o - crafty_turtle
254                 )
255   :precondition (and
256             (>= (count ?i1) 1)
257             (>= (count ?i2) 1)
258             )
259   :effect (and
260             (decrease (count ?i1) 1)
261             (decrease (count ?i2) 1)
262             (increase (count ?o) 1)
263             )
264   )
265
266
267 (:action craft_stick
268   :parameters ( ?i1 - planks
269                 ?o - stick
270                 )

```

```

271      ;;;duration ( = ?duration 1 )
272      :precondition ( and
273          (>= (count ?i1) 2)
274          )
275      :effect (and
276          (decrease (count ?i1) 2)
277          (increase (count ?o) 4)
278          )
279      )
280
281      (:action craft_crafting_table
282          :parameters ( ?i1 - planks
283                          ?o - crafting_table
284                          )
285          ;;;duration ( = ?duration 1 )
286          :precondition ( and
287              (>= (count ?i1) 4)
288              )
289          :effect (and
290              (decrease (count ?i1) 4)
291              (increase (count ?o) 1)
292              )
293          )
294
295      (:action craft_furnace
296          :parameters ( ?i1 - cobblestone
297                          ?o - furnace
298                          )
299          ;;;duration ( = ?duration 1 )
300          :precondition ( and
301              (>= (count ?i1) 8)
302              )
303          :effect (and
304              (decrease (count ?i1) 8)
305              (increase (count ?o) 1)
306              )
307          )
308
309
310
311
312
313      ;;;SMELTING RECIPES ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
314
315
316      (:action smelt_iron
317          :parameters ( ?i2 - fuel
318                          ?i1 - iron_ore
319                          ?o - iron_ingot
320                          )
321          :precondition (and (>= (count ?i1) (fuelvalue ?i2))
322                          (>= (count ?i2) 1)
323                          (>= (furnaces) 1)
324                          )
325          :effect (and
326              (decrease (count ?i1) (fuelvalue ?i2))
327              (decrease (count ?i2) 1)

```

```
328         (increase (count ?o) (fuelvalue ?i2))
329     )
330 )
331 (:action smelt_glass
332   :parameters (  ?i2 - fuel
333                 ?i1 - sand
334                 ?o - glass
335                 )
336   :precondition (and  (>= (count ?i1) (fuelvalue ?i2))
337                     (>= (count ?i2) 1)
338                     (>= (furnaces) 1)
339                     )
340   :effect (and
341             (decrease (count ?i1) (fuelvalue ?i2))
342             (decrease (count ?i2) 1)
343             (increase (count ?o) (fuelvalue ?i2)))
344             )
345           )
346 )
347
348
349 (:action smelt_stone
350   :parameters (  ?i2 - fuel
351                 ?i1 - cobblestone
352                 ?o - stone
353                 )
354   :precondition (and  (>= (count ?i1) (fuelvalue ?i2))
355                     (>= (count ?i2) 1)
356                     (>= (furnaces) 1)
357                     )
358   :effect (and
359             (decrease (count ?i1) (fuelvalue ?i2))
360             (decrease (count ?i2) 1)
361             (increase (count ?o) (fuelvalue ?i2)))
362             )
363           )
364 )
```

# Literaturverzeichnis

- [1] Fox, M. ; LONG, D. : PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. In: J. Artif. Intell. Res. 20 (2003), 61–124. <http://dx.doi.org/10.1613/jair.1129>. – DOI 10.1613/jair.1129
- [2] GHALLAB, M. ; KNOBLOCK, C. ; WILKINS, D. ; BARRETT, A. ; CHRISTIANSON, D. ; FRIEDMAN, M. ; KWOK, C. ; GOLDEN, K. ; PENBERTHY, S. ; SMITH, D. ; SUN, Y. ; WELD, D. : PDDL - The Planning Domain Definition Language. (1998), 08
- [3] NOLAN, A. ; NOLAN, E. ; NOLAN, R. ; NOLAN, F. : Modelling a Meta Smart City on Mars in Minecraft. In: 2021 World Automation Congress, WAC 2021, Taipei, Taiwan, August 1-5, 2021, IEEE, 222–227
- [4] SCALA, E. ; HASLUM, P. ; THIÉBAUX, S. ; RAMÍREZ, M. : Interval-Based Relaxation for General Numeric Planning. In: ECAI, 2016
- [5] WICHLACZ, J. ; TORRALBA, A. ; HOFFMANN, J. : Construction-Planning Models in Minecraft. <http://dx.doi.org/10.5281/zenodo.3239243>. Version: Jun. 2019