Informaticup 2023

Profit

von

Lisa Binkert, Leopold Gaupe, Yasin Koschinksi

RWU Hochschule Ravensburg-Weingarten University of Applied Sciences

Abgabedatum: 15.01.2023

Inhaltsverzeichnis

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 University of Applied Science im Studiengang Master Informatik mit dem Profil Künstliche Intelligenz und Autonome Roboter eingeschrieben sind.

Im Kapitel 2 wird die Aufgabenstellung des Informaticup 2023 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.

2 Aufgabenbeschreibung

Die Aufgabe des InformatiCups 2023 ist das Lösen und Optimieren des rundenbasierten 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, wobei das Ziel das Maximieren dieser Punkte ist.

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 100x100 großen Rastergitter. Ein Raster ist entweder leer oder durch ein Objekt besetzt.

Die Abbildung reffig:task1 zeigt ein mögliches Spielfeld. In diesem Beispiel ist das Feld 30x20 groß und enthält drei Lagerstätten mit den Ressourcen Subtyp 0, 1 und 2, sowie zwei Hindernisse.

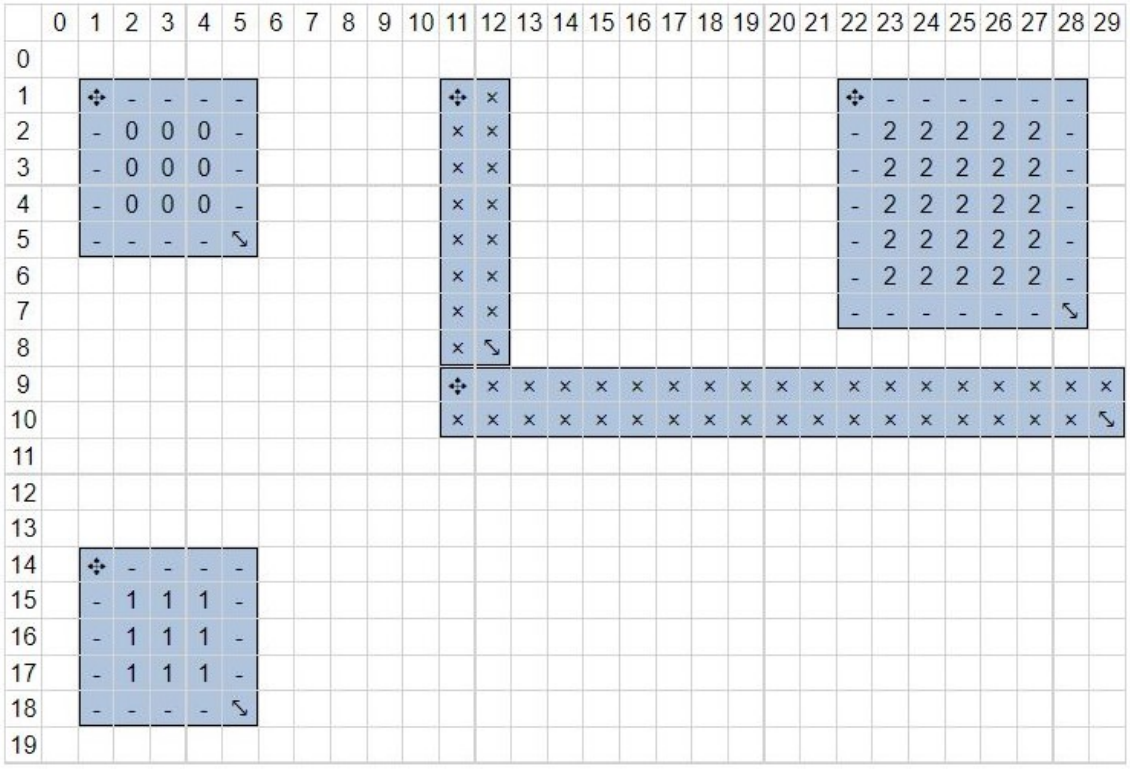

Abbildung 1: Beispielumgebung eines Profit-Spiels

Das linke obere Raster befindet sich an der Stelle (0,0), die rechte untere (Breite-1, Höhe-1). Die Größe des Spielfeldes ist pro Spiel vorgegeben.

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 Menge an Ressourcen einer Lagerstätte wird durch die Größe der Lagerstätte festgelegt. Diese wird durch die Menge an Rastern x 5 festgelegt. Ist also eine Lagerstätte $3 \cdot 3$ dann enthält es $3 \cdot 3 \cdot 5 = 45$ Ressourcen eines bestimmten Subtyps. Insgesamt gibt es 8 Subtypen (0-7).

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 4x2 oder 2x4 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

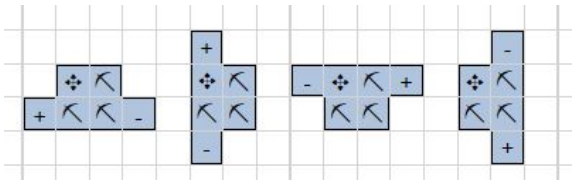

Abbildung 2: Darstellung der vier Minen-Subtypen

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.

Förderband
Um Ressourcen von einer Lagerstätte zu einer Fabrik zu befördern, können Förderbänder genutzt werden. Förderbänder sind nicht zwangsläufig notwendig. Sie stellen zusätzliche Verbindungsstücke zwischen Mine und Verbinder, Mine und Fabrik oder Verbinder und Fabrik dar.

Diese haben wie andere Objekte auch einen Eingang (+) und einen Ausgang (-). Ein Förderband ist entweder 3 oder 4 Raster lang und kann, wie in der Abbildung 3 gezeigt, je in vier Subtypen mit unterschiedlicher Ausrichtung verwendet werden. Somit hat das Förderband insgesamt acht Subtypen. Im Gegensatz zu allen anderen

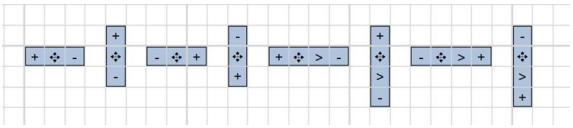

Abbildung 3: Darstellung der acht Förderband-Subtypen

Objekten dürfen sich Förderbänder auch kreuzen.

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).

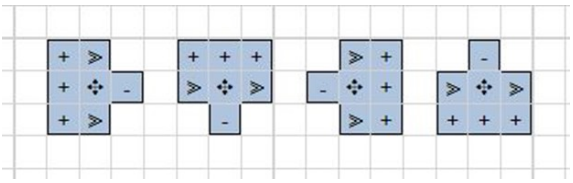

Abbildung 4: Darstellung der acht Förderband-Subtypen

Fabrik
Für jede Fabrik ist definiert, wie viele und welche Ressourcen gebraucht werden, um ein Produkt herzustellen. Da es maximal acht Produkte geben kann, gibt es von der Fabrik insgesamt acht verschiedene Subtypen, für jedes Produkt einen. Jede Fabrik ist 5x5 groß. Die äußeren Raster 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.

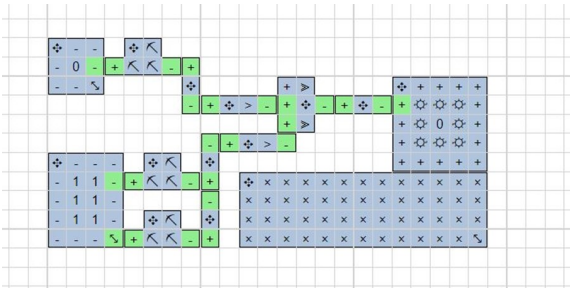

Abbildung 5: Darstellung der acht Förderband-Subtypen

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 folgende Tabelle definiert

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 da sind
Lagerstätte	-	Gibt bis zu 3 Ressourcen an jeden Eingang einer Mine

Tabelle 1: Beschreibung der Aktionen "Beginn der Runde"und "Ende der Runde"

die Aktionen der einzelnen Objekte. Das Spielfeld hat neben der Feldgröße und den vorhandenen Objekten auch das Attribut "turns". Dieses Attribut gibt die maximal erlaubte Anzahl an Spielrunden an.

Pro Runde werden Ressourcen von Lagerstätten abgebaut und Stück für Stück über die gebauten Förderbänder und gegebenenfalls Verbinder zu einer Fabrik befördert. Wenn eine oder alle Ressourcen eines Produktes abgebaut wurden und damit kein weiteres Produkt mehr produziert werden kann, stoppt die Produktion dieser Fabrik. 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 Ziel des Spiels ist es, die Punktezahl zu maximieren und die Rundenanzahl zu minimieren.

2.3 Darstellung in JSON

Der Input des Spiels erfolgt im JSON-Format, in dem Lagerstätten, Hindernisse und Produkte definiert sind. Das JSON für das Beispiel in Abbildung 6 sieht wie folgt aus: Das Spielfeld wird durch die angegebene Breite (width) und Höhe(height) de-


```
{"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

finiert. Es enthält fünf verschiedene Objekte (objects), 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

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

Tabelle 2: Beschreibung der JSON Struktur einzelner Objekte

(ressources) pro Produkt und die Anzahl an Punkten (points) pro Produkt angegeben. Die Position der Ressource im Array bestimmt den jeweiligen Subtyp.

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

3 Lösungsansätze

Mit AlphaGo und später AlphaZero hat das Unternehmen Deepmind gezeigt, dass ein Computer durch Deep Reinforcement Learning lernen kann, hochkomplexe Spiele wie Go und Schach auf professionellem Niveau zu spielen. [Se16] Schach und Go besitzen ein festes 8x8 bzw. 19x19 Spielfeld, wohingegen das zu lösende Spiel "Profit!" ein bis zu 100x100 Feld umfasst. Es ist jedoch von der Spielweise weniger komplex und sollte deshalb ebenfalls mit verschiedenen Reinforcement Learning Methoden lösbar sein. In den folgenden Abschnitten werden mögliche Lösungsansätze vorgestellt.

3.1 Reinforcement Learning

In dieser Arbeit bietet sich Reinforcement Learning (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 sieht folgendermaßen aus:

- Set an Umgebungs-Zuständen S (states)
- Set an Agenten-Aktionen A(actions)
- Set 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. Durch die Aktion kann er in einen neuen Zustand kommen. Mathematisch kann dies folgendermaßen ausgedrückt werden:

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

Ziel für den Agenten ist es, so viele Belohnungen wie möglich zu bekommen. Die Abbildung **??** soll den 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 [**?** , S.237ff]
In dieser Arbeit stellt das Spielfeld die Umgebung dar. Es kann maximal 100x100 Raster 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.
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.

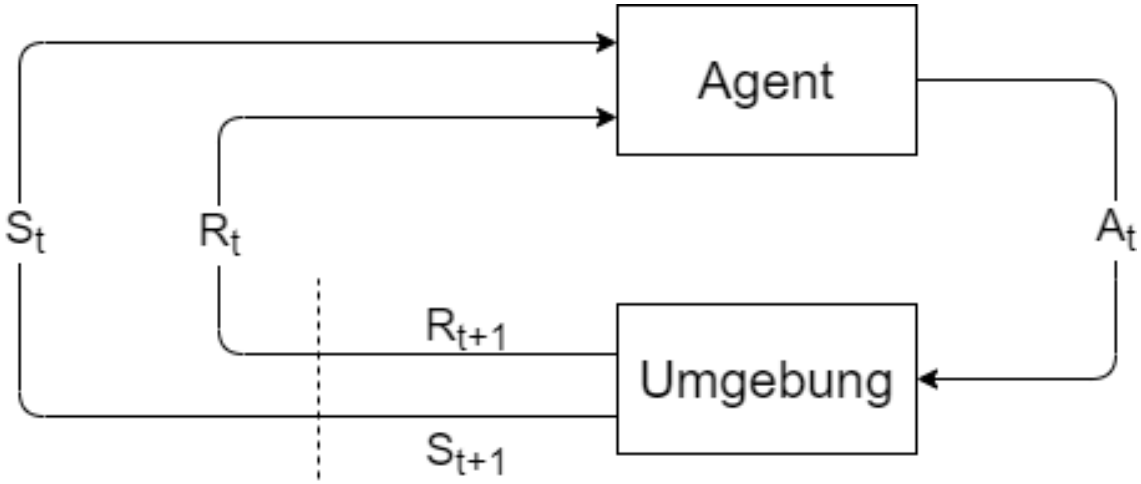

Abbildung 7: Diagramm der Interaktion des Agenten mit der Umgebung [SB20, S.48]

Das Set an Belohnungen, also die Rewards kann frei gewählt werden und ist nicht vom Spiel vorgegeben. Es macht Sinn den Agenten zu belohnen, wenn er Produkte erstellen kann und zu bestrafen, wenn er illegale Aktionen ausführen möchte
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.

3.2 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.

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

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 γ eingeführt, mit $0 <= \gamma <= 1$.

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

Spätere Rewards werden dann weniger beachtet 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, auch state-value-function genannt, gibt an, wie gut ein State s ist, wenn der Agent der policy π folgt.

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

Die action-state-function q, auch q-function, bestimmt wie gut eine Aktion q ist, wenn sich der Agent in state s befindet und der policy π 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. [**?** , 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. [**?** , S.281f]

Um zu bestimmen, ob eine Policy besser ist als eine andere, werden die v werde angeschaut. Eine Policy π ist besser als eine Policy π' 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] :

$$v_\pi * (s) = max E[R_{t+1} + \gamma v_\pi * (S_{t+1})|S_t = s, A_t = a] \tag{6}$$

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

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) 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 ϵ-greedy Methode umgesetzt. In dieser Methode gibt es eine ϵ 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 ϵ 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. [**?** , 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. [**?**]
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-function. Häufig wird die state-value-function vom Critic gelernt. [SB20, S.321]

3.4 Mone-Carlo-Search-Tree

Dieser Ansatz des Mone-Carlo-Search-Tree MCST entspricht grundlegend der Umsetzung dieses Papers [Se16] , angepasst auf unser Problem.

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 Environment 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 Parent in diesen Zustand zu gelangen
- Die Wahrscheinlichkeit, dass ausgehend vom Parent 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 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 unserer Implementation 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 gesamplet, 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 -¿ 1) oder nicht (verloren -¿ -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 nutzen wir den Cross Entropy loss.
Der Critic wiederum soll im gegebenen Zustand möglich genau vorhersagen können, ob das Modell das Spiel gewinnt oder verliert. Hierfür nutzen wir den Mean Squared Error (MSE) zwischen der Vorhersage des Modells und des tatsächlichen Outcomes, 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 Predictions durch das Modell scheinen nach einer Analyse der Laufzeit am aufwändigsten zu sein und bremsen das Sammeln von Trainingsdaten so stark, dass wir uns gegen diesen Ansatz entscheiden mussten.

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 irgendwie 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 evt für andere Produkte gebraucht werden. Daher ist es schwierig eine gute Metrik zu finden, wie idealerweise eine Fabrik gebaut werden soll. Häufig ist auch die Anzahl der möglichen Fabrik-Positionen auch auf wenige begrenzt.

Ein regelbasierter Algorithmus würde, um ein Produkt zu lösen, als erstes die Fa-

brik und die Minen an den Lagerstätten zu setzen und anschließend versuchen diese miteinander zu verbinden. Die Auswahl, wann welches Produkt gebaut werden soll und welche überhaupt gebaut werden sollen, würde sich errechnen lassen.

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 Lösungsumsetzung

4.1 Programmiersprache und Bibliotheken

Das Projekt wurde in der Programmiersprache Python umgesetzt. Diese hat einen einfache und übersichtliche Syntax, sowie große Unterstützung von vielen Bibliotheken im Bereich des Maschinellen Lernens.
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 Reinforcement Learning Agent mit unserer Implementation der ”Profit!Umgebung.
Für einen einheitlichen Code wurden folgende Code Konventionen eingehalten.

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. Wir haben diese Umgebung in Python nach implementiert, um die volle Kontrolle über die Spieldynamik zu erhalten. Somit können Reinforcement Learning Agenten uneingeschränkt ihre Umgebung abfragen.

Das Spiel hat in seiner Grundform aufgrund des bis zu 100x100 großen Spielfeldes einen sehr großen Zustands- und Aktionsraum, weshalb wir uns dafür entschieden haben, das Problem nach dem Teile-und-Herrsche-Prinzip in mehrere Teilprobleme zu zerlegen.

Um eine geeignete Abstraktion zu finden, haben wir uns daran orientiert wie ein Mensch das “Profit”-Spiel spielen 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 haben wir uns dafür entschieden, eine Mine als Startgebäude des Agentens anstatt einer Lagerstätte zu verwenden. Somit können wir die Position des Agenten auf den einzigen Ausgang des zuletzt platzierten Gebäudes legen. 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 Reinforcement Learning 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 **??** beschriebene Hindernisse hinzufügen. Eine fehlerhafte Umgebung könnte dafür sorgen, dass die Lösung des Spiels fehlerhaft ist. Daher war wichtig, dass diese Umgebung mit allen dazugehörigen Klassen korrekt umgesetzt wurde, weshalb Unittests diese getestet haben.

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

4.3.1 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.

Eine Aufgabe kann entweder aus einer JSON-Datei oder einem übergebenen JSON-String generiert werden. Gelöste Aufgaben werden als JSON-Datei unter /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.

4.3.2 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 am Egress 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.

4.3.3 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 Produkt-

kombination 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 Lagerstätte 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*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.

4.3.4 Testing

Der Code wurde mit dem Unit Test Framework unittest von Python getestet. Dieses bietet eine einfache Anwendung 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.[**?**] Die Tests und die Implementierung des Codes wurden parallel ausgeführt von unterschiedlichen Entwicklern. Damit soll 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 bauen Gebäude auf legale und illegale Weise in eine Umgebung und prüfen, ob diese das neue Gebäude korrekt akzeptiert bzw. verwirft. Beispielsweise darf ein Raster 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. Außerdem wurden Funktionieren zum Platzieren von Gebäuden

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.

Optimal Score
Der Optimal Score wurde anhand der gegebenen Aufgaben überprüft. Dabei wurde für jede dieser vier Ausgaben von Hand ausgerechnet, was bestenfalls rauskommen kann, was anschließend mit dem Ergebnis des Algorithmus abgeglichen wurde.

4.4 Untergeordneter Agent

4.5 Übergeordnete Agent

4.6 Wartbarkeit

5 Benutzerhandbuch

6 Evaluation

7 Fazit

Literaturverzeichnis

[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