**InformatiCup 2023**

# Profit!

von

**Lisa Binkert, Leopold Gaube und Yasin Koschinski**

RWU Hochschule Ravensburg-Weingarten

Abgabedatum: 15.01.2023

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# 1 Einleitung

Der InformatiCup ist ein Wettbewerb, der von der Gesellschaft für Informatik jährlich veranstaltet wird. Studierende aller Fachrichtungen an Universitäten und Hochschulen in Deutschland, Österreich und der Schweiz dürfen daran teilnehmen.

Dieses Dokument beschreibt die Lösung für den InformatiCup 2023 des Teams “Die Schmetterlinge”. Das Team besteht aus Lisa Binkert, Leopold Gaube und Yasin Koschinski, welche alle an der RWU Hochschule Ravensburg-Weingarten im Studiengang Master Informatik studieren.

Die Aufgabe des InformatiCups 2023 ist das Lösen des Spiels Profit. Die Aufgabenstellung wird im Kapitel 2 genauer erläutert. In Kapitel 3 werden verschiedene Lösungsansätze beschrieben, von denen nur ein Teil in die endgültige Lösung eingeflossen ist (Kapitel 4). Die Anwendung der Lösung wird in Kapitel 5, dem Benutzerhandbuch, beschrieben. In Kapitel 6 wird die Lösung evaluiert und bewertet, worauf in Kapitel 7 ein Fazit folgt.
Der Code der Lösung kann ebenfalls in dem Github-Respository des Projekts angeschaut werden.
Das kann über den Link: https://github.com/tarexo/informaticup-profit erreicht werden.

# 2 Aufgabenbeschreibung

Die Aufgabe des InformatiCups 2023 ist das Lösen und Optimieren des Spiels “Profit!”. Das Spiel simuliert rundenbasierte Prozesse, in denen durch das Platzieren von verschiedenen Gebäude Ressourcen abgebaut und Produkte erstellt werden können. Das Herstellen von Produkten wird mit Punkten belohnt. Das Ziel des Spiels ist es die Punkte zu maximieren. Dies sollte möglichst effizient, also mit minimaler Rundenanzahl, erreicht werden.

In den folgenden Abschnitten werden die Regeln und der Ablauf des Spiels sowie die Codierung des Spiels im JSON-Format kurz erläutert.

## 2.1 Spielregeln

Das Spielfeld besteht aus einem maximal $100 \times 100$ großen Rasterfeld. Ein Feld ist entweder leer oder durch ein Objekt besetzt. Das linke obere Feld befindet sich an der Stelle (0,0), das rechte untere an (Breite-1, Höhe-1). Die Größe des Spielfeldes ist für jede Aufgabe vorgegeben. Gebäude können Ein- und Ausgänge besitzen, über die Ressourcen an angrenzende Gebäude weitergegeben werden. An einen Ausgang darf maximal ein Eingang horizontal oder vertikal angrenzen. An einen Eingang dürfen jedoch mehrere Ausgänge anliegen, was bewirkt, dass Ressourceflüsse zusammengeführt werden.
Zu Beginn des Spiels sind bereits Hindernisse und Lagerstätten mit Ressourcen vorhanden. Ebenso werden die maximale Rundenanzahl und die Produkte, die produziert werden können, festgelegt.

Die Abbildung 1 zeigt ein mögliches Spielfeld. In diesem Beispiel ist das Feld $30 \times 20$ groß und enthält drei Lagerstätten mit den Ressourcen Subtyp 0, 1 und 2, sowie zwei Hindernisse.

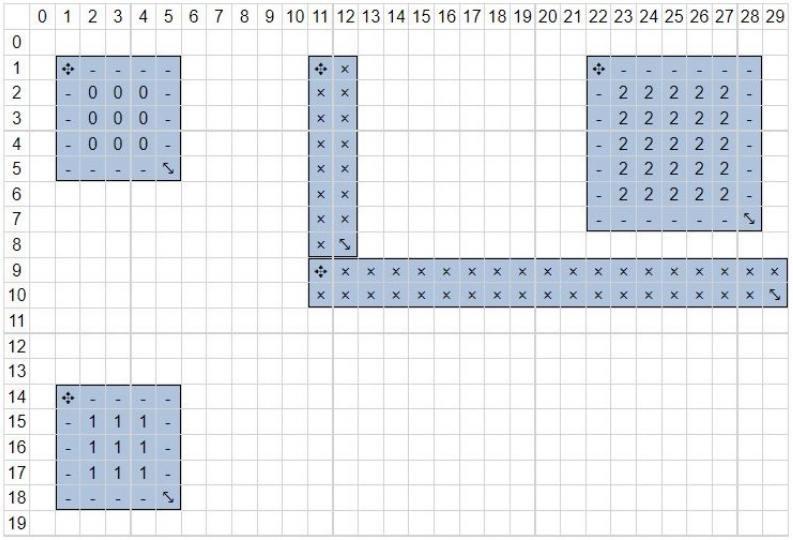

Abbildung 1: Beispielumgebung eines ”Profit!”-Spiels

### Lagerstätte

Die Menge an Ressourcen einer Lagerstätte wird durch die Größe der Lagerstätte festgelegt. Diese wird auf das Fünffache der Anzahl an Lagerstättenfelder gesetzt. Ist also eine Lagerstätte $3 \times 3$ groß dann enthält es $3 \cdot 3 \cdot 5 = 45$ Ressourcen eines bestimmten Subtyps. Insgesamt gibt es 8 Subtypen (0-7).

### Hindernisse

Hindernisse im Spielfeld stellen Felder dar, in denen kein anderes Gebäude gebaut werden kann. Diese sind beliebig groß, haben aber immer eine rechteckige Form.

### Produkt

Für jedes Spiel ist mindestens ein Produkt definiert, maximal acht. Ein Produkt benötigt eine beliebige Kombination der acht Ressourcen. Die benötigte Menge der jeweiligen Ressource ist ebenfalls definiert. Beispielsweise kann zur Herstellung von Produkt 0 dreimal die Ressource 0 und einmal die Ressource 1 benötigt werden. Jedes Produkt gibt eine bestimmte Anzahl an Punkte.

### Mine

Um die Ressourcen in den Lagerstätten abzubauen, gibt es Minen. Sie ist $4 \times 2$ oder $2 \times 4$ Felder groß und hat vier Subtypen, die die Rotation der Mine bestimmen. Die Abbildung 2 zeigt die vier verschiedenen Subtypen der Mine. Jede Mine hat einen Eingang (+) und einen Ausgang (-). Der Eingang muss an einem Ausgang einer Lagerstätte anliegen, um die Ressourcen abzubauen. An dem Ausgang können Eingänge von Förderbändern, Verbindern oder Fabriken anliegen.

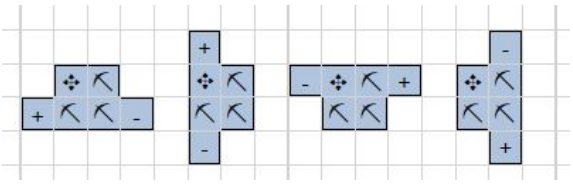

Abbildung 2: Darstellung der vier Minen-Subtypen

### Förderband

Um Ressourcen von einer Lagerstätte zu einer Fabrik zu befördern, können Förderbänder genutzt werden. Sie stellen zusätzliche Verbindungsstücke zwischen Gebäuden dar. Förderbänder haben einen Eingang (+) und einen Ausgang (-) und sind entweder 3 oder 4 Felder lang. Die Abbildung 3 zeigt die je in vier Subtypen der beiden Varianten. Somit hat das Förderband insgesamt acht Subtypen. Im Gegensatz zu allen anderen Objekten dürfen sich Förderbänder auch kreuzen.

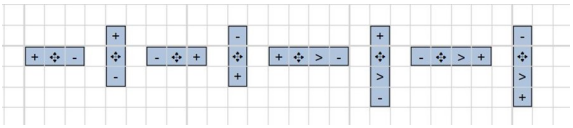

Abbildung 3: Darstellung der acht Förderband-Subtypen

### Verbinder

Wenn ein Produkt mehrere Ressourcen benötigt, können diese mit einem Verbinder zusammengeführt und gemeinsam zur Fabrik befördert werden.
Ein Verbinder hat drei Eingänge (+) und einen Ausgang (-). Auch hier gibt es vier Subtypen, die jeweils die Rotation des Verbinders bestimmen (Abbildung 4).

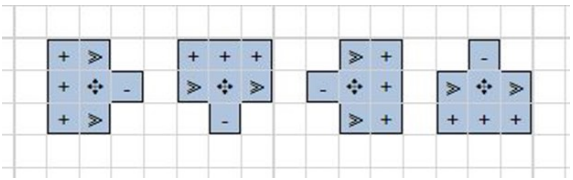

Abbildung 4: Darstellung der vier Verbinder-Subtypen

### Fabrik

Für jede Fabrik ist definiert, wie viele und welche Ressourcen gebraucht werden, um ein Produkt herzustellen. Eine Fabrik stellt immer nur ein Produkt her. Da es maximal acht Produkte geben kann, gibt es von der Fabrik insgesamt acht verschiedene Subtypen, für jedes Produkt einen. Jede Fabrik ist $5 \times 5$ groß. Die äußeren Felder sind Eingänge für Ressourcen, insgesamt 16.
Mit einer Kombination aus Mine, Förderband und Verbinder werden die Ressourcen von den Lagerstätten zur Fabrik befördert. Die Abbildung 5 zeigt, wie so ein Aufbau aussehen kann. Im Beispiel benötigt das Produkt 0 die Ressourcen 0 und 1.

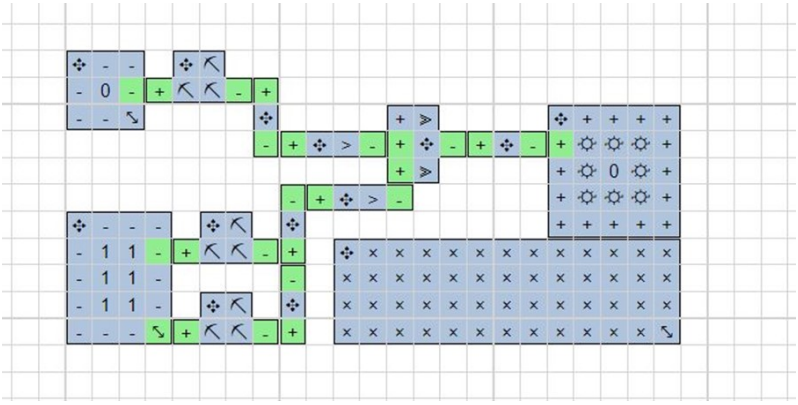

Abbildung 5: Lösungsbeispiel einer Aufgabe

## 2.2 Spielablauf

Das Spiel läuft rundenbasiert ab. Jede Runde beginnt mit einer “Beginn der Runde”-Aktion und endet mit einer “Ende der Runde”-Aktion. Die Tabelle 1 definiert die Aktionen der einzelnen Objekte. Das Spielfeld hat neben der Feldgröße und den

| **Objekt** | **Beginn der Runde** | **Ende der Runde** |
|---|---|---|
| Mine | Nimmt Ressourcen auf | Gibt angenommene Ressourcen weiter |
| Förderband | Nimmt Ressourcen auf | Gibt angenommene Ressourcen weiter |
| Verbinder | Nimmt Ressourcen auf | Gibt angenommene Ressourcen weiter |
| Fabrik | Nimmt Ressourcen auf | Produziert so viele Produkte wie Ressourcen verfügbar sind |
| Lagerstätte | - | Gibt bis zu 3 Ressourcen an jeden Eingang einer Mine |

Tabelle 1: Beschreibung der Aktionen der einzelnen Objekte

vorhandenen Objekten auch das Attribut “turns”. Dieses Attribut gibt die maximal erlaubte Anzahl an Spielrunden an.

Pro Runde werden Ressourcen von Lagerstätten mit Hilfe von Minen abgebaut und Stück für Stück über die gebauten Förderbänder und gegebenenfalls Verbinder zu einer Fabrik befördert. Die Produktion einer Fabrik stoppt, wenn die für das Produkt benötigten Ressourcen vollständig abgebaut und zu dieser Fabrik befördert sind. Ist das bei allen gebauten Fabriken der Fall, so endet das Spiel und der Spieler erhält eine Gesamtpunktzahl und die Anzahl der dafür benötigten Runden. Das Spiel kann auch vorzeitig beendet werden, wenn die vordefinierte maximale Rundenanzahl erreicht wird.

Das Ziel des Spiels ist es, die Punktezahl zu maximieren und dabei als Nebenziel die Rundenanzahl zu minimieren.

## 2.3 Darstellung in JSON

Der Input des Spiels erfolgt im JSON-Format (JavaScript Object Notation), in dem Lagerstätten, Hindernisse und Produkte definiert sind. Das JSON für das Beispiel in Abbildung 1 wird in der Abbildung 6 dargestellt. Das Spielfeld wird durch die angegebene Breite und Höhe definiert. Es enthält fünf verschiedene Objekte, drei Lagerstätten und zwei Hindernisse. Jedes dieser Objekte besitzt eine x- und y-Koordinate, die die Position im Feld bestimmt, sowie eine Höhe und eine Breite. Lagerstätten haben einen Subtyp, welcher die hier verfügbare Ressource festlegt. Nach den Objekten werden die Produkte definiert. Im JSON sind hierfür der Subtyp, die benötigten Ressourcen pro Produkt und die Anzahl an Punkten pro Produkt angegeben. Die Position der Ressource im Array bestimmt den jeweiligen Subtyp.

```
{"width":30,"height":20,"objects":[
{"type":"deposit","x":1,"y":1,"subtype":0,"width":5,"height":5},
{"type":"deposit","x":1,"y":14,"subtype":1,"width":5,"height":5},
{"type":"deposit","x":22,"y":1,"subtype":2,"width":7,"height":7},
{"type":"obstacle","x":11,"y":9,"width":19,"height":2},
{"type":"obstacle","x":11,"y":1,"width":2,"height":8}],
"products":[{"type":"product","subtype":0,"resources":[3,3,3,0,0,0,0,0]
,"points":10}],"turns":50}
```


Abbildung 6: JSON Darstellung einer Aufgabe

Jedes Objekt kann durch einen JSON-String dargestellt werden, der die Eigenschaften des Objekts definiert. In folgender Tabelle 2 wird für jeden Typ ein beispielhafter JSON-String aufgeführt:

| **Objekt** | **JSON** |
|---|---|
| Mine | {”type”:”mine”, ”subtype”:0, ”x”:0, ”y”:0} |
| Förderband | {”type”:”conveyor”, ”subtype”:0, ”x”:0, ”y”:0} |
| Verbinder | {”type”:”combiner”, ”subtype”:0, ”x”:0, ”y”:0} |
| Fabrik | {”type”:”factory”, ”subtype”:0, ”x”:0, ”y”:0} |
| Lagerstätte | {”type”:”deposit”, ”subtype”:0, ”x”:0, ”y”:0, ”width”:1, ”height”:1} |
| Hindernisse | {”type”:”obstacle”, ”x”:0, ”y”:0, ”width”:1, ”height”:1} |

Tabelle 2: Beschreibung der JSON Struktur einzelner Objekte

# 3 Mögliche Lösungsansätze

Mit AlphaGo und später AlphaZero hat das Unternehmen Deepmind gezeigt, dass ein Computer durch Deep Reinforcement Learning (RL) lernen kann, hochkomplexe Spiele wie Go und Schach auf professionellem Niveau zu spielen. [Se17]
Schach und Go besitzen ein festes $8 \times 8$ bzw. $19 \times 19$ Spielfeld, wohingegen das zu lösende Spiel "Profit!" ein bis zu $100 \times 100$ Feld umfasst. Es ist jedoch von der Spielweise weniger komplex und sollte deshalb ebenfalls mit verschiedenen RL Methoden lösbar sein. In den folgenden Abschnitten werden mögliche Lösungsansätze vorgestellt.

Wie bei vielen Problemen der Informatik gibt es auch hier nicht nur eine Lösung. Für diese Arbeit wurden mehrere Lösungsansätze ausprobiert, die in den folgenden Abschnitten kurz erläutert werden.

## 3.1 Reinforcement Learning

Für dieses Projekt bietet sich RL, auf deutsch das bestärkende Lernen, an. RL-Methoden gehören zu den Methoden des maschinellen Lernens(ML), welche wiederum ein Teilbereich der künstlichen Intelligenz (KI) sind. Grundsätzlich geht es darum, dass ein Agent durch eine Trial-and-Error-Methode seine Umgebung kennenlernt. Die Umgebung kann sich dabei dynamisch ändern. RL beschreibt eher eine Klasse an Problemen, als Set von verschiedenen Techniken

Das Standard Modell besteht aus folgenden Mengen:

- Menge an Umgebungs-Zuständen $S$ (states)
- Menge an Agenten-Aktionen $A$(actions)
- Menge an Belohnungen $R$ (rewards)

Der Agent befindet sich in der Umgebung in einem bestimmten Zustand. Er wählt eine Aktion aus und bekommt dafür eine positive, eine negative oder gar keine Belohnung. Aktionen können Agenten zusätzlich in einen neuen Zustand führen. Mathematisch kann dies folgendermaßen ausgedrückt werden:

$$f(S_t, A_t) = (S_{t+1}, R_{t+1}) \tag{1}$$

Ziel für den Agenten ist es, die durch Aktionen erhaltene Belohnungen zu maximieren. Die Abbildung 7 soll zeigen wie der Agent mit seiner Umgebung interagiert. Durch Ausprobieren lernt der Agent seine Umgebung kennen und lernt eine Strategie oder Policy $\pi$ mit der er seine Belohnungen maximieren kann. Im Gegensatz zu Supervised Learning wird dem Agenten nie mitgeteilt was die beste Aktion auf lange Sicht gewesen wäre [KLM96, S.237ff]
In dieser Arbeit stellt das Spielfeld die Umgebung dar. Es kann maximal $100 \times 100$ Felder groß sein. Der Zustand der Umgebung wird durch die vorhandenen Gebäude bestimmt. Anfangs sind nur Lagerstätten und Hindernisse vorhanden, mit jedem neuen Bauteil verändert sich der Zustand des Spiels.

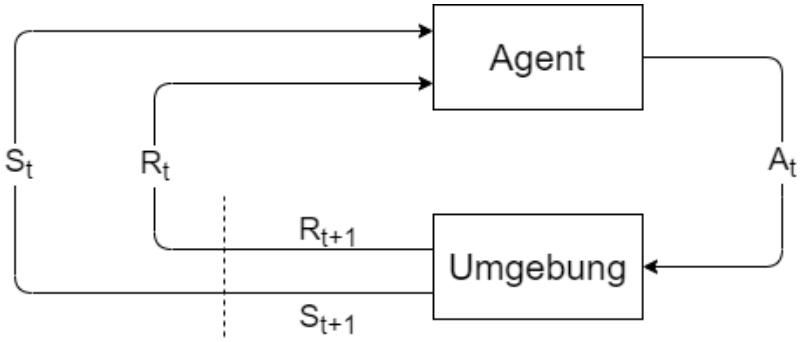


Abbildung 7: Diagramm der Interaktion des Agenten mit der Umgebung [SB20, S.48]

Die Aufgabe des Agenten ist es, Fabriken zu bauen, mit Minen die Ressourcen abzubauen und diese mit Förderbändern und Verbindern zur entsprechenden Fabrik zu befördern, damit Produkte gebaut werden können. Die Aktionen, die der Agent also ausführen kann, definieren das Aktionen-Set $A$.
Das Set an Belohnungen, also die Rewards, kann frei gewählt werden und ist nicht vom Spiel vorgegeben. Es ist sinnvoll den Agenten zu belohnen, wenn er Produkte erstellen kann und zu bestrafen, wenn er illegale Aktionen ausführen möchte

## 3.2 Deep-Q-Learning

Deep Q Learning ist eine RL Methode, die die ideale Policy mittels eines Neuronalen Netzes approximiert. Sie basiert auf dem sogenannten Q-learning.
Das Ziel des RL-Programms ist es, die Rewards $R$, die es während des Spiels erhält, zu maximieren. Der Rückgabewert, oder auch Return genannt, gibt die Summe der Rewards zurück. Die Gleichung 2 beschreibt diesen Return.

$$G_t = R_{t+1} + R_{t+2} + ... + R_T \tag{2}$$

Der erste Reward ist zum Zeitpunkt $t + 1$ und geht bis zum Zeitpunkt $T$, sofern $T$ eine endliche Menge an Zeitschritten ist. Wenn das nicht der Fall ist, wird eine discount rate $\gamma$ eingeführt, mit $0 <= \gamma <= 1$. Der Return wird wie in Gleichung 3 gezeigt angepasst.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{T-1} \gamma^k R_{t+k+1} \tag{3}$$

Spätere Rewards werden dann weniger stakt gewichtet als frühere.[SB20, S.54]

Die Policy $\pi(a|s)$ gibt die Wahrscheinlichkeit an, dass die Aktion $a$ gewählt wird, wenn der State $s$ gegeben ist. Es ist also eine Wahrscheinlichkeitsverteilung von $s \in S$ über alle $a \in A$. Gesucht ist die optimale Policy.

Neben der Policy gibt es auch Value-Fuctions, also Funktionen, die bestimmen, wie gut ein gewisser State ist oder wie gut eine Aktion in einem bestimmten State ist.

Die Funktion $v$ (Gleichung 4), auch state-value-function genannt, gibt an, wie gut ein State $s$ ist, wenn der Agent der policy $\pi$ folgt.

$$v_\pi(s) = E(G_t|S_t = s) \tag{4}$$

Die action-state-function $q$ (Gleichung 5), auch q-function, bestimmt wie gut eine Aktion $a$ ist, wenn sich der Agent in state $s$ befindet und der Policy $\pi$ folgt.

$$q_\pi(s, a) = E(G_t|S_t = s, A_t = a) \tag{5}$$

Die q-function gibt sogennnante q-values für jede mögliche Aktion aus, anhand dessen bestimmt werden kann welche Aktion der Agent aussuchen soll. [WD92, S.279f] [SB20, S.58] Das Ziel von Q-learning ist die beste policy $\pi^*$ zu finden. Die optimale policy wird über das finden der optimalen Funktionen $q^*$ und $v^*$ bestimmt. [WD92, S.281f]

Um zu bestimmen, ob eine Policy besser ist als eine andere, werden jeweiligen Values $v$ verglichen. Eine Policy $\pi$ ist besser als eine Policy $\pi'$ andere wenn $v_\pi(s) >= v'_\pi(s)$ ist, für alle $s \in S$. Gesucht ist also $v^*_\pi(s) = max\ v_\pi(s)$ Das gleiche gilt auch für die $q$ Funktion. $q^*_\pi(s, a) = max\ q_\pi(s, a)$.[SB20, S.62]
Um die optimalen Funktionen $v$ und $q$ zu finden müssen beide die Bellman-Gleichung erfüllen [SB20, S.73] . Das zeigen die Gleichung 6 und 7

$$v^*_\pi(s) = maxE[R_{t+1} + \gamma v^*_\pi(S_{t+1})|S_t = s, A_t = a] \tag{6}$$

$$q^*_\pi(s, a) = E[R_{t+1} + \gamma max q^*_\pi(S_{t+1}, a')|S_t = s, A_t = a] \tag{7}$$

Im Q Learning werden die $q$ Werte für jeden State $s$ und für jede Aktion $a$ gespeichert und Schritt für Schritt angepasst bis die beste Policy gefunden wurde. Anfangs sind alle Werte mit 0 initialisiert. Für das Lernen der Policy wird eine Exploitation(Ausnutzung) und Exploration(Erkundung) Methode verwendet. Anfangs soll das Programm seine Umgebung erkunden und sie damit kennenlernen, da es Anfangs nicht weiß, welche Aktionen zu einem Reward führen. Dieses Verhalten wird auch Exploration genannt. Bei der Exploitation nutzt das Programm das gelernte Wissen, um die Summe der Rewards zu maximieren. [SB20, S.26] Exploitation und Exploration werden mit einer $\epsilon$-greedy Methode umgesetzt. In dieser Methode gibt es eine $\epsilon$ Wert, der die Wahrscheinlichkeit bestimmt, mit welcher eine zufällige Aktion $a$ ausgesucht wird. Anfangs ist dieser Wert hoch und das Programm macht viele zufällige Züge. Je mehr es gelernt hat, desto kleiner wird der $\epsilon$ Wert gesetzt und gelernte gute Aktionen werden gewählt.[SB20, S.100f]

Deep-Q-Learning (DQN) basiert auf den gleichen Ideen wie Q-Learning. Hier wird allerdings das Approximieren der idealen Q-Funktion von einem neuronalen Netz übernommen, wie beispielsweise einem CNN (Convolutional Neural Network). DQN ist eine bewährte Methode in RL und wurde bereits in anderen Projekten erfolgreich umgesetzt. Beispielsweise wurde das Spiel Atari mit einem CNN-basierten Agenten, welcher ein DQN, neben anderen ML-Methoden nutzt, gelöst. [Oe15, S.6f]

## 3.3 Actor-Critic

Genau wie DQN ist Actor-Critic (AC) eine Methode, die in RL häufig eingesetzt wird. Ein Actor-Critic-Modell besteht aus zwei Komponenten, dem Actor und dem Critic. Der Actor entscheidet, welche Aktionen ausgeführt werden sollen, der Critic bewertet diese Aktionen. [KT99]
AC ist dabei auf ähnlichen Ideen aufgebaut wie DQN. Statt einem neuronalen Netz, welches die Q-Werte lernt, besteht AC aus zwei neuronalen Netzen, dem Actor-Netz und dem Critic-Netz. Das Actor Netz soll die Policy approximieren, das Critic-Netz die Value-Funktion. Häufig wird die *state-value-function* vom Critic gelernt. [SB20, S.321]
Durch die Aufteilung in die Actor und in die Critic Komponente soll dieses Modell schneller seine Strategie lernen. Ein AC kann zusammen mit einem Monte-Carlo-Search-Tree trainiert werden, um das Training zu stabilisieren. Dieser Ansatz wird im kommenden Abschnitt erläutert.

## 3.4 Monte-Carlo-Search-Tree

Der hier vorgestellte Ansatz des Monte-Carlo-Search-Tree MCST entspricht grundlegend der Umsetzung des Papers zu AlphaGO Zero [Se16] , angepasst auf das Problem diese Projektes.

Als Modell wird wieder ein AC genutzt. Dieses wird mit einem MCST verbunden. Das Ziel hierbei ist, das Training zu stabilisieren und bessere Vorhersagen treffen zu können. Der Ablauf während des Trainings lässt sich grob in zwei Schritte unterteilen, welche wiederholt werden.

1. Sammeln von Erfahrungswerten mit Hilfe des MCTS
2. Training des AC mit den Erfahrungswerten

### Sammeln von Erfahrungswerten

Das Spiel wird mehrmals bis zu einem terminierenden Zustand durch gespielt.
Zunächst wird die Umgebung zurückgesetzt, um ein neues, zufällig generiertes Spielfeld zu erhalten.
Um nun die erste Aktion zu ermitteln, wird der MCTS aufgerufen.
In jedem Zustand wird MCTS genutzt, um die nächste bestmögliche Aktion herauszufinden. Der MCTS ist ein Baum bestehend aus Knoten, welche je folgende Informationen beinhalten:

- Der Zustand des Spielfelds
- Die Aktion, die ausgeführt werden muss, um vom Eltern-Knoten in diesen Zustand zu gelangen
- Die Wahrscheinlichkeit, dass ausgehend vom Eltern-Knoten diese Aktion gewählt wird (“prior”)

- Die Bewertung $W$ des Knotens (entspricht der Ausgabe des Critic-Netzwerks) Wie häufig dieser Knoten Besucht wurde ($N$)
- Die mittlere Bewertung $Q = W/N$

Der Aufbau des Baums verläuft in Iterationen. Eine Iteration ist dabei in folgende Phasen aufgeteilt:

1. **SELECTION**
   Von der Wurzel wird mit Hilfe des sogenannten UCB Scores jeder Nachfolger bewertet. Dieser bestraft häufiges Besuchen desselben Knoten und stellt Exploration sicher. Es wird der Nachfolger ausgewählt, der diese Formel maximiert. Dies wird wiederholt bis ein Blattknoten erreicht ist

2. **EXPANSION**
   Ist ein Blattknoten erreicht, wird dieser expandiert. Die Werte der Nachfolger $W$, $Q$ und $N$ werden mit 0 initialisiert. Der Prior entspricht der Wahrscheinlichkeit, dass diese Aktion ausgehend vom Parent ausgewählt wird und wird mit Hilfe des Actors ermittelt.
   Der Wert $W$ des gefundenen Blattknoten wird über das Critic-Netzwerk ermittelt. $N$wird um 1 inkrementiert und Q entsprechend aktualisiert.

3. **BACKUP**
   In diesem Schritt wird der Baum vom erreichten Blattknoten ausgehend zur Wurzel traversiert. Dabei werden die Werte $W$, $Q$ und $N$ jedes Knotens wie folgt aktualisiert:
   $$W = W + W_{blatt} \tag{8}$$
   $$N = N + 1 \tag{9}$$
   $$Q = W/N \tag{10}$$

Die Phasen 1-3 werden beliebig oft wiederholt (in AlphaGo 1.600x, in der Implementation dieses Projektes 160x). Anschließend ermittelt der MCTS die bestmögliche Aktion wie folgt.

4. **DECISION**
   Ausschlaggebend dafür, welche Aktion vom Wurzelknoten aus gewählt wird, ist die Häufigkeit der Besuche $N$ der Nachfolger. Befindet sich das Modell nicht im Training, wird die Aktion gewählt, welche $N$ maximiert. Während des Trainings wird aus den $N$ aller Nachfolger eine Verteilung ermittelt. Anschließend wird aus den Nachfolgern gesampelt, wobei die ermittelte Verteilung als Gewichtung genutzt wird. Damit ist es am wahrscheinlichsten, dass der Nachfolger mit maximalem $N$ gewählt wird, es bleiben aber noch Exploration möglich. Die gewählte Aktion ist die Rückgabe des MCTS.

Die so ermittelte Aktion wird nun ausgeführt. Der neue Zustand sowie die Verteilung der Aktionen im Wurzelknoten des MCTS (s.o.) werden je in einer Liste gespeichert (der Spielzug wird also gemerkt).

Anschließend ist der gewählte Nachfolger des MCTS der neue Wurzelknoten. Der oben beschrieben Ablauf wird wiederholt, um die nächste Aktion zu ermitteln.

Dies wird so lange wiederholt, bis ein terminierender Zustand erreicht ist. Die gesammelten Zustände und dazugehörigen Aktionsverteilungen werden nun um einen weiteren Wert ergänzt, welcher Auskunft darüber gibt, ob dieser Spieldurchlauf erfolgreich war (gewonnen $\rightarrow 1$) oder nicht (verloren $\rightarrow -1$).

Die Liste an Erfahrungswerten enthält nun zahlreiche Tupel der Form:
( board_state, action_distribution, game_outcome)

#### Training des AC mit den Erfahrungswerten

Beim Training des AC Models ist es das Ziel, dass sich das Modell möglichst ähnlich dem MCTS verhält. Es entsteht also eine zyklische Abhängigkeit. Das Modell trainiert, um möglichst dem MCTS ähnlich zu sein, die MCTS nutzt wiederum das Modell, um Entscheidungen zu treffen.

Das Modell erhält im Training Batches der Erfahrungswerte. Als Input dient der aktuelle Zustand des Spielfelds board_state. Das AC Modell hat zwei Outputs. Der Actor, welcher eine Wahrscheinlichkeitsverteilung der Aktionen in diesem Zustand ausgibt, und der Critic, der den Zustand bewertet.

Der Actor soll nun möglichst der Aktionverteilung des MCTS Nahe kommen. Hierfür wird der Cross-Entropy-Loss genutzt.

Der Critic wiederum soll im gegebenen Zustand möglich genau vorhersagen können, ob das Modell das Spiel gewinnt oder verliert. Hierfür wird der Mean Squared Error (MSE) zwischen der Vorhersage des Modells und des tatsächlichen Outcomes genutzt, welcher in den Erfahrungswerten *game_outcome* gegeben ist.

#### Problem

In der aktuellen Version werden Trainingsdaten über 1024 Spiele ermittelt, welche das Modell dann in 100 Epochen mit einer Batch Size 8 trainiert.

Der wiederholte Aufbau des MCTS benötigt allerdings viel Zeit. Denn jeder Knoten benötigt eine Kopie der *Environment*-Instanz. Das Kopieren dieser sowie die zahlreichen Vorhersagen durch das Modell scheinen nach einer Analyse der Laufzeit am aufwändigsten zu sein und bremsen das Sammeln von Trainingsdaten so stark, dass dierse Ansatz nicht für dieses Projekt ausgewählt wurde.

### 3.5 Regelbasierter Ansatz

Neben den Machine Learning Ansätzen wäre auch ein regelbasierter Ansatz denkbar. Anstatt ein Modell auf das Problem zu trainieren, werden feste Regeln im Code implementiert, die das Programm abarbeitet.

Bei vielen Gebäuden macht es keinen Sinn, diese beliebig zu platzieren. Minen sollten beispielsweise an Lagerstätten angeschlossen werden, woran dann eine Fabrik oder ein Förderband angeschlossen wird. Wenn die Position der Fabrik steht, könnte mit Pathfinding-Algorithmen die Fabrik mit den gegebenen Lagerstätten verbunden werden.

Das ideale Setzen der Fabrik ist auch als Mensch nicht einfach. Es sollte möglichst Nahe bei allen benötigten Lagerstätten sein, ohne mögliche Wege zu blockieren, die gegebenenfalls für andere Produkte gebraucht werden. Daher ist es schwierig eine gute Metrik zu finden, wie idealerweise eine Fabrik platziert werden soll. Häufig ist die Anzahl der möglichen Fabrik-Positionen auch auf wenige Möglichkeiten begrenzt.

Ein regelbasierter Algorithmus würde, um ein Produkt zu erstellen, als erstes die Fabrik und die Minen an den Lagerstätten zu setzen und anschließend versuchen diese miteinander zu verbinden. Welche Produkte produziert werden sollen ließe sich berechnen und optimieren, um die erreichte Punktzahl zu maximieren.

# 4 Lösungsumsetzung

In diesem Kapitel wird die Umsetzung des Projekts beschrieben. Hierfür wurde eine Mischung aus einem DQN und einem regelbasierten Ansatz gewählt.

## 4.1 Programmiersprache und Bibliotheken

Das Projekt wurde in der Programmiersprache Python umgesetzt. Diese bietet eine einfache und übersichtliche Syntax, sowie große Unterstützung von vielen Bibliotheken im ML-Bereich.
Neben Numpy und Pandas zur Datenverarbeitung, wird TensorFlow bzw. Keras genutzt, um verschiedene neuronale Netzwerke aufzubauen und zu trainieren. Die OpenAI Gym vereinfacht die Interaktion zwischen dem RL-Agent mit der erstellten Implementierung der "Profit!"-Umgebung.
Für einen einheitlichen Code wurden folgende Code Konventionen eingehalten.

| | |
|---|---|
| Klassennamen | "UpperCamelCase"-Variante |
| Konstanten | "SNAKE_CASE_ALL_CAPS "-Variante |
| Variablen | "lower_snake_case "-Variante |
| Funktionen | "lower_snake_case "-Variante |

Tabelle 3: Beschreibung der Code Konventionen

## 4.2 Grundidee

Bei "Profit!" handelt es sich um ein Einzelspieler-Spiel mit einer deterministischen Umgebung, in der es keine versteckten Informationen gibt. Eine Implementierung des Spiels wurde über die Webseite https://profit.phinau.de/ bereitgestellt, in der ein Mensch mittels "drag & drop" Gebäude platzieren und den Ressourcenabbau simulieren kann. Diese Umgebung wurde in Python nach implementiert, um die volle Kontrolle über die Spieldynamik zu erhalten. Somit können RL-Agenten uneingeschränkt ihre Umgebung abfragen.

Das Spiel hat in seiner Grundform aufgrund des bis zu $100 \times 100$ großen Spielfeldes einen sehr großen Zustands- und Aktionsraum, weshalb die Entscheidung gefallen ist, das Problem nach dem Teile-und-Herrsche-Prinzip in mehrere Teilprobleme zu zerlegen.

Um eine geeignete Abstraktion zu finden, wurde sich daran orientiert wie ein Mensch beim "Profit"-Spiel spielen üblicherweise vorgehen würde: Anstatt im ganzen Spielfeld unzusammenhängende Gebäude zu platzieren, geht ein menschlicher Spieler (in der Regel) systematischer vor, indem er als erstes eine Fabrik an einer geeigneten Position platziert und von einer Lagerstätte ausgehend aneinander schließende Gebäude setzt, bis eine Verbindung zur Fabrik hergestellt wurde. Dabei behält ein menschlicher Spieler einen groben Überblick über das ganze Spielfeld, um die ungefähre Richtung, in die er bauen möchte, zu bestimmen. Um mögliche Hindernisse

und Sackgassen zu vermeiden oder Förderbänder mit anderen Förderbändern zu überkreuzen, wird die lokale Umgebung des zuletzt gesetzten Gebäudes betrachtet, um das optimale Bauteil auszuwählen.

Anhand dieser Vorgehensweise ist ersichtlich, dass ein Agent für das Verbinden eines beliebigen Startgebäudes mit einer Fabrik zuständig sein soll. Dieser Agent soll ähnlich wie ein menschlicher Spieler aneinander angrenzende Gebäude platzieren, bis das Zielgebäude erreicht ist. Minen, Förderbänder und Verbinder besitzen alle nur einen Ausgang. Lagerstätten können viele verschiedene Ausgänge haben. Zur vereinfachten Implementierung wurde sich dafür entschieden, eine Mine als Startgebäude des Agentens anstatt einer Lagerstätte zu verwenden. Somit wird die Position des Agenten auf den einzigen Ausgang des zuletzt platzierten Gebäudes gelegt. Des Weiteren müssen nur vier benachbarte Positionen in Betracht gezogen werden, an denen der Eingang eines neuen Gebäudes platziert werden kann.

Um “Profit!” zu lösen, braucht es noch einen weiteren Agenten, der bestimmt, welche Lagerstätte bzw. Mine mit welcher Fabrik verbunden werden soll und was zu tun ist, falls eine Verbindung nicht hergestellt werden kann. Außerdem muss dieser Agent bestimmen, wo Fabriken und Minen platziert werden sollen, damit der erste Agent diese verbinden kann.

Im Weiteren wird der erste Agent als untergeordnet und der zweite als übergeordnet bezeichnet. Beide können sowohl durch RL-Methoden trainiert, als auch mit Hilfe eines regelbasierten Ansatzes gelöst werden.

## 4.3 “Profit!” Umgebung

Der erste Schritt der Lösungsumsetzung war das Nachimplementieren von ”Profit!”selber. Hierfür wurde eine Umgebungsklasse (Environment) erstellt, die wie die Umgebung der Website agieren soll. Die Umgebung kann aus einer JSON-Datei erstellt werden und hat auch die gleichen Eigenschaften. Anfangs beinhaltet sie nur Lagerstätte und Hindernisse. Über Methoden lassen sich dann die in Kapitel 2 beschriebene Gebäude hinzufügen. Eine fehlerhafte Umgebung könnte dafür sorgen, dass die Lösung unzulässige Gebäude verwendet. Daher war es wichtig, dass diese Umgebung mit allen dazugehörigen Klassen korrekt umgesetzt wurde. Um dies sicherzustellen wurden Unittests für die einzelnen Funkionen implementiert

Die folgenden Abschnitte beschreiben, wie sich diese Profit Umgebung zusammensetzt und wie sie getestet wurden.

### Environment-Klasse und Gebäude

Die *Environment*-Klasse prüft für jedes neu platzierte Gebäude, ob alle in 2.1 definierten Regeln eingehalten werden. Jedes Gebäude speichert alle Referenzen zu an eigenen Ausgängen angrenzenden Gebäuden. Somit ist es einfach rekursiv zu ermitteln, ob eine Lagerstätte mit einer Fabrik verbunden ist. Die Gebäude selber erben alle von ihrer Basisklasse *Building*. Die *Building*-Klasse enthält die Attribute

die alle Gebäude gemeinsam haben: Eine $x$- und $y$-Koordinate, einen Subtypen und eine Form. Sie beinhaltet auch verschiedene Funktionen die für das verwenden von Gebäuden wichtig sind. Lagerstätte und Hindernisse erben von der sogenannten *UnplaceableBuildings*-Klasse, welche wiederum von *Building* erbt. Damit soll sichergestellt werden, dass keine weiteren Lagerstätte oder Hindernisse vom Agenten gebaut werden können. Die Abbildung 8 zeigt ein Klassendiagramm der Gebäudetyen.

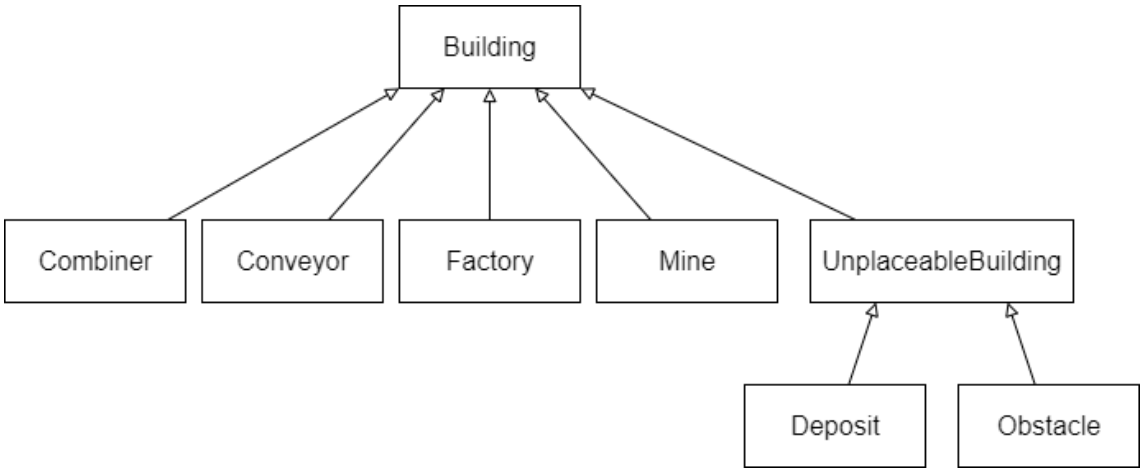


Abbildung 8: Klassendiagramm der Gebäudetypen

Die *Environment*-Klasse kennt die Gebäude und kann sie als Objekte in die Umgebung aufnehmen. Gelöste Aufgaben werden als JSON-Datei unter dem Pfad */tasks/solutions* gespeichert und die Liste an platzierbaren Gebäuden wird ausgegeben. Alternativ kann eine für Menschen lesbare Repräsentation des Spielfeldes auf der Standardausgabe ausgegeben werden. Um das Spiel zu simulieren gibt es die Spiel Simulation, welche im nächsten Abschnitt vorgestellt wird.

### Spiel-Simulation

Die Simulation des Spiels dient zur Überprüfung der erreichten Punktzahl eines gegebenen Spielfelds. Zusätzlich wird die Anzahl der benötigten Runden ermittelt, um die angegebene Punktzahl zu erreichen. Der Simulator befindet sich in der Datei *simulator.py* und ist als Klasse implementiert.

Bei der Initialisierung einer Klasseninstanz wird dem Konstruktor eine Instanz der *Environment*-Klasse übergeben. Diese enthält alle Informationen über die auf dem Spielfeld befindlichen Objekte, deren Position und die jeweiligen Verbindungen zwischen ihnen. Außerdem können der *Environment*-Klasse die für dieses Spiel möglichen Produkte und das dazugehörige Rezept entnommen werden.

Jedes platzierbare Objekt verfügt über einen eigenen Cache und einen internen Ressourcenspeicher, welcher die Anzahl der aktuell gehaltenen Ressourcen des jeweiligen Objekts angibt. Zusätzlich hat jedes Objekt typabhängig eine Funktion, welche zum Beginn (*start_of_round_action*) und zum Ende (*end_of_round_action*) einer Runde aufgerufen wird.

Die *start_of_round_action*-Funktion des Förderbandes beispielsweise entnimmt alle im Cache befindlichen Ressourcen und fügt diese dem internen Speicher hinzu. Am Ende der Runde überträgt die *end_of_round_action*-Funktion die im Speicher befindlichen Ressourcen in den Cache des Ausgangs des befindlichen Objekts.

Während der Simulation wird zum Start einer Runde in zufälliger Reihenfolge die *start_of_round_action*-Funktion jedes Objekts aufgerufen. Anschließend werden die *end_of_round_action*-Funktionen aufgerufen. Eventuell entstehende Produkte und damit Punkte werden mit der bisherigen Gesamtpunktzahl summiert.

Der Simulator gibt nach Abschluss der Simulation die erreichte Gesamtpunktzahl sowie die dafür benötigten Runden zurück.

### Optimale Score

Mit dem Optimal Score kann für eine gegebene Umgebung ermittelt werden, welche Punktezahl in der Theorie maximal erreicht werden kann. Außerdem wird ermittelt, welche Kombination von Produkten zu dieser optimalen Punktzahl führen. Wenn mehrere Produkte sich Ressourcen teilen müssen, kann es sein, dass es besser ist, ein Produkt wegzulassen, damit dieses einem anderen Produkt nicht die Ressourcen wegnimmt. Für die Berechnung wird angenommen, dass ideale Bedingungen herrschen. Die Positionen, an welchen sich die Lagerstätte und Hindernisse befinden, werden nicht in Betracht gezogen. Im ersten Schritt wird eine Liste erstellt, die alle möglichen Kombinationen von Produkten enthält. Anschließend wird für jede Produktkombination der Optimal Score berechnet. Der Score und die Produktkombination werden in eine Liste geschrieben, welche sortiert ausgegeben wird. Die Rückgabe des Optimal Scores ist also eine sortierte Liste an Scores und Produktkombinationen, wobei die beste Kombination an erster Stelle steht.

Um den Score für ein einzelnes Produkt zu berechnen, werden die Punktezahl sowie die benötigten Ressourcen des Produkts gebraucht, ebenso wie die Lagerstätte und die Gesamtressourcen der Umgebung. Die Anzahl der Runden wird um zwei reduziert, da mindestens zwei Runden gebraucht werden, bis eine Ressource eine Fabrik erreicht.

Die Berechnung des Optimal Scores erstellt einen Vektor mit den Ressourcen aller Lagerstätten, die vom Produkt gebraucht werden. Ebenso wird ein Vektor erstellt, welcher die maximale Anzahl aller Minen beinhaltet. Eine Lagerstätte von der Breite $b$ und der Höhe $h$ kann platzbedingt maximal $b \cdot h$ Minen versorgen. Dieser Vektor wird anschließend mit drei multipliziert, da jede Mine pro Runde drei Ressourcen aufnehmen kann.

Der Ressourcenvektor wird anschließend durch den Minenvektor geteilt. Dadurch wird ermittelt, wie viele Runden mindestens gebraucht werden, um alle Ressourcen abzubauen. Ist diese Zahl kleiner als die gegebene Rundenzahl-2, dann wird der Ressourcenvektor durch den Produktressourcenvektor geteilt, wodurch sich die Anzahl

an maximal zu produzierenden Produkten ergibt. Diese Zahl wird mit der Produktpunktzahl multipliziert, wodurch sich der optimale Score für dieses Produkt ergibt. Ist die errechnete Mindestrundenzahl größer als die gegebene, so wird ermittelt, wie viele Produkte in den gegebenen Runden maximal produziert werden können. Hier wird der Minenvektor mit der Anzahl der Runden multipliziert und durch den Produktressourcenvektor geteilt. Das Ergebnis sind die maximal reproduzierbaren Produkte, welche mit der Produktpunktzahl multipliziert den optimalen Score ergibt.

Wenn mehrere Produkte existieren, so gibt es zwei Möglichkeiten. Entweder die Ressourcen für die Produkte sind unabhängig voneinander, dann wird der Optimal Score aller Produkte addiert, oder Produkte teilen sich Ressourcen. Ist das der Fall, dann wird der Eintrag im Ressourcenvektor durch die Anzahl an Produkten, die sich diese Ressource teilen, geteilt. Anschließend wird der Optimal Score für jedes Produkt berechnet und addiert.

Der Optimal Score entspricht nicht dem tatsächlich möglich erreichbaren besten Wert. Da die Position der Lagerstätten und Hindernisse nicht beachtet wird, kann nicht überprüft werden, wie viele Minen tatsächlich verwendet werden können und wie viele Runden eine Ressource tatsächlich benötigt, um eine Fabrik zu erreichen. Der Optimal Score soll nur eine Aussage darüber geben, welche Punktzahl für ein Produkt gut ist und wann eine Lösung nicht weiter verbessert werden kann.

### Testing

Der Code wurde mit dem Unit Test Framework *unittest* von Python getestet. Dieses bietet eine einfache Weise Testfälle zu definieren und auszuführen. Jede Testklasse erbt von *unittest.TestCase*, die Testfälle werden als Methoden die mit test beginnen angegeben. Über die assert-Funktion können verschiedene Bedingen abgeprüft werden.[Fou] Die Tests und die Implementierung des Codes wurden von unterschiedlichen Entwicklern parallel ausgeführt . Damit sollte sichergestellt werden, dass beispielsweise Denkfehler oder Bugs in der Implementierung weniger wahrscheinlich auch in den Tests vorkommen, wodurch Fehler besser entdeckt werden können. Als Hilfe für die Testimplementierung wurde die Webseite https://profit.phinau.de/ verwendet. Die Webseite wurde für den Informaticup zur Verfügung gestellt und beinhaltet eine interaktive Implementierung des Spiels Profit. Die Implementierung des Spiels sollte identisch zu dem der Webseite sein.

### Environment & Gebäudeplazierung

In den Environment-Tests werden die verschiedenen Bedingungen der "Profit!"-Umgebung und ihre Gebäude überprüft. Es werden mehrere Test-Umgebungen aus einem JSON-String geladen. Die daraus resultierenden Environment-Objekte werden anschließend auf mögliche Fehler in ihrer Darstellung überprüft.
Es gibt viele verschiedene Regeln, welche Gebäude neben welchen gebaut werden dürfen. Die Gebäude-Tests platzieren Gebäude auf legale und illegale Weise in eine Umgebung und prüfen, ob diese das neue Gebäude korrekt akzeptiert bzw. verwirft. Beispielsweise darf ein Feld nicht von zwei verschiedenen Gebäuden belegt werden

oder Förderband-Eingänge dürfen nicht neben Lagerstätten-Ausgängen liegen. Auch die Ausrichtung der Gebäude-Objekte wird überprüft. In den Tests wird versucht, alle legalen und illegalen Handlungen abzudecken, um sicherzustellen, dass später keine illegalen Aktionen gelernt werden.

**Spielsimulation**
Für die Spielsimulation-Tests wird getestet, ob das endgültige Ergebnis des Spiels demselben entspricht wie dem der zur Verfügung gestellten Webseite. Beide Spielsimulationen müssen sowohl die gleiche Punktzahl haben als auch die gleiche Anzahl an Runden ausgeben.

**Optimal Score**
Der Optimal Score wurde anhand der gegebenen Aufgaben überprüft. Dabei wurde für jede dieser vier Ausgaben das beste Ergebnis händisch berechnet und anschließend mit dem Ergebnis des Algorithmus verglichen.

## 4.4 Untergeordneter Agent

Für das Verbinden einer Mine mit einer Fabrik wurde sich für einen RL-Ansatz entschieden. Dabei wurde mit vielen Netzwerkarchitekturen und möglichen Inputs experimentiert, um zwischen Modellkomplexität und benötigter Rechenzeit abzuwägen.

Das AC Modell hat im direkten Vergleich zu einem DQN schlechtere Ergebnisse geliefert und die MCST-Implementierung hat aufgrund höherer Rechenzeit für das Trainieren keinen Mehrwert geliefert. Nach mehreren erfolglosen Versuchen, die zugrundeliegenden Probleme zu lösen, mussten diese Lösungsansätze aufgegeben werden.

Das beste Ergebnis wurde mit einem DQN mit zwei versteckten Schichten und insgesamt 215.968 Parametern erzielt. Es bekommt nicht den Zustand des ganzen Spielfeldes, sondern nur einen Ausschnitt davon, sowie vorgefertigte Features als Input.

**Inputs**

Damit der Agent mit unterschiedlichen Spielfeldgrößen umgehen kann und um die Modellkomplexität in Grenzen zu halten, beschränken sich das Sichtfeld auf einen lokalen $15 \times 15$ Bereich, der die Position des Agenten umgibt. Für den Agenten ist es wichtig zu wissen, wo sich im Spielfeld nicht-freie Felder, innere Förderband-Felder, sowie Fabrikeingänge (und Eingänge, die bereits mit der Ziel-Fabrik verbunden sind) befinden. Diese Informationen werden jeweils in den Kanälen eines $15 \times 15 \times 3$ binären Tensors codiert. Für den Fall, dass das Sichtfeld über den Spielfeldrand hinausragt, wird das Spielfeld durch Hindernisse erweitert.

Falls sich kein Fabrikeingang im Sichtfeld des Agenten befindet, sollte dieser zumindest die ungefähre Richtung, in der sich die Ziel-Fabrik befindet, wissen. Deshalb gibt es sechs zusätzliche binäre Inputs, die die relative $x$- und $y$-Position (für jeweils

niedriger, höher, gleich) angeben.

Aus einem bisher nicht ersichtlichen Grund versuchte der Agent selbst nach längerem Training gelegentlich illegale Aktionen zu tätigen. Die konnte nur durch das Hinzufügen der legalen Aktionen als binärer Input entgegengewirkt werden.

### Outputs

Es wurde festgestellt, dass Verbinder nur in seltenen Fällen einen Mehrwert gegenüber Förderbändern bieten, da letztere auch zum Kombinieren von mehreren Ressourceflüssen genutzt werden können und gleichzeitig weniger Platz verbrauchen. Um den Aktionsraum noch weiter zu reduzieren und um das Training zu beschleunigen, wurden aus diesen Gründen alle Verbinder aus dem Spiel entfernt. Auch Minen sollten nur zum Abbauen von Ressourcen an Lagerstätten benutzt werden und sind somit für den untergeordneten Agenten nicht von Relevanz. Der Agent muss nur noch lernen, Förderbänder an geeigneten umliegenden Positionen zu platzieren. Der Aktionsraum ist somit auf $8 \cdot 4 = 32$ reduziert, acht Gebäude-Subtypen an jeweils vier mögliche Positionen. Die Abblidung 9 zeigt die verwendete Netzwerkarchitektur des untergeordneten Agenten

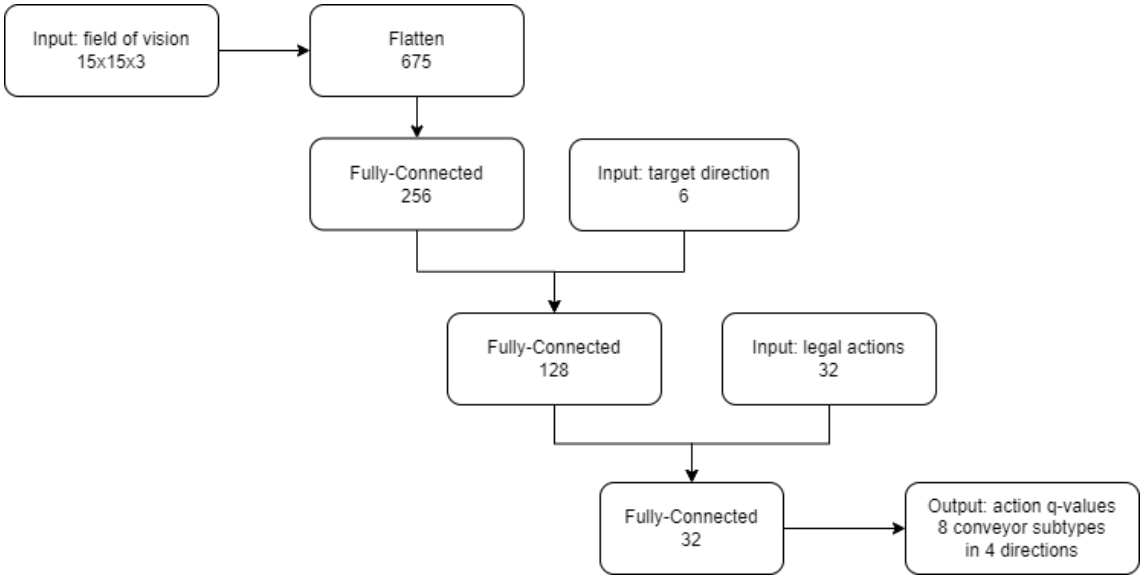


Abbildung 9: Darstellung der Netzwerkarchitektur

### Task Generator

Für das Training des DQN werden viele verschiedene Aufgabenstellungen benötigt. Diese von Hand zu erstellen wäre sehr zeitintensiv gewesen, weshalb stattdessen einen Task Generator implementiert wurde.

Die einfachste Variante einer Aufgabe kann in vier Schritten erstellt werden:

1. Jeweils eine Lagerstätte und eine Fabrik werden an zufälligen Positionen in einem leeren Spielfeld platziert

2. Diese werden mit Hilfe einer einfachen Distanz-Heuristik mit zufälligen (legalen) Gebäuden verbunden.

3. Anschließend wird auf jedem leeren Feld mit einer geringen Wahrscheinlichkeit ein 1x1-Hindernis platziert.

4. Die verbindenden Gebäude zwischen Lagerstätten und Fabrik werden bis auf die erste Mine wieder entfernt.

Diese vier Schritte werden in der Abbildung 10 nochmals verdeutlicht.

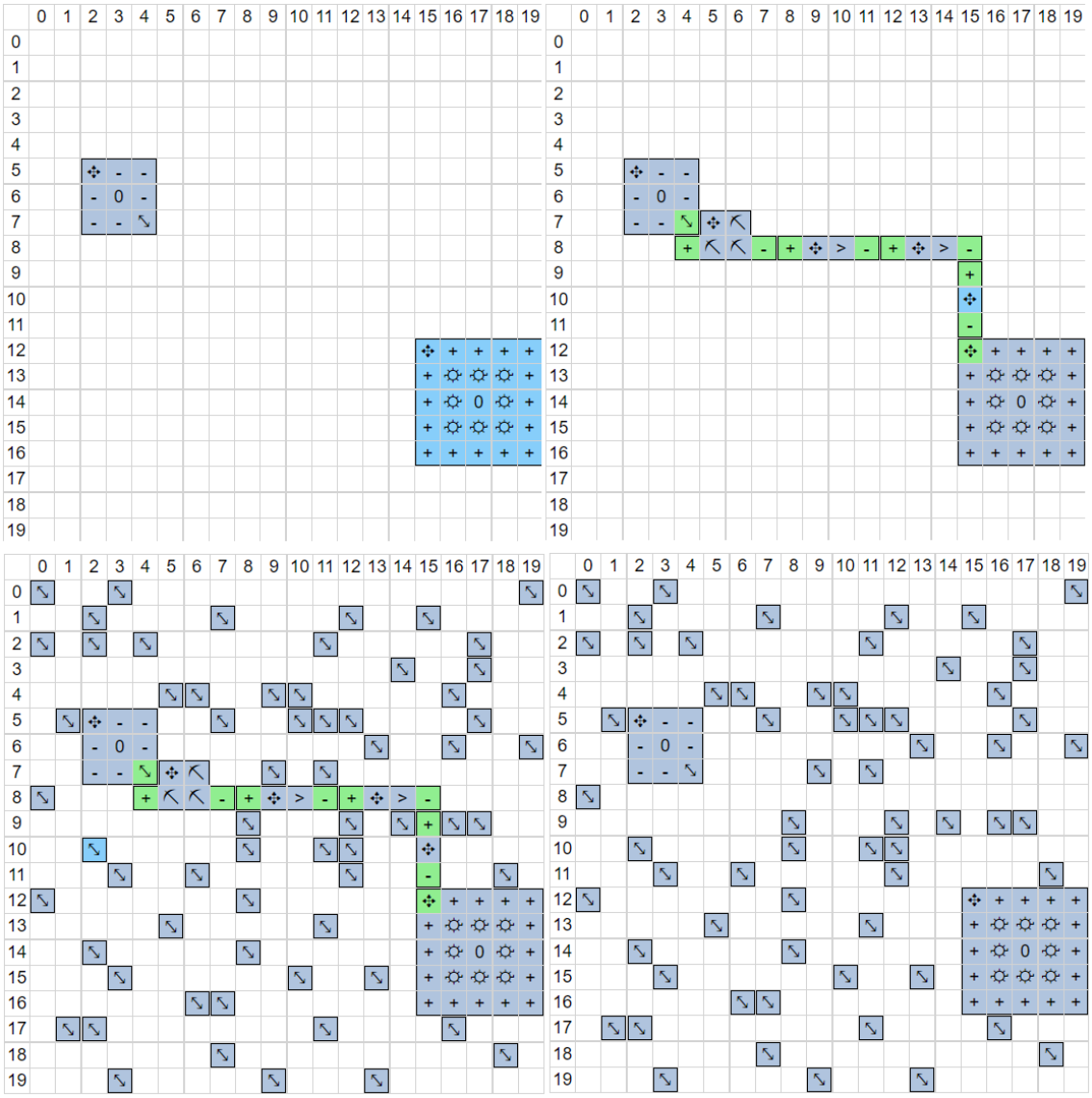

Abbildung 10: Abblidung der vier Schritte des Task Generators

Der Task Generator erstellt somit eine nicht-triviale Aufgabe, eine Mine mit einer Fabrik in einem Hindernis-Labyrinth zu verbinden, die mindestens eine Lösung hat. Damit der RL-Agent auch lernen kann, Förderbänder mit anderen Förderbändern zu

überkreuzen oder bereits existierende Verbindungen zur Ziel-Fabrik zu nutzen, werden zusätzliche Lagerstätten-Fabrik-Verbindungen erstellt. Dafür werden die Schritte eins und zwei wiederholt, ohne die verbindenden Gebäude in Schritt vier zu löschen.

#### Training

Trainiert wurde das DQN mit über 100.000 Epochen auf einer Spielfeldgröße von $30 \times 30$ mit dem Adam-Optimierer und einer konstanten Lernrate von 0,001. Um dem Problem von "sparse rewards" vorzubeugen, erstellt der Task Generator in frühen Epochen einfachere Aufgaben, in denen die Lagerstätte und Fabrik nur einen geringen Abstand haben. Außerdem wird für jedes neu gesetzte Gebäude eine geringe negative Belohnung von -0,05 gegeben, diese wird auf bis zu -0,01 verringert oder auf bis zu -0,09 erhöht, je nachdem ob sich der Abstand zur Ziel-Fabrik verringert oder erhöht. Diese Art von "reward shaping" soll bewirken, das Modell zu motivieren, in die "richtige Richtung" zu laufen und dabei so wenig Gebäude wie möglich zu verwenden. Für das erfolgreiche Erreichen der Fabrik wird eine große Belohnung von 1,0 gegeben; ein illegaler Zug wird mit -1,0 bestraft. Als Abschlagsfaktor für nachfolgende Belohnungen wird $\gamma = 0,9$ verwendet.

Nach dem Training ist das Modell in der Lage 83% der Aufgaben des Task Generators zu lösen. Der Agent lernt Hindernisse zu umgehen und sich auf die Fabrik mit möglichst wenigen Gebäuden zuzubewegen. Es ist zu bemerken, dass nicht immer erkannt wird, welcher Pfad in einer Sackgasse mündet, was zum Scheitern einer Aufgabe führen kann.

### 4.5 Übergeordneter Agent

Der übergeordnete Agent hat die Aufgabe, günstige Positionen für Fabriken und Minen zu finden, sowie zu koordinieren, welche Lagerstätte (bzw. welche zugehörige Mine) mit welcher Fabrik verbunden werden soll.

Ursprünglich wurde ein Multi-Agenten-System zu entwickeln angedacht, in dem die einzelnen untergeordneten Agenten eine kleine Belohnung erhalten wenn sie die ihnen zugewiesene Fabrik erreichen, eine mittlere Belohnung für alle Agenten die es schaffen gemeinsam ein Produkt erfolgreich zu produzieren und eine große Belohnung, wenn sogar die theoretisch maximale Punktzahl erreicht wird. Von diesem Ansatz wurde sich erhofft, dass die Agenten lernen zusammenzuarbeiten, um ähnliche Probleme wie die bereitgestellte "task.004" zu lösen, in der es nur wenig Platz für Gebäude gibt und somit Verbindungen doppelt genutzt werden sollten. Dies konnte aufgrund der nahenden Abgabefrist nicht mehr umgesetzt werden.

Der untergeordnete Agent braucht für das Erstellen einer Verbindung zwischen Mine und Fabrik nur wenig Zeit (meistens unter einer Sekunde). Deshalb wurde bei dem übergeordneten Agenten ein Brute-Force Ansatz gewählt. Die Produkte werden nacheinander in der Reihenfolge abgearbeitet, die theoretisch die zuvor berechnete optimale Punktzahl erreichen kann. Dafür wird für jedes mögliche Produkt genau

eine Fabrik an eine zufällige legale Position gesetzt. Anschließend werden für jede Lagerstätte alle möglichen Minen probiert, bis eine Verbindung mithilfe des untergeordneten Agenten hergestellt werden kann. Dabei werden Minen, deren Ausgang näher an der Zielfabrik liegt, als erstes verwendet. Falls mit keiner Mine eine Verbindung möglich ist, wird die Fabrik noch bis zu zehn Mal an eine andere zufällige Position verschoben.

Der übergeordnete Agent verwendet ein einfaches Zeitmanagement. In der Regel braucht das Programm wenige Sekunden zum finden einer Lösung. Falls im Anschluss noch mehr als 50% der Rechenzeit verbleiben, wird versucht, die Lösung zu verbessern. Dies kann in Ausnahmefällen trotzdem dazu führen, dass die gegebene Rechenzeit überschritten wird.

Ausgehend von der initialen Lösung wird für jeder erfolgreiche Lagerstätten-Fabrik-Verbindung versucht, noch weitere Verbindungen hinzuzufügen. Dies ist teilweise mit wenigen Gebäuden möglich, wenn Förderbänder an andere Förderbänder oder Minen angeschlossen werden, die bereits zur Ziel-Fabrik führen. Die zusätzlichen Verbindungen können die Anzahl der benötigten Runden reduzieren oder sogar bei einer geringen maximalen Rundenanzahl die Punktzahl erhöhen. Nach jeder Verbesserungsrunde, berechnet der Simulator die neue Punktzahl und die dafür benötigte Rundenanzahl. Wenn keine weitere Verbesserung erreicht wird, oder die Lösung sich verschlechtert, wird die beste Umgebung verwendet. Die platzierbare Gebäude werden als Liste im JSON-Format zurückgegeben.

## 4.6 Wartbarkeit

Die Spiel-Umgebung wurde so implementiert, dass es einfach ist, neue Gebäudetypen hinzuzufügen oder zu entfernen. Eine neue Gebäude-Klasse sollte von der Building- bzw. Unplaceable-Building-Klasse erben (siehe Abb. 8). Es muss für alle unterschiedlichen Subtypen die Form als zweidimensionale Liste angegeben werden, bestehend aus “+”, “-”, für Ein- und Ausgänge, sowie “ “ für freie Felder und einen beliebigen Buchstaben für inerte Felder. Über ein Python-Dictionary werden die Regeln definiert, welche Gebäude an andere Gebäude angrenzen dürfen. Die Abbildung 11 zeigt den Code, der die legalen Verbindungen definiert.

```
LEGAL_CONNECTIONS = {
    Deposit: [Mine],
    Mine: [Conveyor, Combiner, Factory],
    Conveyor: [Conveyor, Combiner, Mine, Factory],
    Combiner: [Conveyor, Combiner, Mine, Factory],
    Factory: [],  # Factories do not have outputs
    Obstacle: [],  # Obstacles do not have outputs
}
```

Abbildung 11: Code zum Definieren legaler Verbindungen

Die Environment-Klasse muss beim Hinzufügen oder Entfernen eines neuen Gebäudes nicht weiter angepasst werden, weil alle nötigen Regeln anhand der Gebäude-Subtyp-Form und dem Dictionary für legale Verbindungen abgeleitet werden können.

Zum Lösen der InformatiCup Aufgabenstellung ist die Anpassungsfähigkeit von Gebäuden nicht erforderlich, da alle Gebäude und ihre Regeln von Anfang an festgelegt wurden. Der gewählte Aufbau erhöht jedoch die Wartbarkeit und Debugfähigkeit des Systems. In der Anfangsphase hatte der RL-Agent aufgrund fehlerhafter Implementierung Schwierigkeiten zu lernen. Zum Debuggen wurde das Spiel zwischenzeitlich stark vereinfacht. Minen, Förderbänder, Verbinder wurden durch ein vereinfachtes $2x1$ bzw. $1x2$ Förderband mit nur vier Subtypen ersetzt. Auch Lagerstätten und Fabriken wurden auf eine Größe von $1x1$ reduziert. Eine ähnliche Spieldynamik bleibt erhalten, während die Komplexität sinkt, weil jedes Gebäude nur maximal einen Ein- und Ausgang besitzt.

# 5 Benutzerhandbuch

Dieses Kapitel soll erklären wie mit dem Projekt weitergearbeitet werden kann und wie Änderungen, gerade am Modell des untergeordneten Agenten vorgenommen werden können. Ebenso wird erläutert wie der Code in dem bereitgestellten Docker-Container auszuführen ist.

**Einstellungen**
Alle Hyperparameter der Agenten, Einstellungen für den Task Manager, sowie weitere Debug Informationen lassen sich gesammelt in der *settings.py* Datei einstellen. Es kann beispielsweise eine andere Modell-Architektur für den untergeordneten Agenten gewählt oder die Hinderniswahrscheinlichkeit des Task Generators angepasst werden. Auch das zuvor erwähnte vereinfachte Spiel kann durch Setzen von $SIMPLE\_GAME = True$ aktiviert werden.

**Trainieren eines untergeordneten Agenten**
Im Anschluss an eine Änderung in *settings.py* muss ein neuer untergeordneter Agent mittels *train_model.py* trainiert werden. Dies kann je nach Einstellungen und verwendeter Hardware mehrere Stunden in Anspruch nehmen. Um die Trainingszeit zu verringern, sollte in Betracht gezogen werden, die maximale Episodenzahl in *settings.py* zu reduzieren.

**Evaluation von untergeordneten Agenten**
Nachdem mehrere Agenten trainiert wurden, kann mithilfe von *evaluate_model.py* überprüft werden, welches Modell die meisten Aufgaben des Task Generators lösen kann und sich somit am besten in einem Hindernis-Labyrinth bewegt. Standardmäßig wird auf verschiedenen Spielfeldgrößen von $20 \times 20$, $30 \times 30$ und $50 \times 50$ evaluiert. Das Ergebnis dient nur als relativer Anhaltspunkt. Es sagt nichts darüber aus, wie gut der Agent mit “echten” Aufgaben zurechtkommt, da die Aufgaben des Task Generators sich von Menschen erstellten Aufgaben unterscheiden.

**Lösen von Aufgaben**
Wird *solve_game.py* mit einem Pfad zu einer JSON-Datei aufgerufen, wird nur diese Aufgabe gelöst. Alternativ kann mit “solve” als Parameter auch eine Aufgabe über die Standardeingabe eingelesen werden. Es werden alle Ausgaben auf die Standardausgabe unterdrückt, bis zum Schluss eine Lösung als Liste von platzierbaren Gebäuden zurückgegeben wird.

Das Ausführen von *solve_game.py* ohne weitere Parameter bewirkt, dass automatisch alle zuvor definierten Aufgaben nacheinander gelöst werden. Hierbei werden auch die initiale Lösung sowie jede Verbesserung davon auf der Standardausgabe ausgegeben. Um sich einen ausführlichen Lösungsweg anzeigen zu lassen, sollte in *settings.py* $DEBUG{=}True$ gesetzt werden. Damit werden auch fehlgeschlagene Verbindungswege schrittweise angezeigt.

**Unittests**
Durch Aufruf von *all_unit_test.py* werden alle in 4.3.4 angegebenen Unittests durchlaufen und auf mögliche Fehler hingewiesen.

**Docker**
Das Dockerfile nutzt als Baseimage *tensorflow/tensorflow:2.11.0*, welches über die öffentliche Registry *hub.docker.com* bezogen werden kann. Dieses Image bietet bereits die meisten Pakete, die für das Projekt benötigt werden.

Die noch nicht vorinstallierten Pakete werden während des Builds nachinstalliert. Die Pakete werden in der Datei *requirements.txt* aufgelistet, welche sich im selben Ordner wie das Dockerfile befinden muss.

Als Entrypoint nutzt das Image den Befehl python *solve_game.py* solve. Dieses Skript erwartet über die Standardeingabe ein JSON-String, versucht das Problem zu lösen und gibt, ebenfalls über die Standardausgabe, die Lösung als JSON aus.
Um das Image erfolgreich zu bauen, muss folgende Ordnerstruktur vorliegen:

- Dockerfile
- requirements.txt
- informaticup-profit
  - solve_game.py
  - ...

Mit

```
docker build . --tag <tag_name>
```

lässt sich ein Build triggern.
Um einen Container zu starten, wird folgender Befehl genutzt:

```
docker run -i --rm --network none --cpus 2.000 --memory 2G
 --memory-swap 2g <tag_name>
```

Im Anschluss erwartet die Standardeingabe eine Aufgaben im JSON-Format. Das Programm errechnet eine Lösung und gibt diese bei der Standardausgabe als Liste von platzierbaren Objekten zurück. Dies kann mit einem großen Spielfeld mehrere Minuten dauern.
Alternativ lässt sich eine Eingabe auch direkt an den Container geben, indem Pipes verwendet werden.

```
echo ‘{"width":40,"height":20,"objects": ... }’ | docker run
-i --rm --network none --cpus 2.000 --memory 2G --memory-swap 2g <tag_name>
```

# 6 Evaluation und Diskussion

Grundsätzlich lässt sich sagen, dass das Ergebnis des Projekts positiv zu bewerten ist. Ein Großteil der Aufgaben wird gelöst, einige auch mit sehr hoher Punktzahl und geringer Rundenzahl.

Zum Evaluieren des Programms wurden 23 Tasks verwendet. Vier davon sind vom InformatiCup 2023 zur Verfügung gestellt worden, der Rest wurde selbst erstellt. Diese wurden in zwei Kategorien aufgeteilt: einfach und schwer. Die einfachen Aufgaben hatten nur ein einzelnes Produkt definiert, die schweren mehrere. Eine Tabelle mit den Ergebnissen befindet sich im Anhang in Kapitel 8.1. Es wird dabei bewertet, wie nah das Programm an den optimalen Score herangekommen ist. Hierbei muss jedoch festgehalten werden, dass es nicht immer möglich ist, die optimale Punktzahl zu erreichen. Der Optimal Score soll nur einen Hinweis liefern, wie die Punktzahl interpretiert werden kann.

Von den 23 Aufgaben konnte das Programm bei 21 mindestens ein Produkt produzieren und somit Punkte erzielen. Die beiden nicht lösbaren Aufgaben sind die Aufgaben 8 und 18, welche sich im Anhang im Abschnitt 8.2 befinden.
Die Aufgabe 8 besteht aus einer Art Zick-Zack-Weg, durch den Förderbänder gelegt werden müssen. Der problematische Teil ist in Abbildung 14 im Anhang mit einem roten Rahmen markiert. Das Program versucht so schnell wie möglich nach rechts (in Richtung der Fabrik) zu gehen, wodurch es in den Aussparungen der Hindernisse läuft. Dieser falsche Schritt kann dann nicht mehr rückgängig gemacht werden. Die Abbildung 12 soll zeigen, wie dieser falsche Schritt zur nicht Lösung des Problems führen konnte. Da das Programm versucht, immer möglichst direkt in die Richtung der Fabrik zu laufen, wurde diese Konstruktion in der Aufgabe dem Programm zum Verhängnis.

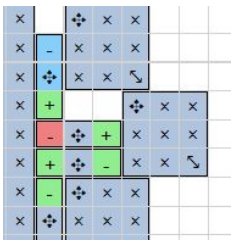

Abbildung 12: Problematisch Teil der Aufgabe 8

Die zweite nicht lösbare Aufgabe mit der Nummer 18 kann nur durch eine sehr spezifische Gebäude-Kombination gelöst werden. Die Fabrik kann nur an einer Position gesetzt werden, ebenso wie die Minen zu den Lagerstätten. Während die Verbindung von Lagerstätte 0 zur Fabrik mehrere Förderband Kombinationen zulässt, muss die Verbindung von Lagerstätte 1 zur Fabrik mit einem Förderband des Subtypen 4 erstellt werden. Wird vom Agenten fälschlicherweise der Subtyp 1 gewählt, was eine legale Aktion ist, so ist es nicht mehr möglich, die Fabrik mit Förderbänder zu erreichen.

Hier ist das Hauptproblem, dass die zwei untergeordneten Agenten nicht miteinander kommunizieren. Der erste Agent erreicht sein Ziel, ohne dabei zu beachten, dass dabei der Weg für den zweiten Agenten versperrt wird. Das gemeinsame Ziel, Produkte zu produzieren, schlägt fehl.

In einzelnen Aufgaben lässt sich außerdem eine suboptimale-Strategie des untergeordneten Agenten feststellen. Wenn sich die Fabrik direkt hinter einem Hindernis befindet, weiß der untergeordnete Agent nicht in welche Richtung er laufen soll. Dies führt dazu, dass eine nicht-optimale Schleife mit überkreuzten Förderbändern entstehen kann (vgl Abb. 13, entnommen aus Lösung der Aufgabe 15). Der Agenten kann lokal gute Entscheidungen treffen, ihm fehlt jedoch der große Überblick über die Gesamtaufgabe.

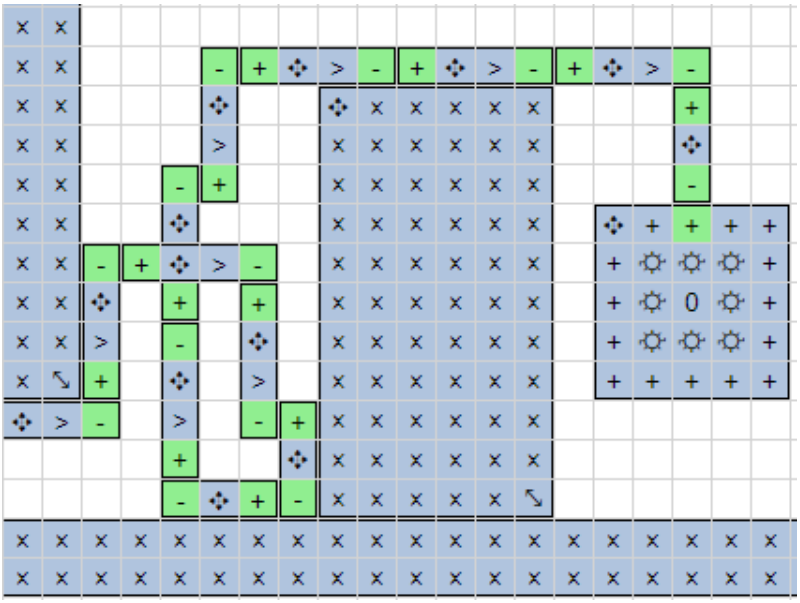

Abbildung 13: Darstellung einer Förderband-Schleife

In den 21 gelösten Aufgaben konnte 11 mal der Optimal Score tatsächlich erreicht werden. Besonders hervorzuheben ist die erste der vorgegebenen Aufgaben. Hier konnte das Programm nicht nur den Optimal Score erreichen, sondern auch eine geringere Rundenanzahl erzielen, als die von den Teammitgliedern erstellte manuelle Lösung. Das Programm erreichte 410 Punkte mit 26 Runden, während die manuelle Lösung 410 bei 43 Runden erzielte. Wenn das Programm eine Lösung gefunden hat, versucht es diese Lösung weiter zu optimieren, um die Punktzahl zu erhöhen und die Rundenzahl zu verringern. Es wird versucht mehrere Minen zu platzieren und diese mit bereits bestehenden Förderbandverbindungen zu verbinden. Dadurch konnte das Programm eine sehr gute Leistung bei dieser Aufgabe erzielen. Die gelöste Aufgabe wird im Anhang in 8.3 in der Abbildung 16 dargestellt.

Des Weiteren soll die Aufgabe 22 hervorgehoben werden. Diese hat ebenso die bestmögliche Punktzahl erreicht. Die Aufgabe wird im Anhang in Abschnitt 8.3 in Abbildung 17 dargestellt. Die beste Lösung für diese Aufgabe zu finden, ist aus

menschlicher Sicht nicht intuitiv. Es gibt viele Produkte, die die gleichen Ressourcen benötigen. Der naive Ansatz, alle gegebenen Produkte zu produzieren, ist bei dieser Aufgabe nicht der beste Weg. Die einzelnen Produkte müssen sich Ressourcen teilen, daher ist die beste Variante nur das Produkt 3 zu produzieren, da mit diesem Produkt die meisten Punkte erzielt werden können. Der Optimal Score berechnet neben der besten Punktzahl auch die beste Produktkombination aus. Mit diesem Wissen kann der Agent bei vielen Produkten die beste Produktkombination auswählen, was bei Aufgabe 22 dazu führt, dass nicht einmal versucht wird, die Produkte 0-2 herzustellen. Das Produzieren von Produkt 3 führt zur besten Punktzahl. Die Aufgabe 22 wurde als optionale Beispielaufgabe abgegeben.

In 17 Aufgaben wurde eine Punktezahl erreicht, die 50% oder höher als die jeweiligen Optimal Scores waren. Nur in 6 Fällen lag die erspielte Punktezahl darunter. Durchschnittlich wurden 70% des Optimalen Scores erreicht, was als gutes Ergebnis bewertet werden kann.

Der Zeitaspekt in der Lösung wird zwar beachtet, aber nicht aktiv bei den erstellten Lösungen überprüft. Der Grund dafür ist, dass die Implementierung, die die Zeit berücksichtigt, erst spät hinzugefügt wurde. Beim Erstellen der Tasks wurde daher auch diesem Punkt wenig Beachtung geschenkt. Weshalb ein Standardwert von 300 Sekunden in allen Tasks verwendet wurde. Dieser Wert konnte von 22 der 23 Aufgaben eingehalten werden, nur die Aufgabe 21 hat diesen Wert mit 395,25 Sekunden überschritten. Aufgabe 21 ist ein Spielfeld mit Größe $100 \times 100$, was dem größtmöglichen Spielfeld entspricht. Außerdem umfasst diese Aufgabe große Lagerstätten, wodurch es viele möglich platzierbare Minen gibt. Allgemein kann beobachtet werden, dass größere Spielfelder auch mehr Zeit zum Lösen brauchen.

Durchschnittlich hat das Programm für alle Aufgaben 31 Sekunden. Wenn die Aufgabe 21 in dieser Rechnung ausgeschlossen wird, reduziert sich dieser Wert auf 14,4 Sekunden.

# 7 Fazit

Wie in Kapitel 3 beschrieben, gab es sehr viele Ideen und Ansätze, wie die Aufgabe des InformatiCup 2023 gelöst werden könnte. Leider konnte in den drei Monaten, die für diese Aufgabe Zeit war, nicht alles umgesetzt und ausprobiert werden. Zeitbedingt konnten daher Ansätze wie AC oder MCST nicht erfolgreich implementiert werden. Ein Großteil der Zeit wurde für die fehlerfreie Implementierung der "Profit"-Umgebung investiert, was zu einem Zeitdruck am Ende des Projekts führte. Da der untergeordnete Agent auf diese Umgebung trainiert wurde, musste sichergestellt werden, dass diese korrekt implementiert wurde.
Der Aufwand, eine ML-Lösung zu entwickeln, wurde am Anfang des Projekts unterschätzt. Es ist daher möglich, dass ein regelbasierter "Pathfinding"-Ansatz für den untergeordneten Agenten weniger zeitintensiv zu entwickeln gewesen wäre.
Mit mehr Zeit wäre es möglich gewesen, die Lösungen des Programms noch weiter zu optimieren, sodass Aufgaben, die nicht oder nicht gut gelöst wurden, ebenfalls eine gute Punktzahl erreichen. Der Brute-Force Ansatz, der für die Positionierung der Fabrik gewählt wurde, könnte durch ein gezieltes Platzieren verbessert werden, so dass der untergeordnete Agent kürzere Wege zur Fabrik hat. Auch das Platzieren von mehreren Fabriken für ein Produkt wird in unsere Implementierung noch nicht umgesetzt.

Wie sich zeigt, gibt es noch Potenzial, die Lösung des Projekts zu verbessern. Manuell erstellte Lösungen sind zwar in vielen Fällen gleich gut oder sogar besser, trotzdem konnte das Programm menschlich erstellte Lösungen schlagen.

Die Evaluation zeigt, dass es deutlich mehr gute und sehr gute Ergebnisse gab als schlechte. Fast die Hälfte der Aufgaben hat die bestmögliche Punktzahl erreicht, was eine sehr gute Statistik ist, wenn beachtet wird, dass der optimale Score nicht immer erreicht werden kann. Zusammenfassend sehen wir in unseren Ansätzen weiteres Potenzial, sind mit den Ergebnissen aber dennoch zufrieden.

Das Team "Die Schmetterlinge" bedankt sich bei der Gesellschaft für Informatik für das Ausrichten des InformatiCups 2023, bei Prof. Markus Schneider für die Betreuung des Projektes und ist gespannt, welche Lösungen die anderen Teams präsentieren werden.

# Literaturverzeichnis

[Fou] Foundation, Python S.: *unittest — Unit testing framework*. https://docs.python.org/3/library/unittest.html

[KLM96] Kaelbling, Leslie P. ; Littman, Michael L. ; Moore, Andrew W.: Reinforcement learning: A survey. In: *Journal of artificial intelligence research* 4 (1996), S. 237–285

[KT99] Konda, Vijay ; Tsitsiklis, John: Actor-Critic Algorithms. In: Solla, S. (Hrsg.) ; Leen, T. (Hrsg.) ; Müller, K. (Hrsg.): *Advances in Neural Information Processing Systems* Bd. 12, MIT Press, 1999

[Oe15] Oh, Junhyuk ; et. al: Action-Conditional Video Prediction using Deep Networks in Atari Games. In: Cortes, C. (Hrsg.) ; Lawrence, N. (Hrsg.) ; Lee, D. (Hrsg.) ; Sugiyama, M. (Hrsg.) ; Garnett, R. (Hrsg.): *Advances in Neural Information Processing Systems* Bd. 28, Curran Associates, Inc., 2015

[SB20] Sutton, Richard S. ; Barto, Andrew G.: *Reinforcement learning: An introduction*. 2. MIT press, 2020. – ISBN 9780262039246

[Se16] Silver, David ; ; et. al: Mastering the game of Go with deep neural networks and tree search. In: *nature* 529 (2016), Nr. 7587, S. 484–489

[Se17] Silver, David ; ; et. al: Mastering the game of Go without human knowledge. In: *nature* 550 (2017), Nr. 7676, S. 354–359

[WD92] Watkins, Christopher J. C. H. ; Dayan, Peter: Q-Learning. 8 (1992), S. 279–292. http://dx.doi.org/https://doi.org/10.1007/BF00992698. – DOI https://doi.org/10.1007/BF00992698

# 8 Anhang

## 8.1 Ergebnisse der Evaluierung

| Aufgabe | Optimaler Score | Erreichter Score | Runden | Zeit in s | Score in % |
|---|---|---|---|---|---|
| 001.task.json | 410 | 410 | 26 | 7,49 | 100 |
| 002.task.json | 120 | 90 | 20 | 0,1 | 75 |
| 003.task.json | 60 | 30 | 14 | 6,7 | 50 |
| 004.task.json | 720 | 120 | 47 | 41,04 | 16,67 |
| 005.task.json | 2220 | 1050 | 50 | 30,06 | 47,3 |
| 006.task.json | 40 | 40 | 8 | 0,23 | 100 |
| 007.task.json | 200 | 80 | 29 | 0,61 | 40 |
| 008.task.json | 520 | 0 | 0 | 2,86 | 0 |
| 009.task.json | 490 | 490 | 46 | 21,13 | 100 |
| 010.task.json | 180 | 180 | 20 | 0,3 | 100 |
| 011.task.json | 280 | 280 | 31 | 0,82 | 100 |
| 012.task.json | 620 | 620 | 50 | 14,65 | 100 |
| 013.task.json | 300 | 300 | 44 | 2,05 | 100 |
| 014.task.json | 400 | 280 | 100 | 0,4 | 70 |
| 015.task.json | 450 | 240 | 62 | 4,48 | 53,33 |
| 016.task.json | 1030 | 970 | 100 | 93,16 | 94,17 |
| 017.task.json | 1030 | 780 | 100 | 24,67 | 75,73 |
| 018.task.json | 170 | 0 | 0 | 0,27 | 0 |
| 019.task.json | 40 | 40 | 16 | 0,71 | 100 |
| 020.task.json | 150 | 150 | 37 | 5,36 | 100 |
| 021.task.json | 8340 | 570 | 99 | 395,25 | 6,83 |
| 022.task.json | 900 | 900 | 63 | 12,74 | 100 |
| 023.task.json | 280 | 280 | 38 | 46,75 | 100 |

Tabelle 4: Ergebnisse der Evaluierung in Tabellenform

## 8.2 Nicht lösbare Aufgaben

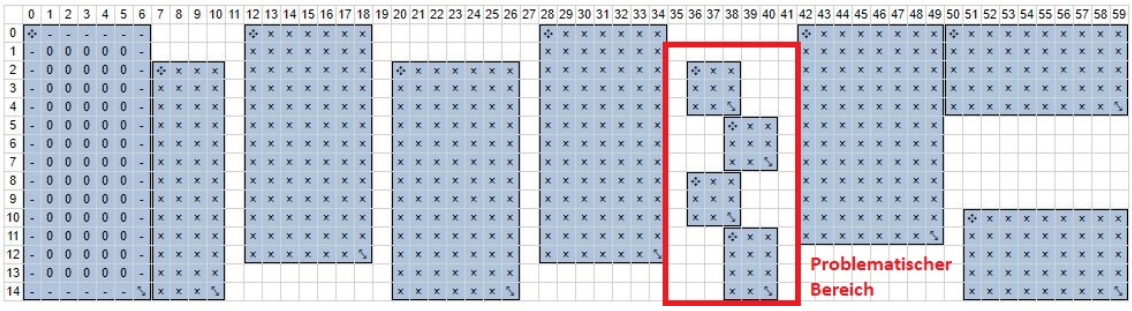


Abbildung 14: Nicht lösbare Aufgabe 8

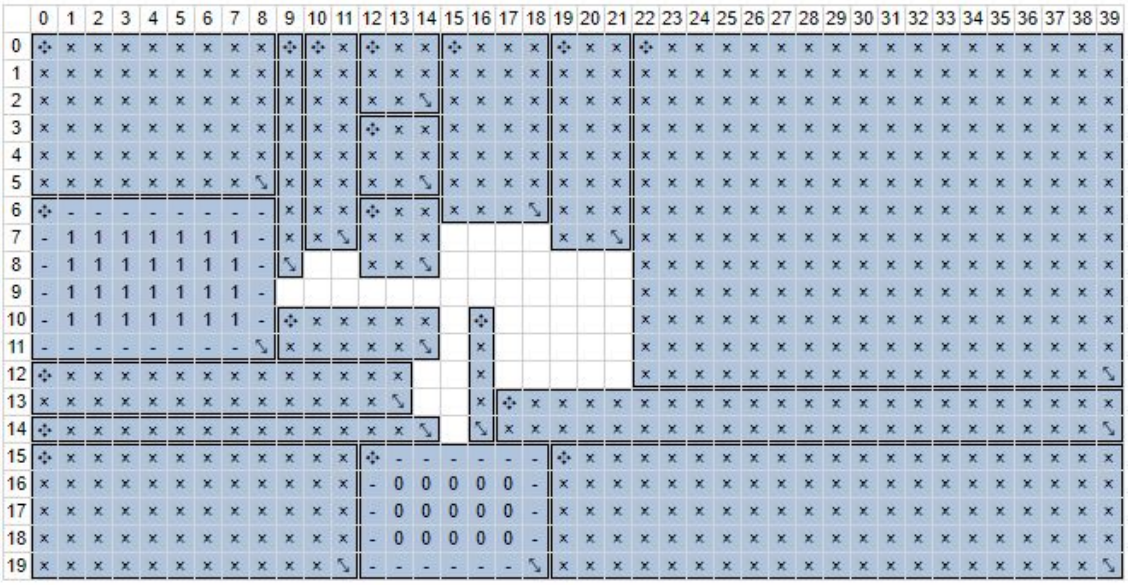

Abbildung 15: Nicht lösbare Aufgabe 18

## 8.3 Sehr gut gelöste Aufgaben

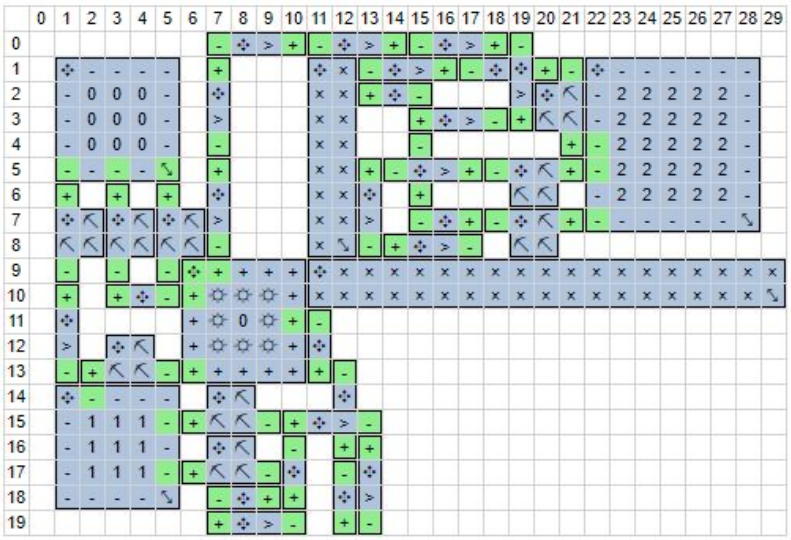

Abbildung 16: Sehr gut gelöste Aufgabe 1

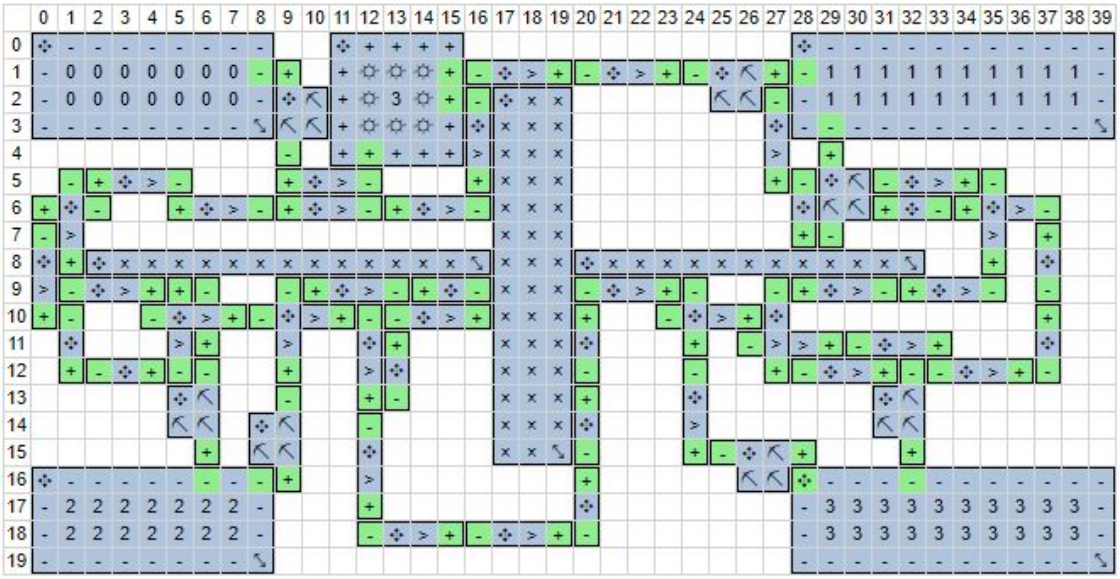

Abbildung 17: Sehr gut gelöste Aufgabe 22