

DIPLOMARBEIT

Einsatz von LiDAR im autonomen Fahren

Ausgeführt im Schuljahr 2023/24 von:

Philip Fenk, 5AHIF-03
Emilio Zottel, 5AHIF-22
Marco Molnár, 5AHIF-10
Adrián Kalapis, 5AHIF-09

Betreuer:

Dipl.-Ing. Christoph Schreiber
Dipl.-Ing. Wolfgang Raab

St. Pölten, am 24. März 2024

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche erkenntlich gemacht habe.

Philip Fenk

Emilio Zottel

Marco Molnár

Adrian Kalapis

St.Pölten, am 24. März 2024

Diplomandenvorstellung



Philip FENK

Geburtsdaten:
28.08.2005 in St. Pölten

Wohnhaft in:
Kressgasse 5
3040 Neulengbach

Werdegang:
2019 - 2024:
HTBLuVA St.Pölten, Abteilung für Informatik
2015 - 2019:
BRG/BORG St. Pölten
2011 - 2015:
Volksschule Neulengbach

Kontakt:
philip.fenk@gmail.com



Emilio ZOTTEL

Geburtsdaten:
11.05.2005 in St. Pölten

Wohnhaft in:
Waldstraße 8
3061 Schönenfeld

Werdegang:
2019 - 2024:
HTBLuVA St.Pölten, Abteilung für Informatik
2015 - 2019:
Neue Mittelschule Neulengbach
2011 - 2015:
Volksschule St. Christophen

Kontakt:
emilio.zottel@gmail.com



Marco MOLNÀR

Geburtsdaten:
02.01.2005 in Lilienfeld

Wohnhaft in:
Josef-Reither Straße 17a
3430 Tulln an der Donau

Werdegang:
2019 - 2024:
HTBLuVA St.Pölten, Abteilung für Informatik
2015 - 2019:
Bundesgymnasium Tulln
2011 - 2015:
Volksschule Asperhofen

Kontakt:
mmarco.molnar@gmail.com



Adrián KALAPIS

Geburtsdaten:
17.06.2003 in Pancevo

Wohnhaft in:
Waldbachstraße 2/1
3041 Siegersdorf

Werdegang:
2019 - 2024:
HTBLuVA St.Pölten, Abteilung für Informatik
2015 - 2019:
NMS Neulengbach
2010 - 2015:
Grundschule Zarko Zrenjanin

Kontakt:
kalapis.adrian03@gmail.com

Danksagungen

Philip Fenk

Anfangs möchte ich meine aufrichtige Dankbarkeit gegenüber meinen Klassenkollegen zum Ausdruck bringen, die durch ihre einzigartigen Persönlichkeiten die Schulzeit erträglicher gemacht haben. Zudem sind sich auch Freundschaften mit einigen entstanden, für die ich sehr dankbar bin.

Besonders möchte ich mich aber bei meinen Eltern und meiner Schwester bedanken, die mich ermutigt haben, die Schule durchzuziehen und nicht abzubrechen. Zudem haben sie mich auch während meiner gesamten schulischen Laufbahn unterstützt und motiviert.

Ein weiterer Dank gebührt Dipl.-Ing. Christoph Schreiber für seine Betreuung und Beratung während dem gesamten Prozess der Diplomarbeit.

Zuletzt möchte ich mich bei Frau Mag. Heidemarie Reichhart, meinem Klassenvorstand, bedanken, die sich die 5 Jahre kontinuierlich für meine Klasse eingesetzt hat, immer hinter uns stand und durchgehend ein offenes Ohr hatte.

Emilio Zottel

An dieser Stelle möchte ich mich bei allen bedanken, die mich während der Erstellung meiner Diplomarbeit unterstützt haben.

Mein besonderer Dank gilt meinen Eltern, die mir stets zur Seite gestanden sind und ohne die ich es nicht so weit geschafft hätte.

Auch danke ich meiner Freundin Johanna für ihr Verständnis und ihre unaufhörliche Unterstützung, besonders in den stressigen Phasen meines Lebens.

Abschließend möchte ich mich sowohl bei meiner Klassenvorständin Mag. Heidemarie Reichhart für ihren kontinuierlichen Beistand als auch bei Herrn Dipl.-Ing. Christoph Schreiber für seine Freundlichkeit und hilfreichen Tipps bedanken.

Ohne die Unterstützung all dieser Personen wäre ich jetzt nicht dort, wo ich bin.

Marco Molnár

Ich möchte mich bei einigen besonderen Menschen bedanken, die mich während meiner Schullaufbahn unterstützt haben.

Erstens möchte ich mich bei meiner Mutter bedanken, die mich immer ermutigt hat und trotz schwieriger Zeiten und hauptsächlicher Alleinerziehung mir immer alle Möglichkeiten geboten hat, ohne größere Sorgen, meine Ziele zu verfolgen.

Ein herzliches Dankeschön geht auch an meine Freundin, die mit ihrer unermüdlichen Unterstützung und Liebe an meiner Seite war. Sie hat mir jeden Tag die nötige Motivation und Kraft gegeben, um weiterzumachen.

Meiner Familie möchte ich ebenfalls danken, für ihre Ermutigung und Förderung, die mir geholfen haben, meinen Weg zu gehen.

Ein weiterer Dank gilt meinen Freunden und Klassenkameraden, die 5 Jahre lang an der HTL mit mir gelitten haben. Eure Freundschaft, euer Humor und euer Zusammenhalt haben meine Schulzeit um Einiges erträglicher gemacht.

Letztens bedanke ich mich bei meinem Betreuungslehrer, für diese Diplomarbeit, Dipl. Ing. Christoph Schreiber, und unserer Klassenvorständin, Mag. Heidemarie Reichhart.

Von ganzem Herzen danke ich euch allen für eure bedeutsame Rolle in meinem Leben während meiner Schulzeit. Eure Unterstützung hat mich zu dem Menschen gemacht, der ich heute bin.

Adrián Kalapis

Ich möchte mich an dieser Stelle bei all denjenigen bedanken, die mich während der Erstellung dieser Diplomarbeit und während meiner gesamten schulischen Laufbahn motiviert und unterstützt haben.

Ebenfalls möchte ich mich bei Herrn Dipl.-Ing. Christoph Schreiber für die Betreuung und Unterstützung bei meiner Diplomarbeit bedanken. Zusätzlich möchte ich mich auch bei der Frau Mag. Heidemarie Reichhart für ihre Unterstützung als Klassenvorständin bedanken.

Ich möchte auch meinen Gruppenmitgliedern danken, denn ohne deren Kooperation, Wissen und Engagement wäre diese Arbeit nicht realisiert worden.

Letztens möchte ich mich bei meinen Freunden und Klassenkameraden bedanken, denn ohne sie wäre ich völlig durchgedreht!

Zusammenfassung

Einsatz von LiDAR im autonomen Fahren:

Das Ziel dieser Diplomarbeit ist es, ein Modellauto mit Hilfe eines LiDAR-Sensors zum autonomen Fahren zu bringen. Dazu wurde ein Modell in der Simulationsumgebung CARLA trainiert.

Zu Beginn wird ein Überblick über die unterschiedlichen Sensor technologien, wie Ultraschall, Radar, LiDAR und Kamera gegeben. Besonders wird auf die LiDAR-Technologie eingegangen, da diese in unserem Projekt eingesetzt wird. Des Weiteren wird auf die individuellen Einsatzgebiete, Vorteile sowie Nachteile eingegangen. Danach folgen unterschiedliche Ansätze der Sensordatenfusion und die Auswahl des Sensors für unser Projekt mit Hilfe der Scoring Methode.

Das zweite Kapitel behandelt das Problem der Berechnung des kürzesten Pfades. Es werden einige verschiedene Algorithmen beschrieben und von Grund auf neu implementiert, um ein möglichst intuitives Verständnis zu erwecken. Zum Schluss werden alle Algorithmen miteinander verglichen und eine Übersicht über deren Vor- und Nachteile gegeben.

Im dritten Kapitel wird ein Überblick über die verschiedenen Arten von Machine Learning Algorithmen und Reinforcement Learning gegeben. Dabei wird genauer auf Model Based Reinforcement Learning und dessen Details eingegangen. Es wird auch ein eigener MBRL Agent implementiert, um die Vorteile von Model-Based Algorithmen zu zeigen. Des Weiteren wird über die Optimierung und Weiterentwicklung von MBRL Algorithmen und Agents gesprochen.

Im letzten Teil der Diplomarbeit wird anfangs der Ansatz des Model Free Reinforcement Learnings erklärt. Zusätzlich werden das Explore-Exploit-Dilemma und die Monte Carlo Methoden beschrieben. Danach werden auch einige MFRL-Agenten vorgestellt und deren Funktionsweise genauer erläutert. Am Ende werden noch Anwendungsgebiete von MFRL vorgestellt.

Abstract

Utilization of LiDAR in Autonomous Driving:

The aim of this project is to achieve autonomous driving for a model car using a LiDAR sensor. For this, a model was trained in the simulation environment CARLA.

Firstly, an overview of various sensor technologies such as ultrasound, radar, LiDAR, and cameras is provided. Special attention is given to LiDAR technology as it is used in our project. Furthermore, the discussion delves into the specific applications, advantages, and disadvantages of each technology. Subsequently, various approaches to sensor data fusion are explored, followed by the selection of the sensor for our project using the scoring method.

The second chapter addresses the shortest path problem. Several different algorithms are described and re-implemented from scratch to evoke as intuitive an understanding as possible. Finally, all of them are juxtaposed, and their respective strengths and weaknesses are summarized.

In the third chapter, various types of machine learning and reinforcement learning algorithms are presented. This includes a detailed examination of Model-Based Reinforcement Learning and its specifics. Additionally, a custom MBRL agent is implemented to showcase the advantages of model-based algorithms. Furthermore, discussion revolves around the optimization and further development of MBRL algorithms and agents.

In the final part of the thesis, the approach of Model Free Reinforcement Learning is initially explained. Additionally, the Explore-Exploit dilemma and the Monte Carlo methods are described. Afterward, some MFRL agents are introduced, and their functioning is further highlighted. Finally, applications of MFRL are presented.

Inhaltsverzeichnis

| | |
|--|------------|
| Vorwort | i |
| Erklärung | i |
| Diplandenvorstellung | iii |
| Danksagungen | vii |
| Zusammenfassung | xv |
| Abstract | xvii |
| Inhaltsverzeichnis | xix |
| 1 Hinderniserkennung und -einplanung | 1 |
| 1.1 Einleitung | 1 |
| 1.2 LiDAR-Technologie | 1 |
| 1.2.1 Geschichte von LiDAR | 2 |
| Frühere Anwendungen und ihre Fortschritte | 2 |
| Aktuelle Anwendungen und Ausblick in die Zukunft | 3 |
| 1.2.2 Aufbau des Sensors | 5 |
| Nodding mirror type LiDAR-Sensor | 5 |
| Polygonal type LiDAR-Sensor | 6 |

| | | |
|--------|--|----|
| 1.2.3 | Funktionsweise | 6 |
| | Time-of-Flight | 6 |
| | Phasenverschiebung | 8 |
| 1.2.4 | Datenspeicherung | 9 |
| | LAS | 9 |
| 1.2.5 | Klassifizierung von LiDAR-Daten | 10 |
| 1.2.6 | Punktwolke | 12 |
| 1.2.7 | Datenauswertung Algorithmen | 13 |
| | PointNet | 13 |
| | PointNet++ | 17 |
| 1.2.8 | Einteilung von LiDAR-Sensoren | 20 |
| | Unterteilung nach Funktionsweise | 20 |
| | Unterteilung nach Dimensionalität der Datenerfassung | 22 |
| 1.2.9 | Vorteile von LiDAR | 24 |
| 1.2.10 | Nachteile von LiDAR | 25 |
| 1.3 | Vergleich mit anderen Sensor technologien | 26 |
| 1.3.1 | Überblick über andere Sensoren | 26 |
| 1.3.2 | TOF-Kamera | 27 |
| | Bestandteile | 27 |
| | Funktionsweise | 28 |
| | Anwendungsgebiete | 29 |
| | Vorteile | 30 |
| | Nachteile | 30 |

| | | |
|----------|--|-----------|
| 1.3.3 | Radar | 31 |
| | Funktionsweise | 31 |
| | Unterschied zu LiDAR | 32 |
| | Anwendungsgebiete | 33 |
| 1.3.4 | Ultraschall | 34 |
| | Funktionsweise | 34 |
| | Unterschied zu LiDAR | 35 |
| | Anwendungsgebiete | 36 |
| 1.3.5 | Sensordatenfusion | 37 |
| | Ebenen der Sensorfusion | 37 |
| | Sensorfusion im autonomen Fahren | 38 |
| 1.4 | Bezug auf das Projekt | 40 |
| 1.4.1 | Auswahl LiDAR-Sensor | 40 |
| | Kriterien | 40 |
| | Einordnung der Punkte | 41 |
| | Scoring Methode | 41 |
| | Erkenntnis und Entscheidung | 42 |
| 1.4.2 | Empfangen und Visualisierung der LiDAR-Daten | 43 |
| | Konstanten | 43 |
| | Funktionen | 44 |
| | Pygame-Programm | 44 |
| 2 | Pathfinding | 49 |
| 2.1 | Allgemeines | 49 |

| | | |
|--------|------------------------------------|----|
| 2.1.1 | Einleitung | 49 |
| 2.1.2 | Aufwand | 49 |
| 2.2 | Graphentheorie | 50 |
| 2.2.1 | Visualisierung | 50 |
| 2.2.2 | Kantengewichtete Graphen | 51 |
| 2.2.3 | Gerichtete Graphen | 51 |
| 2.2.4 | Multigraphen | 52 |
| 2.2.5 | Einfache Graphen | 53 |
| 2.2.6 | Netzwerke | 55 |
| 2.2.7 | Symbolik | 55 |
| 2.2.8 | Zyklen | 56 |
| 2.2.9 | Kreise | 56 |
| 2.2.10 | Kreisgraphen | 57 |
| 2.2.11 | Vollständige Graphen | 57 |
| 2.2.12 | Bipartite Graphen | 58 |
| 2.2.13 | Grids | 58 |
| 2.2.14 | Darstellung | 60 |
| | Adjazenzliste | 60 |
| | Adjazenzmatrix | 62 |
| 2.2.15 | Implementierung | 63 |
| 2.2.16 | Erzeugung | 71 |
| 2.3 | Algorithmen | 74 |
| 2.3.1 | Bewertung | 74 |

| | | |
|-------|--|-----|
| 2.3.2 | Hilfsklassen | 75 |
| | EndCondition | 75 |
| | SearchAlgorithm | 77 |
| | PathfindingAlgorithm | 78 |
| | PathTracer | 79 |
| | CycleException | 81 |
| 2.3.3 | Breitensuche | 82 |
| 2.3.4 | Tiefensuche | 84 |
| | Iterativ | 84 |
| | Rekursiv | 88 |
| 2.3.5 | Bestensuche | 91 |
| | Fibonacci-Heap | 96 |
| | Dijkstra-Algorithmus | 99 |
| | A*-Algorithmus | 101 |
| 2.3.6 | Bidirektionale Bestensuche | 103 |
| 2.4 | Vergleich und Evaluation der Algorithmen | 107 |
| 2.4.1 | Hilfsklassen | 107 |
| | Pathfinder | 107 |
| 2.4.2 | Benchmarking | 109 |
| | ModifiableGraph | 109 |
| | 15-Puzzle | 115 |
| | Rubik's Cube | 121 |
| 2.4.3 | Zusammenfassung der Ergebnisse | 122 |

| | | |
|----------|---|------------|
| 2.5 | Projektbezug | 125 |
| 3 | Model Based Reinforcement Learning | 127 |
| 3.1 | Artificial Intelligence, AI | 127 |
| 3.1.1 | Einführung in AI | 127 |
| | Grundlagen der Künstlichen Intelligenz | 128 |
| | Menschliches Handeln | 128 |
| | Menschliches Denkvermögen | 128 |
| | Rationales Denken | 129 |
| | Rationales handeln | 129 |
| | Technische Grundlagen der Künstlichen Intelligenz | 130 |
| | Wichtige AI-Konzepte und Terminologie | 131 |
| 3.1.2 | Einführung in Model Based Reinforcement Learning | 149 |
| | Planning vs. Learning | 149 |
| | Monte Carlo Tree Search (MCTS) | 151 |
| 3.1.3 | Model Predictive Control (MPC) | 154 |
| | Control und Optimization | 154 |
| 3.1.4 | Probabilistic Models im Model Based RL | 158 |
| | Uncertainty Propagation und Risk-Reduction | 158 |
| 3.1.5 | Implementierung | 159 |
| | Flappy Bird Clone | 159 |
| | Environment | 163 |
| | Agent | 167 |
| 3.2 | Projektbezug | 184 |

| | |
|--|------------|
| 4 Model Free Reinforcement Learning | 185 |
| 4.1 Einleitung | 185 |
| 4.1.1 Hintergrund und Kontext | 185 |
| 4.1.2 Bedeutung des Model Free Reinforcement Learning | 186 |
| 4.2 Konzepte der Model Free Reinforcement Learning | 186 |
| 4.2.1 Einleitung | 186 |
| 4.2.2 Erklärung | 187 |
| 4.2.3 Konzepte | 187 |
| Policy | 187 |
| Value-Based vs Policy-Based Algorithmen | 188 |
| Optimal Policy | 188 |
| Angewandter Algorithmus | 189 |
| Vergleich der Model Free Algorithmen | 191 |
| 4.3 Grundlagen des Reinforcement Learnings | 192 |
| 4.3.1 Markov-Decision-Process | 192 |
| Ziel von MDP | 192 |
| Formel für Optimal Policy | 193 |
| 4.3.2 Reward-Functions | 193 |
| 4.3.3 Value-Functions | 193 |
| 4.4 Exploration-Exploitation dilemma | 194 |
| 4.4.1 Balancieren von Exploitation und Exploration | 194 |
| 4.4.2 Exploration Strategies | 196 |
| Greedy Strategy | 196 |

| | |
|--|-----|
| Epsilon-Greedy | 197 |
| Decaying Epsilon-Greedy | 197 |
| Upper Confidence Bound | 198 |
| 4.5 Methoden des Machine Learnings | 199 |
| 4.5.1 Monte-Carlo-Methoden | 199 |
| Monte Carlo Tree Search | 199 |
| 4.5.2 Episodenbasiertes Lernen | 201 |
| 4.5.3 Bewertung von Policies | 201 |
| Vorteile und Einschränkungen | 202 |
| 4.6 Methoden des Policygradients | 203 |
| 4.6.1 Parametrisierung von Policy | 203 |
| 4.6.2 REINFORCE | 203 |
| 4.6.3 Advantage Actor-Critic (A2C) | 203 |
| 4.7 Deep Q-Netzwerke (DQNs) | 204 |
| 4.7.1 Approximation von Q-Functions | 205 |
| 4.7.2 Experience Replay | 205 |
| 4.7.3 Target-Networks | 205 |
| 4.7.4 Umsetzung einer DQN | 206 |
| Aufbau des Agenten | 206 |
| Aufbau der Umgebung | 210 |
| 4.8 Proximal Policy Optimization (PPO) | 212 |
| 4.8.1 Policyoptimierung | 212 |
| Clipping Mechanism | 213 |

| | | |
|------------------------------------|--|------------|
| 4.8.2 | Surrogate Objectives | 213 |
| 4.8.3 | Vergleich DQN Agent vs PPO Agent | 214 |
| 4.9 | Soft Actor Critic (SAC) | 215 |
| 4.9.1 | SAC-Einführung | 215 |
| 4.9.2 | Implementierung der Netzwerke | 216 |
| 4.9.3 | Entropy-Regulation in SAC | 217 |
| 4.9.4 | Q-Value-Optimierung in SAC | 218 |
| 4.9.5 | Zielfunktion von SAC | 218 |
| 4.10 | Anwendung des Model Free Reinforcement Learnings | 219 |
| 4.10.1 | Spielstrategie und Spielen | 219 |
| 4.10.2 | Natürliche Sprachverarbeitung | 219 |
| 4.11 | Projektbezug | 220 |
| Anhang | | 223 |
| Abbildungsverzeichnis | | 223 |
| Tabellenverzeichnis | | 229 |
| Verzeichnis der Listings | | 231 |
| Literaturverzeichnis | | 235 |
| Kapitelzuordnung | | 255 |
| Arbeitsprotokolle | | 257 |

Kapitel 1

Hinderniserkennung und -einplanung

1.1 Einleitung

Dieses Kapitel beschäftigt sich mit unterschiedlichen Sensor technologien, wie LiDAR, Kamera, Radar und Ultraschall und deren Funktionsweise beziehungsweise Einsatzgebieten. Der Hauptfokus wird aber auf die LiDAR-Technologie gelegt, da diese im Projekt verwendet wird.

Mit Hilfe von Sensoren ist unter anderem die Erkennung von Objekten möglich, wodurch der Abstand zu diesen gemessen und auf Basis dieser Messungen visuelle Veranschaulichungen erstellt werden können. [PF:Web04]

1.2 LiDAR-Technologie

LiDAR ist die Abkürzung für *Light detection and ranging* oder *Laser Imaging, Detection and Ranging*, was auf Deutsch übersetzt so viel wie *Lichterkennung und Reichweitenmessung* bedeutet. Es handelt sich dabei um eine Vermessungstechnologie, welche Laserlicht benutzt, um die Entfernung von Objekten zu messen. Anhand dieser Messungen kann eine 3D Karte mit enormer Präzision erzeugt werden. [PF:Web04, PF:Web05]

1.2.1 Geschichte von LiDAR

Das Grundkonzept des LiDAR-Sensors begleitet die Menschheit schon seit etlichen Jahrzehnten, dennoch verlief die Weiterentwicklung und Optimierung nur schleppend. [PF:Web01]

Der Vorgänger von LiDAR ist *Sonar*, welches die Abkürzung für *Sound Navigation and Ranging* ist und früher primär für die Erkundung von Gewässern eingesetzt wurde. Die Abbildung 1.1 zeigt hierbei die Funktionsweise von Sonar. Diese Technologie verwendet für die Objekterkennung Schallwellen, welche von einem Sender, der auch als Empfänger fungiert, ausgesendet werden. Wenn die Welle auf ein Objekt stößt, wird diese reflektiert. Der Abstand wird anhand der Zeit, welche die Welle bis zum Zurückkommen zu dem Empfänger braucht, berechnet. [PF:Web01]

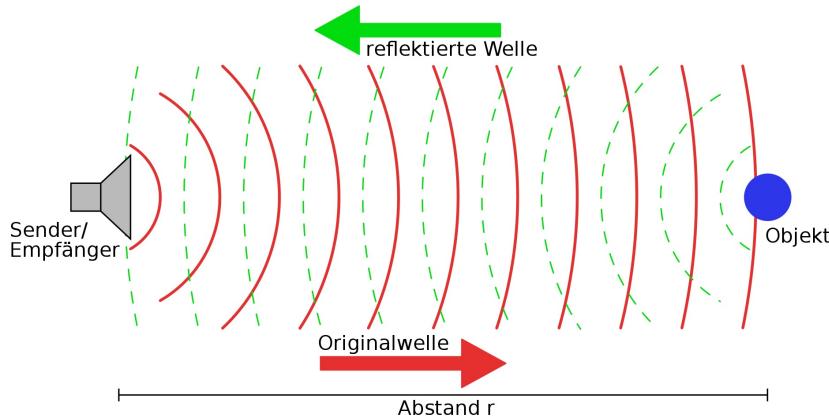


Abbildung 1.1: Funktionsweise von Sonar
[PF:Web13]

Frühere Anwendungen und ihre Fortschritte

Edward Hutchinson Synge legte 1930 mit der Idee, zu messen, wie lange die Zeitspanne ist, bis Licht zu einem Ermittler zurückkehrt, um somit die Entfernung von Objekten zu bestimmen, die Grundsteine von LiDAR. Bei seinen Versuchen nutzte er Suchscheinwerfer, welche auch von etlichen Wissenschaftlern in den folgenden Jahrzehnten für unter anderem die Erkundung der Atmosphäre verwendet wurden. [PF:Web01]

1960 wurde der erste Laser, der *Ruby Laser*, von Theodore Maiman und seinem Team am *Hughes Research Laboratory* vorgestellt. Im folgenden Jahr wurde der erste LiDAR-Prototyp, ebenfalls von demselben Team, erstellt und 1962 auf den Markt gebracht. [PF:Web01]

Die NASA¹ verwendete in den 1970er Jahren LiDAR-Sensoren bei Erkundungen des Mondes oder von Planeten wie Mars und Merkur. Zu dieser Zeit wurde diese Art von Sensor hauptsächlich für Kartierungen eingesetzt. Dennoch verlief die Weiterentwicklung nur schleppend, denn es mangelte an wirtschaftlich nutzbaren und effizienten GPS²-Systemen. [PF:Web01]

Doch im folgenden Jahrzehnt wurde GPS immer innovativer und in Kombination mit Satelliten konnte LiDAR nun auch in der Luft eingesetzt werden. [PF:Web01]

In den mittleren 1990er Jahren wurde das Einsatzgebiet von LiDAR durch maßgebende technische Verbesserungen, wie die Erzeugung von 2000 bis 25000 Pulse pro Sekunde durch einen LiDAR-Sensor, erweitert. Das Einsatzgebiet wurde unter anderem durch die Planung von Stadtentwicklungen und der Kartierung von unentdeckten Gebieten erweitert. *Cyra Technologies* stellte 1998 den ersten 3D-Sensor, welcher auf einem Stativ stand, und mit einer Punktwolkensoftware integriert war, vor. [PF:Web01, PF:Web14]

Durch die ständige Weiterentwicklung gab es nun sämtliche diverse Anwendungsbeziehe, wie beispielsweise auch Hochwasserschutz. Zudem wurden auch die Türen der LiDAR-Archäologie geöffnet. Ein Beispiel hierfür ist das erste LiDAR-basierte Höhenmodell eines archäologischen Gebietes in Copan, Honduras. Dieses Modell wurde von einem Team der *University of Texas*, geleitet von James Gibeaut, im Jahre 2000 erstellt. [PF:Web01]

Aktuelle Anwendungen und Ausblick in die Zukunft

Aufgrund der hohen Effizienz und der Einsparungen von Personalkosten, welche durch LiDAR ermöglicht werden, wird diese Technologie oft gegenüber herkömmlichen Messtechniken bevorzugt. [PF:Web01]

Durch die vielen Einsatzmöglichkeiten kann der Laser-Radar³ in vielen verschiedenen Bereichen eingesetzt werden. Einige Beispiele hierfür wären:

- **Kontrolle und Sicherung des Verkehrs**

Hierbei kann die Technologie vielseitig verwendet werden. Unter anderem zur Mithilfe bei der automatischen Mauterhebung, denn ihr ist es möglich unterschiedliche Fahrzeuge zu klassifizieren und den entsprechenden Tarif zu berechnen. Außerdem wäre auch der Einsatz an Kreuzungen für die Berechnung des

¹National Aeronautics and Space Administration

²Global Positioning System

³LiDAR

idealnen Verkehrsflusses, welcher zusätzlich auch mehr Sicherheit gewährt, vorstellbar. Aber auch die Anwendung im Bahnbereich ist möglich. Hierbei könnte der Sensor Objekte, welche sich ungewollte im Schienenbereich befinden, wie Personen, welche auf die Gleise gestürzt sind, erkennen, identifizieren und schlussendlich einen Alarm auslösen, welcher den Zugführer informiert. [PF:Web07]

- **Präzision von Sicherheitssystemen**

Durch die hohe Resistenz gegen elektromagnetische Störungen, eine präzise Auflösung, die Fähigkeit, Objekte in 3D anzuzeigen und die Unabhängigkeit von Lichtverhältnissen, bietet LiDAR unzählige neue Lösungen im Bereich der Sicherheitsbranche. Da die Technologie zudem sehr genau ist, werden Fehlalarme immer seltener, was Kosten für Sicherheitspersonal enorm reduzieren kann. Durch die Anonymisierung von Daten, wird LiDAR den steigenden Datenschutzbedenken weitgehend gerecht. Denn LiDAR kann zwar Objekte, beziehungsweise Personen, erkennen, verfolgen und klassifizieren, aber währt dennoch die Anonymität von erkannten Menschen. [PF:Web07]

- **Erkennung von Verborgenen in der Kartografie**

In Kombination mit Drohnen können Informationen über Gebiete, die sonst nur schwer für Personen erreichbar sind, gesammelt werden. Es werden zum Beispiel Wälder oder Küsten in 3D-Karten dargestellt, um Regierungen und Institutionen zu helfen, Probleme wie Pflanzenrückgang oder Bodenabtrag zu erkennen. Zudem kann LiDAR nach Naturkatastrophen helfen weitere Informationen über die betroffenen Gebiete zu bekommen, was eine bessere Einschätzung der aktuellen Situation des Gebietes möglich macht. [PF:Web07]

- **Optimierung des Ertrages in der Agrarwirtschaft**

Mit dem Einsatz von LiDAR kann die Bodenbeschaffenheit in einem bestimmten Areal bestimmt werden. Somit wissen Landwirte, welche Art von Dünger in Betracht gezogen werden sollte, beziehungsweise in welcher Form generell angebaut werden kann. Zudem werden Informationen über die Qualität der Pflanzen und in welchem Tempo sie wachsen mitgeteilt. Durch LiDAR wird es möglich Maschinen präzise zu steuern, um die Effizienz zu erhöhen. Da diese Technologie kein Licht benötigt, können Arbeiten bis in die Nacht verlängert werden. [PF:Web07]

1.2.2 Aufbau des Sensors

Derzeit gibt es 2 weitverbreitende Arten von LiDAR- Sensoren. Nämlich *nodding mirror type* und *polygonal type LiDAR-Sensoren*. [PF:Web33]

Nodding mirror type LiDAR-Sensor

In der Regel besteht ein *nodding mirror type LiDAR-Sensor* aus 4 Hauptkomponenten. Die Abbildung 1.2 zeigt den Aufbau und kennzeichnet die Bestandteile. Die erste wesentliche Komponente ist die Lichtquelle⁴. Die Zweite sind die optischen Komponenten⁵, es wird hierbei in der Mehrzahl gesprochen, da mehrere davon in einem Sensor eingebaut werden können. Zum Beispiel wird ein rotierender beziehungsweise schwingender Spiegel⁶ für die Änderung des Lichts und um eine 360-Grad-Ansicht abzudecken verwendet. Für die Fokussierung auf den Fotodetektor wird eine optische Linse eingesetzt. Bei den Fotodetektoren handelt es sich auch um die dritte Hauptkomponente des Sensors, denn dieser dient dazu Licht zu empfangen und das Signal elektronisch zu verarbeiten. Der letzte Bestandteil ist GPS, welches verwendet wird um die exakte Position sowie Ausrichtung im 3D-Raum zu bestimmen. [PF:Web23]

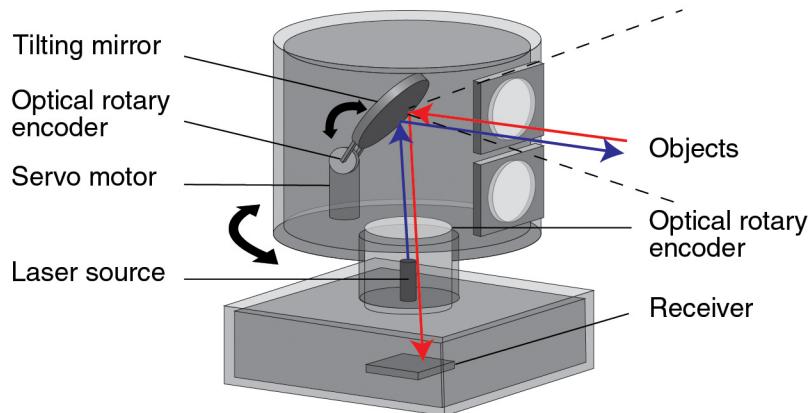


Abbildung 1.2: Aufbau eines nodding mirror type LiDAR-Sensors
[PF:Web23]

⁴Laser source

⁵optical rotary encoder

⁶tilting mirror

Polygonal type LiDAR-Sensor

In der Abbildung 1.3 werden die Bestandteile eines *polygonal type LiDAR-Sensors* grafisch veranschaulicht. Dabei kommt ein *Polygonspiegel*⁷ zum Einsatz, welcher namensgebend ist. Weiters wird eine Laserquelle für die Erzeugung eines horizontalen Laserstrahles⁸ verwendet. Der Laserstrahl wird daraufhin auf den ersten Spiegel ausgerichtet, dieser reflektiert ihn und der reflektierte Strahl wird auf den nächsten Spiegel gerichtet, welcher einen anderen vertikalen Winkel besitzt. Durch diese Prozedur können mehrere vertikale Laserstrahlen erzeugt werden. [PF:Web34]

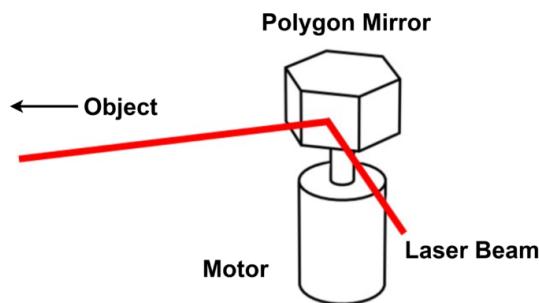


Abbildung 1.3: Aufbau eines polygonal type LiDAR-Sensors
[PF:Web34]

1.2.3 Funktionsweise

Es gibt verschiedene Arten, wie ein LiDAR-Sensor die Entfernung messen kann. Die erste Art misst die Zeit, die der Laserstrahl für die Rückkehr braucht. Hierbei spricht man von *Time-of-Flight*⁹. Bei der zweiten Variante wird die Phasenverschiebung berechnet. [PF:Web05]

Time-of-Flight

Bei diesem Messungsverfahren, benutzt LiDAR die Lichtgeschwindigkeit, um zu messen, wie weit ein Objekt entfernt ist. Die Funktionsweise ist in der Abbildung 1.4 ersichtlich. Das Licht wird üblicherweise in Form einer gepulsten Laserdiode ausgesendet, wobei bei der Aussendung die Zeit zu messen beginnt. Der Lichtimpuls erreicht

⁷Polygon Mirror

⁸Laser Beam

⁹Lichtlaufzeit

nahezu Lichtgeschwindigkeit und wird bei einem Aufprall auf einem Objekt zum LiDAR-Sensor zurückreflektiert. Innerhalb des Sensors trifft der Strahl auf einen Photodetektor, welcher die zuvor gestartete Zeit stoppt. Durch die Kenntnis wie schnell sich das Licht bewegt, können Rückschlüsse auf die Entfernung zu dem Objekt gemacht werden. [PF:Web15]

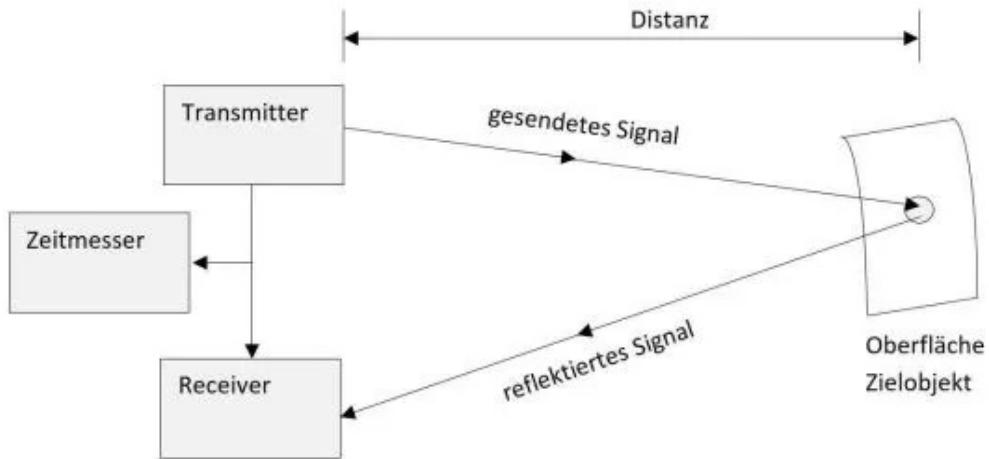


Abbildung 1.4: Funktionsweise Time-of-Flight
[PF:Web16]

Hierbei wird folgende Formel angewandt:

$$d = \frac{c \cdot t}{2}$$

- d ... Entfernung zum Objekt
- c ... Lichtgeschwindigkeit
- t ... Zeit zwischen Lichtaussendung und Erkennung [PF:Web15]

Phasenverschiebung

Für die Bestimmung der Entfernung kann auch die Methode der Messung der Phasenverschiebung verwendet werden. Hierbei kommt eine kontinuierliche Quelle, welche mit konstanter Frequenz moduliert wird, statt einer gepulsten Laserquelle zum Einsatz. [PF:Web31]

Daraus resultiert, dass die Eingabe als Sinuskurve betrachtet werden kann. Lichtsensoren erkennen die Lichtstärke und ob Licht vorhanden ist. Mithilfe des Rücksignals wird ebenfalls eine Sinuskurve gebildet. Die Entfernung kann nun durch den Vergleich der Phasendifferenz mit folgender Formel berechnet werden: [PF:Web31]

$$d = \frac{c \cdot \Delta\phi}{2\pi \cdot f}$$

- d ... Entfernung zum Objekt
- c ... Lichtgeschwindigkeit
- $\Delta\phi$... Phasendifferenz
- f ... Frequenz, mit der die Leistung moduliert wurde [PF:Web31]

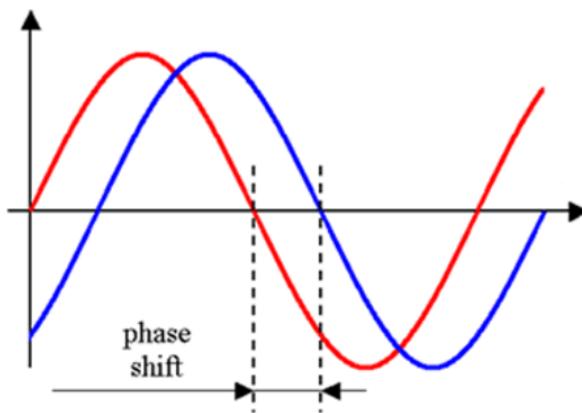


Abbildung 1.5: Erkennung der Phasenverschiebung
[PF:Web32]

Die Abbildung 1.5 veranschaulicht grafisch, wie die Phasenverschiebung berechnet wird. Die rote Sinuskurve repräsentiert hierbei das Licht, welches vom Sensor ausgesendet wird und die blaue Kurve das Licht, welches zurückkommt. [PF:Web32]

1.2.4 Datenspeicherung

Anfangs waren LiDAR-Daten ausschließlich im ASCII¹⁰-Format verfügbar. Doch im Laufe der Zeit stieg die Größe der Sammlungen an Daten von LiDAR und es wurde LAS¹¹, ein Binärformat, für die Aufteilung und Strukturierung verwendet. Heutzutage findet man fast nur noch LAS vor, da es über Vorteile wie die Möglichkeit, zusätzliche Informationen zu speichern oder den Fakt, dass es sich um ein binäres Format handelt, welches die Effizienz von Importprogrammen steigert, verfügt. [PF:Web18]

LAS

LAS wurde von ASPRS¹² für die Speicherung von LiDAR-Punktwolkendaten entwickelt. Es werden detaillierte Informationen zu LiDAR-Punkten gespeichert. Diese Informationen sind: [PF:Web21]

- 3D-Koordinaten (x, y und z)
- Intensitätswerte
- Klassifizierungscodes
- zusätzliche Attribute [PF:Web21]

Die Abbildung 1.6 zeigt hierbei den Aufbau der Datei, welcher in 4 Abschnitte unterteilt ist:

- **Public header block**

Beschreibt die Anzahl der Punkte, das Format, den Umfang der Punktwolke und weitere allgemeine Daten. [PF:Web22]

- **Variable Length Records (VLR)**

Flexible Anzahl an optionaler Datensätze, um diverse Informationen zur Verfügung zu stellen. Das können zum Beispiel Metadaten, das verwendete räumliche Bezugssystem oder Benutzeranwendungsdaten sein. Die maximale Länge der Nutzlast beträgt pro VLR 65635 Bytes. [PF:Web22]

¹⁰American Standard Code for Information Interchange

¹¹LASer

¹²American Society for Photogrammetry and Remote Sensing

- **Point data records**

Daten für jeden individuellen Punkt in der Punktwolke, einschließlich Klassifizierung, Flug- und Scan-Daten und vielen anderen Informationen. [PF:Web22]

- **Extended Variable Length Records**

Wurden mit LAS 1.3 eingeführt und sind ähnlich wie VLRs. Der Unterschied ist jedoch, dass 8-Byte-Größenbeschreibungen verwendet werden, wodurch eine essenziell höhere Datenlast pro Datensatz ermöglicht wird. [PF:Web22]

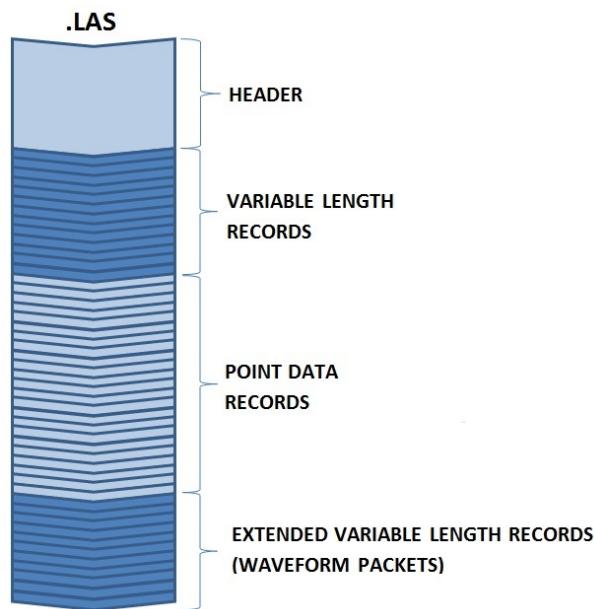


Abbildung 1.6: Aufbau LAS-Datei
[PF:Web20]

1.2.5 Klassifizierung von LiDAR-Daten

Jeder Punkt, welcher von LiDAR erkannt wird, kann klassifiziert¹³ werden. LAS-Dateien definieren mithilfe von Ganzzahlcodes die unterschiedlichen Klassen. [PF:Web03, PF:Web12]

Die Codes werden von der ASPRS für die LAS-Formate 1.1, 1.2, 1.3 und 1.4 festgelegt. Wobei Letztere die aktuellste Version ist. Seit der LAS-Version 1.1 ist es möglich eine sekundäre Klassifizierung, welche durch das hinzufügen einer Class-Flag möglich wurde, von LiDAR-Punkten zu erfassen. [PF:Web12]

¹³Definition des Objekttyps

Man unterscheidet zwischen 4 verschiedenen einstellbaren Klassifizierungs-Flags:

| Feldname | Beschreibung |
|----------------|--|
| Synthetisch | Punkt, welcher nicht durch LiDAR erfasst wurde |
| Schlüsselpunkt | besondere Relevanz im Modellierungsprozess |
| Ausgeschlossen | soll nicht bei der Verarbeitung verwendet werden |
| Überlappend | befindet sich im Bereich von mindestens 2 Flugbahnen |

Tabelle 1.1: Klassifizierungsflags
[PF:Web12]

Wenn mit einer LAS-Version zwischen 1.1 und 1.4 gearbeitet wird, kann die Kategorie dem Schema der ASPRS entnommen werden. In der Tabelle 1.2 werden die Klassifizierungswerte angegeben. LAS 1.1- 1.3 unterstützt nur die Codes 0-31. 1.4 hingegen alle, welche in der Tabelle genannt werden. [PF:Web12]

| Klassifizierungscode | Bedeutung |
|----------------------|------------------------------|
| 0 | nie klassifiziert |
| 1 | nicht zugewiesen |
| 2 | Oberfläche |
| 3 | niedrige Vegetation |
| 4 | mittelhohe Vegetation |
| 5 | hohe Vegetation |
| 6 | Gebäude |
| 7 | Tiefpunkt |
| 8 | Reserviert |
| 9 | Wasser |
| 10 | Schienen |
| 11 | Straßenbelag |
| 12 | Reserviert |
| 13 | Schutzdraht (Schild) |
| 14 | Drahtleiter (Phase) |
| 15 | Strommast |
| 16 | Leitungsverbinder (Isolator) |
| 17 | Brückenfahrbahn |
| 18 | hohes Rauschen |
| 19-63 | Reserviert |
| 64-255 | benutzerdefiniert |

Tabelle 1.2: Klassifizierungswerte von LAS
[PF:Web12]

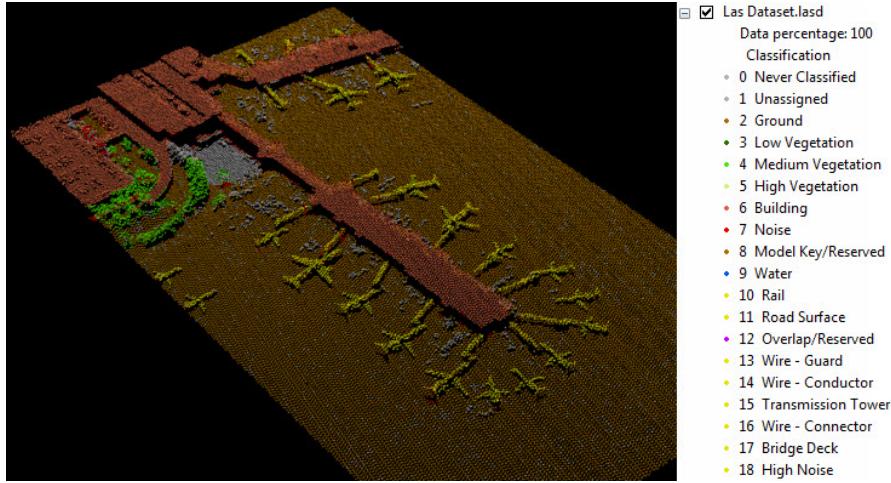


Abbildung 1.7: Klassifizierung der erkannten Punkte
[PF:Web12]

In der Abbildung 1.7 werden die erkannten Punkte nach der Tabelle 1.2 eingeordnet.

1.2.6 Punktwolke

Bei einer Punktwolke¹⁴ handelt es sich um eine Sammlung von Punkten, welche sich im 3 dimensionalen Raum befinden. Dabei wird jedem Punkt eine Koordinate nach kartesischer Konvention¹⁵ sowie mögliche Zusatzinformation zugeordnet. Diese Information kann von dem Zeitpunkt der Erfassung bis hin zur Intensität reichen. [PF:Web37]

Dabei setzen die Punktwolkendaten die Grundsteine für die Visualisierung von Objekten, Umgebungen und Bereichen. Für eine exakte Darstellung sind im Normalfall Millionen an Punkten erforderlich. [PF:Web39]

Eine Punktwolke kann folgendermaßen definiert werden:

$$\{P_i \mid i = 1, \dots, n\}$$

[PF:Web44]

Hierbei ist jeder Punkt (P_i) selbst ein Vektor. [PF:Web44]

¹⁴point cloud

¹⁵(x, y, z)

Die Abbildung 1.8 zeigt eine beispielhafte Punktwolke, in welcher eine Person in der oberen Mitte in türkis zu sehen ist. Diesen Rückschluss kann man aufgrund der Anhäufung an Punkten, welche die Silhouette eines Menschen zeigen, schließen.

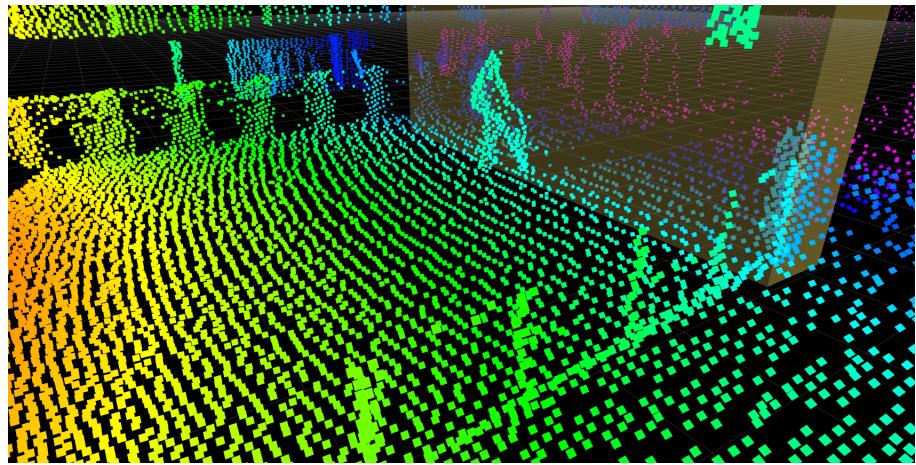


Abbildung 1.8: Beispiel für eine Punktwolke
[PF:Web38]

1.2.7 Datenauswertung Algorithmen

Für die Auswertung von Punktwolken gibt es verschiedene Algorithmen wie PointNet oder PointNet++.

PointNet

Bei PointNet handelt es sich um ein neuronales Netzwerk¹⁶, welches speziell für das Deep Learning mit Punktwolken entwickelt wurde, um 3D-Klassifikation und Segmentation durchzuführen. Es wurde 2017 von Charles R.Qi, Hao Su, Kaichun Mo und Leonidas J. Guibas an der Standford Universität erfunden. [PF:Web44, PF:Web52]

Im Gegensatz zu vielen anderen Datenauswertungs-Algorithmen bietet PointNet die Möglichkeit direkt ungeordnete Punktwolken zu verarbeiten. In der Erklärung über die Architektur und Funktionsweise werden zur Vereinfachung nur die x-, y- und z-Achse und keine Zusatzattribute verwendet. [PF:Web44]

¹⁶Modell, welches Daten verarbeitet und Muster lernt

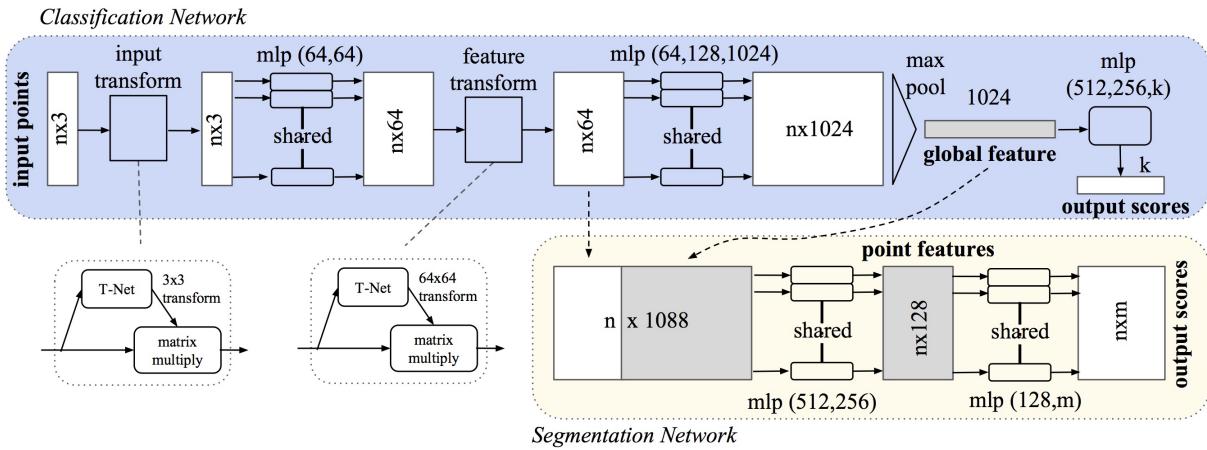


Abbildung 1.9: Architektur von PointNet
[PF:Web44]

Wie in Abbildung 1.9 grafisch veranschaulicht ist, ist die Architektur von PointNet folgendermaßen unterteilt. Das Klassifizierungsnetzwerk¹⁷ erhält als Eingabe n Punkte und wendet auf diese Eingabe- und Merkmalstransformationen an. Daraufhin werden Merkmale der Punkte, mit Hilfe von Max-Pooling, welches den maximal Wert findet, zusammengefasst. Hinaus kommen die Klassifikationswerte k . [PF:Web44]

Das Segmentierungsnetzwerk¹⁸ ist eine Erweiterung des Klassifizierungsnetzwerkes und kombiniert lokale und globale Merkmale und gibt zudem auch individuelle Punkt-werte für jeden Punkt aus. Die Zahlen in den Klammern neben mlp , was für *Multi-Layer Perceptron* steht, repräsentieren die Größe der Schichten. Zudem repräsentiert n die Neuronen Anzahl, mit der Anzahl, welche sich nach nx befindet, pro Schicht. [PF:Web44]

Die Architektur von PointNet wurde von den Eigenschaften einer Punktmenge in \mathbb{R}^n ¹⁹ inspiriert. [PF:Web44]

Bei der Eingabe im \mathbb{R}^n handelt es sich um eine Teilmenge von Punkten aus einem euklidischen Raum²⁰ mit 3 Haupteigenschaften: [PF:Web44]

- **Unsortiert**

Im Gegensatz zu Bildern, welche aus einem Pixel-Array mit einer festen Reihenfolge bestehen, sind Punktwolken nicht nach einer speziellen Reihenfolge angeordnet. Wenn die Eingabe mehrere Punktwolken umfasst, muss unabhängig von der Abfolge, ein konsistentes Ergebnis hinauskommen. [PF:Web44]

¹⁷Classification Network

¹⁸Segmentation Network

¹⁹Darstellung des n-dimensionalen euklidischen Raums

²⁰abstrakter Raum, der grundlegende Konzepte der Geometrie berücksichtigt

- **Interaktion zwischen Punkten**

Punkte können nicht unabhängig voneinander betrachtet werden, da benachbarnte Punkte sinnvolle Teilmengen bilden. Deshalb muss das Modell sowohl lokale Strukturen von näher gelegenen Punkten erkennen als auch die kombinatorische Interaktion zwischen den Strukturen berücksichtigen. [PF:Web44]

- **Unveränderlichkeit bei Transformationen**

Die Repräsentation der Punktfolge soll nicht von unterschiedlichen Veränderungen, wie Drehen oder Verschieben der Punkte, beeinflusst werden. Sodass nach jeder Punkttransformation das Modell immer noch dazu im Stande ist, dieselbe Kategorie beziehungsweise Segmentierung wie zuvor zu erkennen. [PF:Web44]

Das Netzwerk von PointNet hat drei Schlüsselmodule. Nämlich die Max-Pooling-Schicht, als symmetrische Funktion zur Zusammenfassung von Informationen aus allen Punkten, eine Struktur zur Kombination von lokalen und globalen Informationen und zwei gemeinsamen Ausrichtungsnetzwerken, welche für die Ausrichtung der Eingabepunkte und Punktmerkmale zuständig sind. [PF:Web44]

Für die Sicherstellung, dass ein Modell nicht von der Eingaben Daten-Reihenfolge beeinflusst wird, wird eine symmetrische Funktion verwendet. Diese kombiniert die Information der Daten. Weiters nimmt sie n Vektoren als Eingabe und gibt einen Vektor, der Unabhängig von der Abfolge der Eingabe ist, aus. Beispielsweise sind Multiplikation und Addition solche Funktionen. Die Idee besteht also darin, auf eine Gruppe von Punkten eine allgemeine Funktion zu schätzen, indem eine symmetrische Funktion auf die transformierten Elemente in der Menge angewendet werden: [PF:Web44]

$$f(\{x_1, \dots, x_n\}) \approx g(h(x_1), \dots, h(x_n))$$

wobei,

$$f : \mathbb{R}^{R^N} \rightarrow \mathbb{R}, \quad h : \mathbb{R}^N \rightarrow \mathbb{R}^K, \quad g : \underbrace{\mathbb{R}^K \times \cdots \times \mathbb{R}^K}_n$$

und es sich bei g um eine symmetrische Funktion handelt.

Es wird ein Multi-Layer Perceptron²¹ Netzwerk verwendet um h anzunähern und eine Kombination aus einer Max-Pooling-Funktion und einer einfachen Funktion um g anzunähern. Durch eine Sammlung von h -Funktionen kann eine Reihe von f -Funktionen erlernt werden, um unterschiedliche Eigenschaften des Sets zu ermitteln. [PF:Web44, PF:Web53]

²¹Einheit eines neuronalen Netzes

Das Ergebnis der lokalen und globalen Informationsaggregation ist ein Vektor $[f_1, \dots, f_K]$, welcher als globale Signatur des Eingabesatzes fungiert und die wichtigsten Merkmale des gesamten Datensatzes erfasst. Auf diesen Merkmalen kann ein Multi-Layer-Perception-Klassifikator trainiert werden, um Klassifizierungen durchzuführen. Für die Segmentierung der Punkte benötigt man eine Kombination aus lokalem und globalem Wissen. Damit man beides bekommt, wird, nachdem der globale Merkmalsvektor der Punktwolke berechnet wurde, dieser zurückgegeben, um Merkmale für jeden Punkt zu bekommen. Daraufhin werden neue Merkmale, basierend auf den kombinierten Punktmerkmalen, pro Punkt extrahiert. Die Merkmale des Punktes besitzen jetzt Informationen über die unmittelbare Umgebung des Punktes²² und den kompletten Datensatz²³. Nun ist es dem Netzwerk möglich pro Punkt Größen vorherzusagen, die sowohl lokal als global abhängig sind. [PF:Web44]

Das gemeinsame Ausrichtungsnetzwerk wird für die Sicherstellung, dass eine Punktwolke nach Transformationen noch immer über die unveränderte semantische Beschriftung der Punktwolke verfügt, eingesetzt. Das Netzwerk sorgt also dafür, dass die bereits gelernte Darstellung von der Punktmenge den Transformationen invariant ist. Die naheliegendste Lösung ist, dass alle Eingabesätze in einem standardisierten Raum ausgerichtet werden, bevor die Merkmale extrahiert werden. Dazu wird ein kleines Netzwerk namens T-Netzwerk, wie in Abbildung 1.9 abgebildet ist, für die Vorhersage von einer speziellen Art von Matrix²⁴ verwendet. Dann wird diese Matrix unmittelbar auf die Koordinaten der Eingabepunkte angewendet. Bei dem T-Netzwerk handelt es sich um eine simplere Version des Hauptnetzwerks mit elementaren Bausteinen für die Extrahierung der Punktmerkmale, Schichten für die Zusammenführung der Merkmale²⁵ und Schichten, die dafür zuständig sind, Daten zwischen den Netzwerkelementen zu übertragen. [PF:Web44]

Zudem kann diese Idee für die Merkmalsausrichtung verwendet werden. Dazu wird ein weiteres Netzwerk, um die Transformationsmatrix vorherzusagen, eingefügt. Dadurch kommt es aber zu Erschwerungen bei der Optimierung, denn die Transformationsmatrix im Merkmalsraum verfügt über eine enorm höhere Dimension im Vergleich zu der Räumlichen Matrix. Um dem entgegenzuwirken wird eine Regulierung zu dem Softmax-Trainingsverlust²⁶ eingefügt. Die Merkmalstransformationsmatrix soll hierbei nahe an der orthogonalen Matrix liegen. Was bedeutet, dass Zeilen- und Spaltenvektoren normal aufeinander stehen und eine Länge von eins besitzen:

$$L_{\text{reg}} = \|kI - AA^T\|_F^2$$

²²globale Information

²³lokale Information

²⁴Transformationsmatrix

²⁵Max-Pooling

²⁶Kostenfunktion zur Bewertung der Leistung während des Trainings

[PF:Web44, PF:Web54]

Hierbei steht A für die vorhergesagte Merkmalsausrichtungsmatrix des kleineren Netzwerkes. Da eine orthogonale Transformation keine Eingabeinformation verliert ist diese vorteilhaft. Durch die Einführung dieses Terms wurde zudem die Optimierung stabilisiert und die Leistung des Modells verbessert. [PF:Web44]

PointNet++

PointNet++ ist die Weiterentwicklung von PointNet und dient dazu, hierarchisches Merkmalslernen auf Punktsets in einem metrischen Raum zu ermöglichen. Es wurde 2017 von Charles R. Qi, Li Yi, Hao Su und Leonidas J. Guibas an der Stanford Universität erfunden. [PF:Web45, PF:Web46]

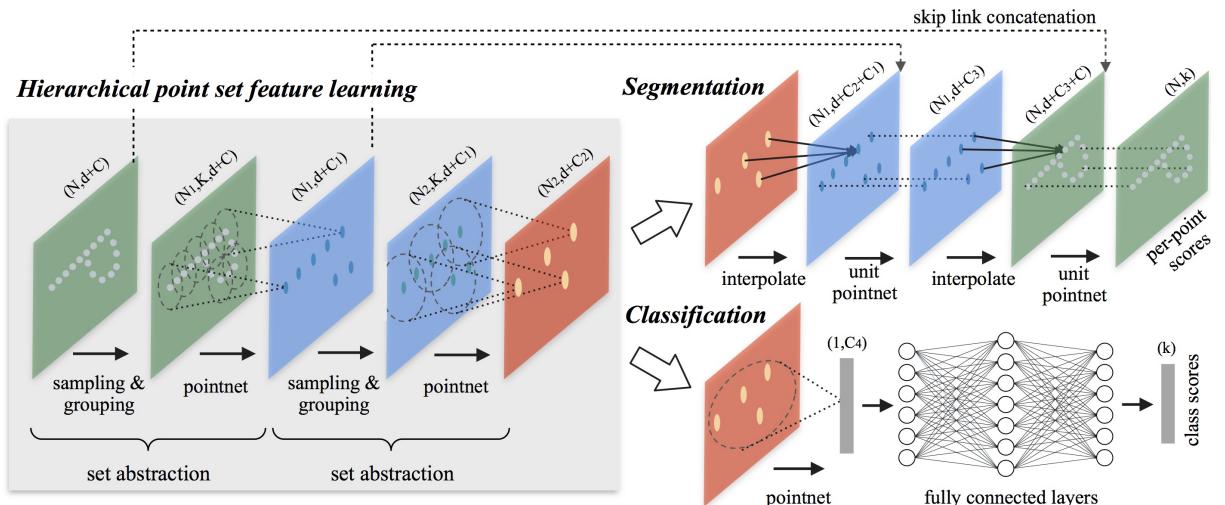


Abbildung 1.10: Architektur von PointNet++
[PF:Web45]

Die Abbildung 1.10 zeigt den Aufbau von PointNet++. Es besteht aus einer Set-Abstraction Ebene. Diese wird für die Berechnung von neuen Merkmalen von Eingangspunkten verwendet. Jede Set-Abstraction dient zur Verarbeitung und Abstrahierung von einer Menge an Eingabepunkten. Dadurch wird eine neue Menge, welche weniger Punkte umfasst, erzeugt. Diese kann durch den Einsatz von PointNet zur Entdeckung von lokalen Merkmalen auf verschiedenen Ebenen der Detailgenauigkeit genutzt werden. [PF:Web45]

Eine Set-Abstraction Ebene hingegen besteht aus einem Sampling-, Grouping- und PointNet Layer. N steht hierbei für die Anzahl der Eingangspunkte, N' repräsentiert

die Anzahl der ausgehenden gesampelten Punkte, d die Dimension der Koordinaten und C wie auch C' die Dimension des Merkmal-Vektors. [PF:Web45]

Der Sampling-Layer wird für die Reduzierung der Punkteanzahl mithilfe von Sampling verwendet. Das bedeutet, dass für eine Eingabe von beispielsweise x_1, x_2, \dots, x_g Farthest Point Sampling²⁷ angewendet wird. Nach der Anwendung ist das Ergebnis eine Teilmenge $x_{i1}, x_{i2}, \dots, x_{ih}$. Für einen zur Menge gehörenden Punkt x_{ij} , gilt, dass dieser den größten Abstand zu $x_{i1}, x_{i2}, \dots, x_{ij-1}$, welche ebenfalls Punkte sind, hat und noch nicht Teil der Menge ist. Durch dieses Verfahren ist es möglich, dass die Punktwolke großflächiger, im Vergleich zu einem zufälligen Auswahlverfahren, abgedeckt werden kann. Die Teilmenge N' identifiziert lokale Merkmale. [PF:Web45]

Der Grouping-Layer ist verantwortlich, lokal gelegene Punkte um die davor identifizierten geometrischen Schwerpunkte zu erfassen. Die resultierende Menge hat die Größe von $N' \times d$. Die Punkte, welche zur Erfassung verwendet werden, werden aus der Menge aller Punkte $N \times (d + C)$ entnommen. Mithilfe einer Ball Query²⁸ bekommt man alle Punkte K innerhalb eines spezifischen Radius um den Schwerpunkt aus N' . Das Ergebnis sind Gruppen von Punktmengen, wodurch eine Matrixerweiterung auf $N' \times K \times (d + C)$ stattfindet. [PF:Web45]

Dennoch ist zu beachten, dass es sich bei K um keinen festen Wert handelt, sondern dieser in Abhängigkeit der Gruppe variieren kann. Die Abhandlung dieser Variation geschieht in der darauffolgenden PointNet Schicht, denn hier entsteht ein Merkmal-Vektor mit fixer Größe aus unterschiedlich vielen Punkten. [PF:Web45]

Am Schluss der Set-Abstraction Ebene werden alle Gruppen im PointNet Layer für die Generierung von lokalen Merkmalen verwendet. Die Eingabematrix dieser Ebene verfügt über eine Größe von $N \times (d + C)$ und die Ausgabe ist $N' \times (d + C')$ groß. [PF:Web45]

Dadurch, dass die Set-Abstraction Ebene hierarchisch aufgebaut ist. verkleinert sich die Punkteanzahl pro Ebene. Die Segmentierung braucht aber die Punktmerkmale für alle ursprünglichen Punkte. Deshalb werden die Merkmale der verkleinerten Punktmengen auf die Ursprünglichen propagiert²⁹. Die punktspezifischen Merkmale, welche in $N_l \times (d + C)$ beschrieben sind, werden auf N_{l-1} propagiert. N_{l-1} steht für die Punkteanzahl bei der Eingabe einer Set-Abstraction Ebene und N_l repräsentiert die Anzahl an gesampelten Punkten nach dieser Ebene l , für die durch die Verwendung von Sampling $N_l \leq N_{l-1}$ gilt. Die Merkmale der N_l Punkte werden auf N_{l-1} bei der Propagation übertragen. Dies geschieht, indem die Eigenschaften der umliegenden Punkte gewichtet geschätzt werden. Die Stärke der Gewichtung hängt hierbei von der

²⁷Auswahl weitest entfernten Punkt, von bereits ausgewählter Teilmenge

²⁸Verfahren zum Abfragen von Punkten in einem Radius eines Schwerpunktes

²⁹Weitergabe von Information zwischen Punkten

Nähe zu dem betrachteten Punkt ab. Darauf werden die geschätzten mit den bereits vorhanden Merkmalen der Set-Abstraction Ebene vereint und in ein *unit pointnet* geleitet, welches die Größe der Merkmalsvektoren reduziert. Diese Prozedur wird für jede Ebene durchgeführt, sodass am Schluss die Merkmale für alle ursprünglichen Punkte vorliegen. [PF:Web45]

Problematisch für das Lernen von Merkmalen ist dennoch die stark variierende Punktwolkendichte. Zum Beispiel können Merkmale von dichten Regionen, nur schwer auf Regionen mit einer geringeren Punktdichte übertragen werden. Wodurch Modelle, welche auf einer geringen Punktwolkendichte trainiert sind, Probleme bei einer hohen Dichte haben. Was bedeutet, dass bei hohen Punktdichten feine regionale Merkmale im Nahbereich gesucht werden können, während bei Geringerer ein größerer Umkreis betrachtet werden sollte. [PF:Web45]

Der Grouping-Layer betrachtet alle Punkte innerhalb der Punktwolke auf einer Skalenebene ohne Berücksichtigung der Dichte oder Struktur der Wolke. Deshalb heißt es *Single-Scale Grouping* genannt. Um dem entgegenzuwirken, muss man die PointNet Schichten an die Dichte anpassen. Durch diesen Vorgang ist es möglich die Merkmale der Regionen zu kombinieren, wobei unterschiedliche Skalen berücksichtigt werden können, wenn es zu Dichteunterschieden kommt. Dafür erstellen die Set-Abstraction Ebenen lokale Merkmale in unterschiedlichen Skalen und kombinieren diese entsprechend der lokalen Punktdichte. PointNet++ bietet hierfür *Multi-Resolution Grouping* und *Multi-Scale Grouping*. [PF:Web45]

Beim *Multi-Scale Grouping* werden auf die Punktwolke Grouping-Layer mit unterschiedlichen Skalen angewandt. Anschließend werden für jede Skala mithilfe von PointNet Layern Merkmale generiert. Nun kann das Netz angepasst an Dichte lernen. [PF:Web45]

Das *Multi-Resolution Grouping* hingegen verwendet zwei Vektoren für die Darstellung von Merkmalen auf der Set-Abstraction Ebene L_i . Beim ersten Vektor handelt es sich um Merkmale der relevanten Unterregionen, welche man durch die Ebene L_{i-1} bekommt. Der zweite Merkmalsvektor verwendet nur ein PointNet und alle Punkte, welche sich in der Region befinden. Die Gewichtung der Vektoren hängt hierbei von der Punktdichte ab. [PF:Web45]

Wenn man die beiden Verfahren vergleicht, stellt sich heraus, dass *Multi-Scale Grouping* rechenintensiver ist. Das liegt vor allem daran, dass hier lokale PointNet-Netze in Verwendung sind, um Skalen in einem großen Umkreis für alle Schwerpunkte abzudecken. Dies macht sich besonders auf der Set-Abstraction Ebene, aufgrund der hohen Anzahl an Schwerpunkten, bemerkbar. [PF:Web45]

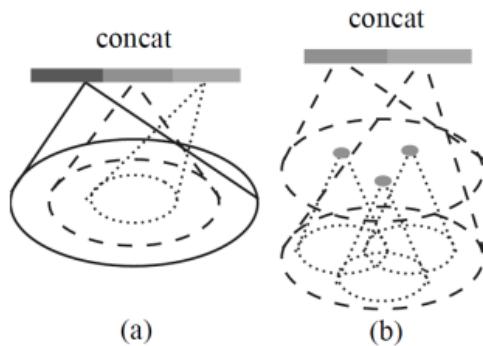


Abbildung 1.11: Darstellung der Grouping-Methoden
[PF:Web45]

Die Abbildung 1.11 zeigt hierbei eine simple grafische Darstellung der Grouping Methoden. (a) zeigt das Multi-Scale Grouping, welches unterschiedliche Skalenebene benutzt und (b) das Multi-Resoultion Grouping, bei dem ein Zugriff auf Merkmale der Unterregionen verwendet wird. [PF:Web45]

1.2.8 Einteilung von LiDAR-Sensoren

Es gibt viele Arten um LiDAR-Sensoren einzuteilen, die häufigsten Unterteilungen sind nach Funktionsweise und nach Dimensionalität der Datenerfassung.

Unterteilung nach Funktionsweise

Die Hauptunterteilung erfolgt in luftgestütztes und terrestrisches³⁰ LiDAR. Beide Unterscheidungsmöglichkeiten haben in der Regel dieselben Bestandteile, nämlich bestehen sie aus einem LiDAR-Sensor, einer Kamera, GPS und einem Trägheitsnavigationssystem. [PF:Web08]

- **Luftgestütztes LiDAR**

Wird entweder an einen Hubschrauber oder an einem Starrflügelflugzeug befestigt. Währenddessen sendet der Sensor Infrarotlaserlichtstrahlen zum Boden und erfasst die zurückkommenden Signale für das Erheben der Daten. Innerhalb von luftgestützten Sensoren kann man noch einmal in topografisch und bathymetrisch³¹ unterteilen. [PF:Web08]

³⁰irdisch

³¹Gewässermessung

- **Topografisches LiDAR** ermöglichen die Erstellung von Oberflächenmodellen, welche in diversen Gebieten, zum Beispiel Stadtplanung, Forstwirtschaft, eingesetzt werden können. [PF:Web08]

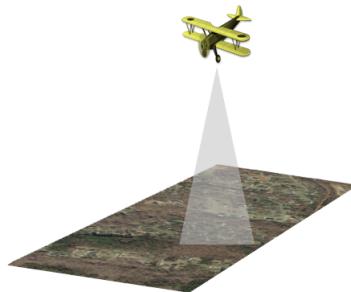


Abbildung 1.12: Topografisches LiDAR
[PF:Web08]

- **Bathymetrisches LiDAR** verfügt über die Fähigkeit Wasser zu durchdringen. Eine große Anzahl dieser Art von Sensoren erfassen zeitgleich Wassertiefe und Höhe und können somit einen Land-Wasser-Übergang ermitteln. Bei der Messung wird neben dem Infrarotlicht, welcher von der Wasserbeziehungsweise Landoberfläche zum Flugzeug reflektiert, auch ein grüner Laser verwendet, welcher die Wasseroberfläche durchdringt. Ihr Einsatzgebiet liegt meist in Küstennähe oder für die Suche von Objekten unter Wasser. [PF:Web08]

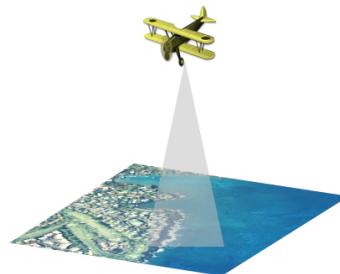


Abbildung 1.13: Bathymetrisches LiDAR
[PF:Web08]

Die Abbildungen 1.12 und 1.13 zeigen den wesentlichen Unterschied der beiden Arten in ihrem Anwendungsgebiet.

- **Terrestrisches LiDAR**

Verwendet augensichere Laser und verfügt über eine enorm dichte und genaue Punkterfassung, was eine genaue Identifikation von Objekten ermöglicht. Die zwei Hauptarten sind mobil und statisch. [PF:Web08]

- Bei einem **mobilen LiDAR** wird eine beliebige Anzahl an Sensoren auf einem beweglichen Fahrzeug befestigt. Mit ihrem Einsatz kann zum Beispiel die Infrastruktur von Straßen analysiert werden. [PF:Web08]
- Bei einem **statischen LiDAR** werden die LiDAR-Punkte von einem statischen Standort, welcher in den meisten Fällen ein Stativ ist, aus gemessen. Die häufigsten Einsatzgebiete sind Bergbau, Vermessungswesen und Ingenieurwesen. [PF:Web08]

Unterteilung nach Dimensionalität der Datenerfassung

Bei der Einteilung in Dimensionalität haben alle LiDAR-Sensoren dasselbe Grundprinzip, der wesentliche Unterschied ist die Anzahl an Dimensionen, welche der Sensor nutzt. [PF:Web10]

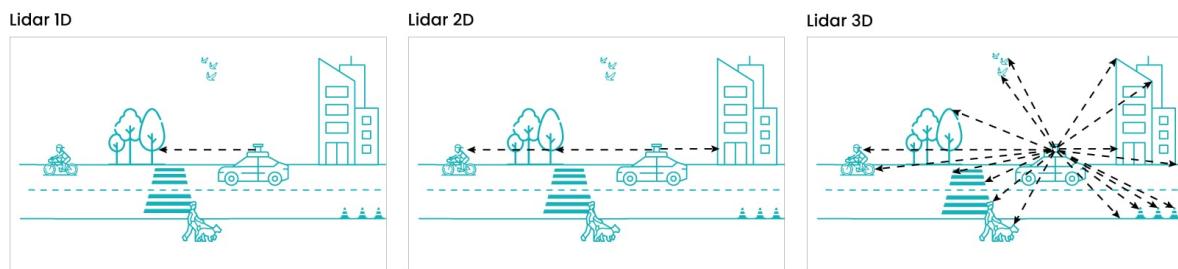


Abbildung 1.14: LiDAR-Sensoren unterteilt in Dimensionalität der Datenerfassung
[PF:Web09]

- **1D-LiDAR**

Werden auch *Linear messende Sensoren* genannt und verwenden einen festen Laserstrahl um den Abstand zwischen 2 Punkten zu messen. Der gemessene Wert liegt auf einer einzigen Achse, welcher die Messung eindimensional macht. Ihr Anwendungsgebiet liegt in Bereichen, wo große Entfernung und schnelle Distanzänderungen gemessen werden müssen. [PF:Web10, PF:Web11]

In der Abbildung 1.15 ist die Funktionsweise eines 1D-LiDAR-Sensors und auch welche Achse verwendet wird, in diesem Fall die x-Achse, zu sehen. Der Quader stellt den Sensor dar, der rote Strahl den Laserstrahl und die graue Fläche das Objekt, zu dem der Abstand gemessen werden soll.

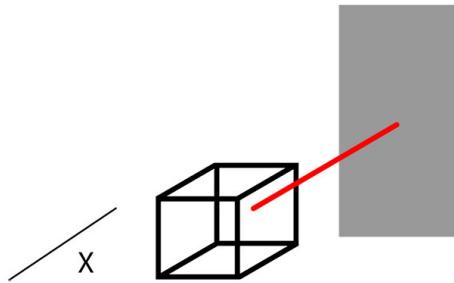


Abbildung 1.15: Funktionsweise eines 1D-LiDAR-Sensors
[PF:Web10]

- **2D-LiDAR**

Verwenden einen rotierenden Laserstrahl, welcher den Abstand durch Impulse, die um eine horizontale Ebene ausgesendet werden, misst. Hierbei werden 2 Dimensionen verwendet, die x- und y-Achse, wie in der Abbildung 1.16 gezeigt wird. Weiters werden eingezeichneten Objekte zeigen wie in Grafik 1.15 den Sensor, den Laserstrahl und das Objekt gezeigt. Der wesentliche Unterschied ist jedoch, dass mehrere Laserstrahlen eingezeichnet sind, da der Sensor rotiert, was der Pfeil unter dem Sensor verstrkend zum Ausdruck bringen soll. [PF:Web10]

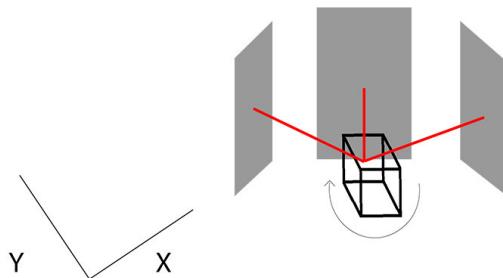


Abbildung 1.16: Funktionsweise eines 2D-LiDAR-Sensors
[PF:Web10]

- **3D-LiDAR**

Hierbei wird das Prinzip des 2D-LiDARs aufgefasst, aber mit dem Unterschied, dass statt einem Laserstrahl mehrere im Einsatz sind. Diese werden auf der vertikalen Achse verteilt. Die Daten erhlt der Sensor von der x-, y- und z-Achse.

Die Grafik 1.17 fasst die selben Grundelemente wie die Abbildungen 1.15 und 1.16 auf, aber mit dem wesentlichen Unterschied, dass die Laserstrahlen nun in smtliche Richtungen ausgesendet werden.

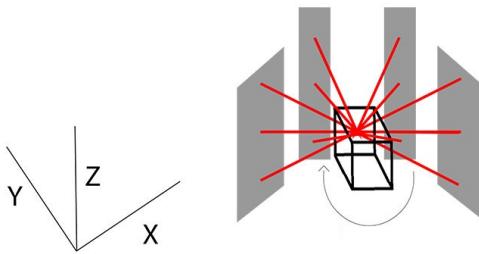


Abbildung 1.17: Funktionsweise eines 3D-LiDAR-Sensors
[PF:Web10]

1.2.9 Vorteile von LiDAR

LiDAR zeichnet sich durch unterschiedliche Vorzüge gegenüber anderen Technologien zur Erfassung und Lokalisierung von Objekten in der Umgebung aus. Die Prägnantesten sind folgende: [PF:Web17]

- **Exakte Genauigkeit**

Erlaubt die Durchführung von akkurate Positions- und Entfernungsmessungen.
[PF:Web17]

- **Anwendungsbreite**

Kann in vielen unterschiedlichen Bereichen wie zum Beispiel Robotik, autonomes Fahren oder Umweltbeobachtung eingesetzt werden. [PF:Web17]

- **Präzise räumliche Darstellung**

Durch die Verwendung von Laserimpulsen, welche fokussiert und kurz sind, wird eine detaillierte Kartierung der Umgebung möglich. [PF:Web17]

- **Effizienz**

Im Vergleich zu anderen Technologien arbeitet LiDAR effizienter und schneller, wodurch eine schnellere Datenerfassung sowie -verarbeitung möglich ist.
[PF:Web17]

1.2.10 Nachteile von LiDAR

Trotz der Vorteile, welche LiDAR bietet, hat diese Technologie auch Nachteile. Die Wesentlichsten sind folgende: [PF:Web17]

- **Kostenaufwand**

Wenn es sich bei dem Verwendungszweck, um einen anspruchsvollen Anwendungsbereich wie Vermessung handelt, können LiDAR-Systeme sehr teuer werden. [PF:Web17]

- **Sensibilität für Umweltbedingungen**

Trotz der Fähigkeit unabhängig von Lichtverhältnissen arbeiten zu können, ist es möglich, dass das System durch Rauch, Staub, Nebel oder andere Partikel beeinträchtigt wird. [PF:Web17]

- **Sichtlinienabhängigkeit**

Um eine genaue Messung durchführen zu können, muss eine klare Sichtachse³² zwischen dem Objekt und Sensor vorhanden sein. Wenn das nicht der Fall ist, besteht die Möglichkeit, dass die Messung ungenau oder erst gar nicht möglich ist. [PF:Web17]

- **Energieverbrauch**

Für die Erzeugung von Laserlichtimpulsen und die Datenverarbeitung wird viel Energie benötigt. [PF:Web17]

- **Einschränkung in der Möglichkeit der Darstellung**

LiDAR verfügt lediglich über die Fähigkeit Form, Lage, Entfernung und Bewegung zu erfassen, aber nicht genau darzustellen. In Bezug auf autonomes Fahren wirkt sich dieser Nachteil besonders auf die Erkennung der Bedeutung von Verkehrsschildern aus. [PF:Web19]

³²geradlinige Verbindung

1.3 Vergleich mit anderen Sensortechnologien

1.3.1 Überblick über andere Sensoren

Heutzutage bietet die Technik eine Vielfalt von Sensoren zur Auswahl, wie in Abbildung 1.18 zu sehen ist. Hierbei wird bevor eine konkrete Kategorisierung stattfindet eine Unterteilung nach Arbeitsprinzip vorgenommen. Die 3 Hauptprinzipien sind *elektromagnetisch*, *optisch* und *akustisch*. [PF:Web23]

Innerhalb dieser Einteilung kann noch einmal nach der Sensorart eingeteilt werden. Beispiele für *elektromagnetische* Sensoren wären Car-2-X und Radar. Video, LiDAR, Laser und Infrarot gehören zu der Oberkategorie *optisch* und Ultraschall zu *akustisch*. Des Weiteren kann innerhalb der Sensoreinteilung noch nach der Art gegliedert werden. [PF:Web23]

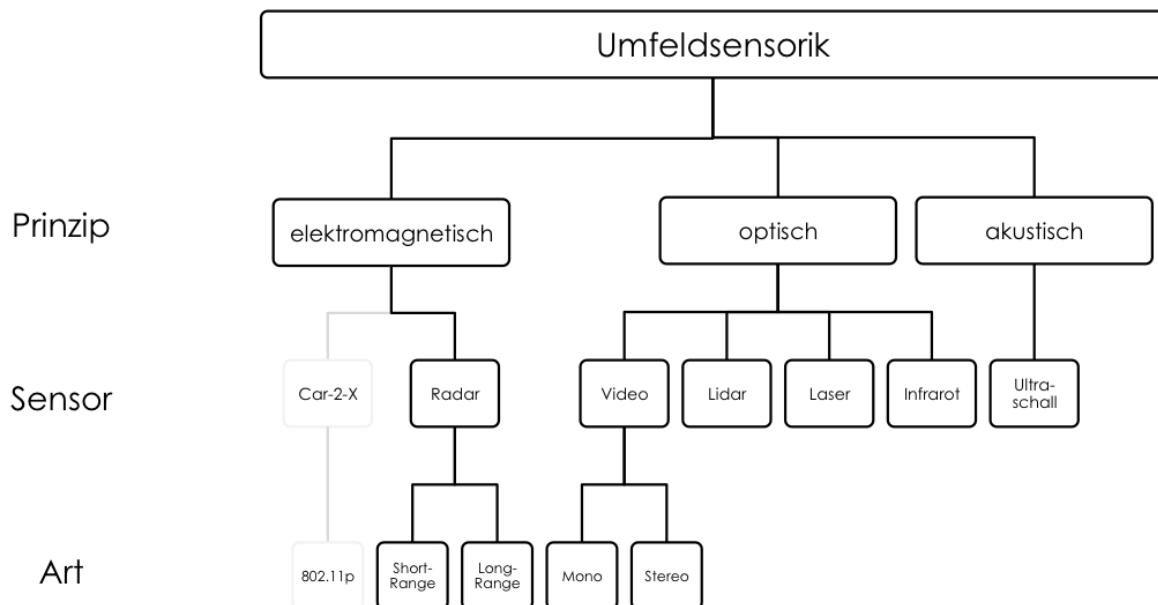


Abbildung 1.18: Unterteilung von Sensoren
[PF:Web23]

1.3.2 TOF-Kamera

Bei TOF-Kameras handelt es sich um 3D-Kamerasysteme, die für die Distanzmessung das *time of flight-Prinzip* verwenden. Das bedeutet, dass Lichtimpulse auf den abgebildeten Bereich gerichtet werden, um die Szene zu beleuchten. Die Kamera misst daraufhin für jeden Punkt innerhalb dieses Bildes, wie lange es dauert, bis das Licht vom Ausgangspunkt zum Objekt und wieder zurück gelangt. [PF:Web57]

Der Messbereich hat bei dieser Messmethode eine Spanne von paar Dezimetern bis hin zu etwa 40 Metern. [PF:Web57]

Bestandteile

Eine übliche und aufs ein minimalste reduzierte TOF-Kamera besteht aus diesen Elementen:

- **Beleuchtungseinheit**

Diese Komponente wird zur Beleuchtung des Bereiches verwendet. Dabei kommen meist LEDs beziehungsweise Laserdioden, welche über die Fähigkeit schnell die Lichtintensität zu ändern verfügen, zum Einsatz. [PF:Web57]

- **Optik**

Hierbei wird das durch die Umgebung reflektierte Licht eingesammelt, wodurch der Bereich auf dem Sensor abgebildet wird. Durch einen optischen Bandpassfilter gelangen nur die Wellenlänge, welche zur Beleuchtung verwendet werden zum Sensor. Durch diesen Filter werden die meisten störenden Hintergrundlichter eliminiert. [PF:Web57]

- **Bildsensor**

Dieser wird für die separate Laufzeitmessung für jeden Punkt verwendet. Im Vergleich zu zum Beispiel Digitalkameras sind die Pixel komplizierter aufgebaut, denn dieser muss neben dem einfallenden Licht auch die Laufzeit messen. Dabei erreichen die Pixel eine Seitenlänge von bis zu 100 µm. [PF:Web57]

- **Ansteuerelektronik**

Für eine möglichst exakte Genauigkeit wird eine anspruchsvolle Elektronik für die Ansteuerung der Beleuchtung und des Sensors verwendet. [PF:Web57]

- **Auswertung**

Das Kamerasystem berechnet meist direkt die Distanz mithilfe der zuvor gemessenen Werte. [PF:Web57]

Funktionsweise

Die Laufzeit kann wie folgt berechnet werden:

$$t_D = 2 \frac{D}{c_{\text{luft}}}$$

- t_D ... Laufzeit
- D ... Entfernung
- c_{luft} ... Lichtgeschwindigkeit [PF:Web57]

Die maximale Distanz, welche durch eine TOF-Kamera abgedeckt werden kann, wird durch die Formel

$$D_{\max} = \frac{c_{\text{luft}} t_0}{2}$$

berechnet, wobei

- D_{\max} ... maximale Distanz
- c_{luft} ... Lichtgeschwindigkeit
- t_0 ... Pulslänge [PF:Web57]

Zum Beispiel arbeitet das Pixel mit G_1 und G_2 , welche als Schalter fungieren und S_1 und S_2 als Speicherelemente. Hierbei werden die Schalter durch ein Pulssignal, mit derselben Länge des Lichtpulses, angesteuert. G_2 ist im Vergleich zu G_1 um eine Pulslänge verschoben. Wenn das reflektierte Licht nun verzögert auf das Pixel auftrifft, gelangt nur ein Signalteil in S_1 , während der andere in S_2 gesammelt wird. Abhängig von der Distanz verändert sich das Verhältnis von S_1 und S_2 . Mithilfe dieser Speicher- elemente kann die Distanz mit folgender Formel berechnet werden: [PF:Web57]

$$D = \frac{c_{\text{luft}} t_0}{2} \cdot \frac{S_2}{S_1 + S_2}$$

- D ... Distanz
- c_{luft} ... Lichtgeschwindigkeit
- t_0 ... Pulslänge
- S_1, S_2 ... Speicherelemente [PF:Web57]

Anwendungsgebiete

TOF-Kameras können in vielen Bereichen eingesetzt werden:

- **Automobilbranche**

Hierbei kommt die Technologie zum Beispiel für Sicherheits- und Fahrassistenzsensoren zum Einsatz. Darunter fallen auch der Notbremsassistent oder die Überprüfung der Fahrposition. [PF:Web57]

- **Robotik**

Durch das Umgebungsbild, welches in Echtzeit abgebildet wird, wird ein weites Spektrum an Möglichkeiten in der Robotik geöffnet. Dieses reicht von der Hindernis Ausweichung bis hin zu der Verfolgung von Personen. [PF:Web57]

- **Medizin**

Patienten können mithilfe von TOF-Kameras exakt im klinischen Umfeld positioniert werden. Zudem können berührungslos Atemsignale gemessen werden oder die Körpergröße und das Gewicht im Liegen bestimmt werden. Das ist besonders hilfreich in Notfallsituationen, bei zum Beispiel bewusstlosen Patienten. [PF:Web57]

Vorteile

Im Vergleich zu anderen Technologien haben TOF-Kameras folgende Vorteile:

- **simpler Aufbau**

Zu den Bestandteilen gehören im Gegensatz zu Lasersensoren keine beweglichen Teilen. Weiters ist der Platzbedarf auch kleiner, da das Objektiv und die Beleuchtung relativ nahe beieinander liegen. [PF:Web57]

- **Geschwindigkeit**

Durch eine Aufnahme der TOF-Kamera kann die komplette Szenerie abgebildet werden und durch eine Bildrate von maximal 160 Bilder pro Sekunde, werden Echtzeitanwendungen ermöglicht. [PF:Web57]

- **effiziente Datenauswertung**

Mithilfe der Distanzinformationen ist es möglich, nur interessante Bildbereiche zu extrahieren, dazu wird ein Distanzschwellwert gesetzt, welcher bestimmt, welche Pixel betrachtet werden. [PF:Web57]

Nachteile

Trotz aller Vorteile gibt es auch ein paar Nachteile:

- **Störung durch andere Systeme**

Wenn mehrere Systeme zur selben Zeit in Betrieb sind, ist es möglich, dass sich diese untereinander stören, wodurch es zu verfälschten Distanzwerten kommen kann. Dieses Problem kann entweder durch unterschiedliche Frequenzen oder durch eine Zeitsteuerung, welche dafür sorgt, dass maximal eine Beleuchtung gleichzeitig aktiv ist, behoben werden. [PF:Web57]

- **Mehrfachreflexion**

Dadurch, dass die gesamte Szenerie und nicht nur einzelne Punkte beleuchtet werden, kann es passieren, dass ein Objekt das Licht mehrfach reflektiert. Das führt dazu, dass die Distanz größer eingeschätzt wird. [PF:Web57]

1.3.3 Radar

Radar, die Abkürzung für *radio detection and ranging* beziehungsweise *radio direction and ranging*, bezeichnet unterschiedliche Verfahren und Geräte zur Erkennung und Ortung auf Basis von elektromagnetischen Wellen im Radiofrequenzbereich³³. [PF:Web25]

Funktionsweise

Der Sensor sendet Radiowellen aus, um die Zeit zu messen, wie lange es dauert bis die Welle ausgesendet, am Objekt reflektiert und abgeprallt wird. Bei dem Signal handelt es sich konkreter um elektromagnetische Wellen, wie auch die Einteilung in Abbildung 1.18 zeigt. Anhand der Welle werden der Winkel, die Entfernung oder die Geschwindigkeit eines Objekts durch die Frequenzverschiebung des reflektierten Signals bestimmt. [PF:Web24]

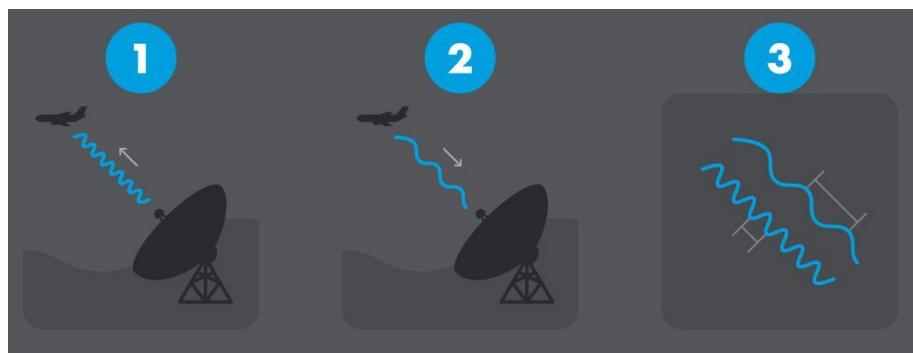


Abbildung 1.19: Funktionsweise Radar
[PF:Web24]

In der Grafik 1.19 wird die Funktionsweise noch einmal grafisch dargestellt. Der erste Schritt zeigt hierbei, wie die elektromagnetischen Wellen ausgesendet werden, welche im nächsten Abschnitt reflektiert und zum Radar-Sensor zurückgesendet werden. Im letzten Schritt wird die Verschiebung der Frequenz der reflektierten Welle gemessen. [PF:Web24]

³³Funkwellen

Bei der Messung der Entfernung wird auf das Prinzip des *Time-of-Flights* zurückgegriffen:

$$R = \frac{c \cdot t}{2}$$

- R ... Entfernung
- c ... Lichtgeschwindigkeit
- t ... gemessene Laufzeit [PF:Web26]

Unterschied zu LiDAR

LiDAR und Radar unterscheiden sich in etlichen Punkten, wie zum Beispiel:

- **Wellenlänge**

Der primäre Unterschied ist die verwendete Art der Wellen. LiDAR verwendet Lichtwellen oder Laser und Radar Radiowellen. [PF:Web24]

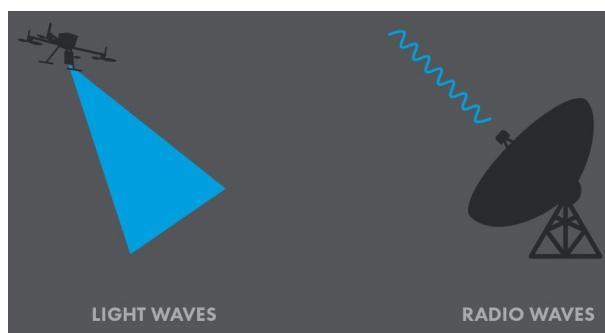


Abbildung 1.20: Unterschied der Wellenarten von LiDAR und Radar
[PF:Web24]

- **Struktur und Funktionsweise**

LiDAR verwendet für die Erfassung des reflektierten Lichtes eines Laserstrahles Bildsensoren. Die Daten, welcher der Sensor gesammelt hat, werden daraufhin zu einer Punktwolke verarbeitet. Durch diese Prozedur werden unter anderem digitale Karten erstellt. [PF:Web24]

Bei Radarsystemen hingegen erfolgt die Umwandlung der reflektierten Wellen in ein Bild auf einem Bildschirm. [PF:Web24]

- **Reichweite**

LiDAR neigt dazu, eine kleinere Reichweite verglichen zu Radar zu haben. Dennoch bieten sie im Nahbereich die Möglichkeit für exakte Messungen. Auf der anderen Seite weist Radar eine größere mögliche messbare Reichweite auf und bietet somit einen immensen Vorteil weit entfernte Objekte zu erfassen. [PF:Web24]

- **Kosten**

LiDAR ist gegenüber Radar in der Herstellung und Implementierung im Regelfall kostengünstiger. [PF:Web24]

Anwendungsbereiche

Radar kann vielseitig eingesetzt werden, einige Einsatzmöglichkeiten sind folgende:

- **Gesundheit**

Sehr sensible Geräte sind in der Lage Herzschläge und minimale Atmung zu erkennen. [PF:Web35]

- **Windparkanlagen**

In diesem Zusammenhang kann die Technologie für die Erkennung von kleinsten Abweichungen der Rotorblätter eingesetzt werden. InnoSenT arbeitete in dem Zeitraum von 2015 bis 2018 erstmals damit. Weiters können Vögel sowie andere Objekte erkannt werden und zum Verlangsamten der Blätter führen. [PF:Web35]

- **Fahrzeugausrüstung**

Assistenzsysteme, welche in Fahrzeugen verbaut sind, verwenden häufig auch die Radar-Technologie. Zum Beispiel gehören hierzu Einparkhilfen, einhalten des Sicherheitsabstandes und die automatische Aktivierung einer Dashcam kurz vor dem Aufprall. [PF:Web35]

- **Überwachung des Straßenverkehrs**

Hier wird Radar für die Überwachung des Verkehrs verwendet. Bei der Überschreitung der zulässigen Geschwindigkeit löst der Sensor aus. [PF:Web36]

1.3.4 Ultraschall

Schall mit Frequenzen, welche sich über dem Hörfrequenzbereich des Menschen befinden, wird als *Ultraschall* bezeichnet. Der Frequenzbereich hierbei liegt zwischen 20 kHz und 1 GHz. Schall mit einer Frequenz unterhalb von 16 Hz wird als *Infraschall* bezeichnet und über 1 GHz als *Hyperschall*. [PF:Web27]

Funktionsweise

Der Ultraschallsensor sendet hierbei einen kurzen und hochfrequenten Schallimpuls zyklisch aus, welcher sich mit Schallgeschwindigkeit in der Luft fortbewegt. Wenn dieser auf ein Objekt trifft, wird er reflektiert und dessen Echo kommt zurück zum Sensor. Die Entfernung wird durch die Zeitspanne zwischen dem Aussenden und Empfangen des Signals berechnet. Dieser Ablauf wird in der Abbildung 1.21 veranschaulicht. [PF:Web28]

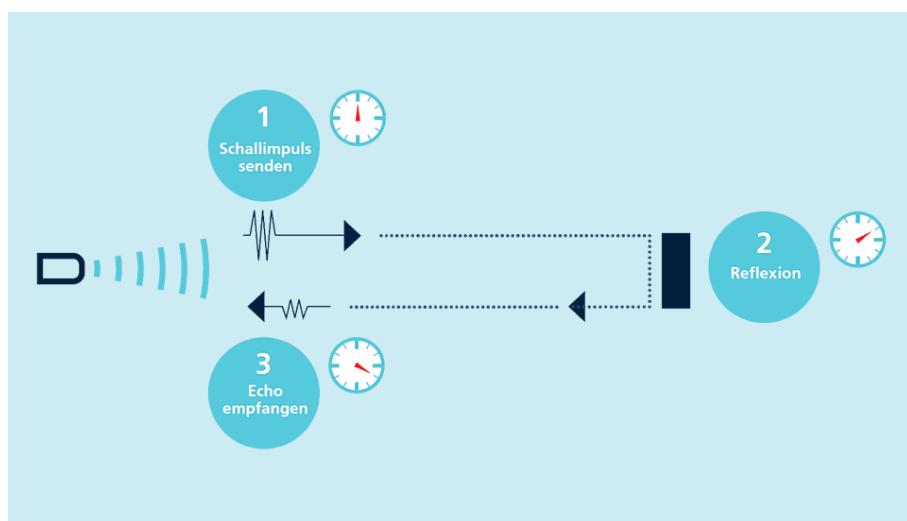


Abbildung 1.21: Funktionsweise Ultraschall
[PF:Web28]

Da es sich hierbei um eine Schall-Laufzeitmessung handelt findet eine exzellente Ausblendung des Hintergrunds statt. Weiters bieten Ultraschallsensoren die Möglichkeit durch Farbnebel oder staubige Luft hindurchzumessen. [PF:Web28]

Für die Berechnung der Entfernung kann folgende Formel verwendet werden:

$$L = \frac{c \cdot t}{2}$$

- L ... Entfernung
- c ... Ultraschallgeschwindigkeit
- t ... gemessene Laufzeit [PF:Web30]

Unterschied zu LiDAR

Ultraschall und LiDAR ähneln sich zwar in vielen Punkten, aber es gibt trotzdem einige Unterschiede:

- **Funktionsprinzip**

Hierbei liegt der Hauptunterschied bei der verwendeten Frequenz, mit welcher die Sensoren arbeiten. LiDAR verwendet für die Entfernungsmessung Laserstrahlen statt Schallwellen. [PF:Web29]

- **Reichweite**

Die maximale Reichweite von 10 Metern, welche Ultraschallsensoren erreichen können, kann von LiDAR, welcher kilometerweite Distanzen schafft zu messen, übertroffen werden. [PF:Web29]

- **Kosten**

LiDAR ist im Vergleich zu Ultraschall in der Herstellung und Implementierung im Regelfall kostengünstiger. [PF:Web29]

Anwendungsgebiete

Die Bereiche in denen Ultraschall eingesetzt wird sind vielseitig:

- **Laufzeitmessung**

Hierbei kann die Technologie für Tiefenmessungen und Untersuchungen des Meeresbodens verwendet werden. Ebenso kann der Füllstand ohne Berührung gemessen werden. [PF:Web43]

- **Übertragung von Information**

In dem Zeitraum von circa 1950 bis 1970 wurde Fernbedienungen für Fernseher damit ausgestattet, sowie das Signal von elektronischen Schaltungen verzögert. [PF:Web43]

- **Medizin**

In diesem Feld ist Ultraschall besonders verbreitet. Sei es bei der Entfernung von Zahnstein bis hin zum Aufschuss von Zellen. [PF:Web43]

- **Arbeitsschutz**

Bei der Produktion innerhalb der Industrie wird die Technik unterschiedlich eingesetzt. Zum Beispiel bei der Reinigung oder beim Schneiden, Schweißen, Bohren, aber auch bei der Überprüfung des Materials. [PF:Web43]

1.3.5 Sensordatenfusion

Die Sensordatenfusion versucht die Stärken der verschiedenen Sensorprinzipien optimal zu vereinen. Durch diese Vereinigung können Informationen gewonnen werden, welche einzelne Sensoren nicht zur Verfügung stellen könnten. Zudem kann auch der Messbereich vergrößert werden, sowie die Zuverlässigkeit der Messung. Es werden somit mehrere Technologien zur Datenerfassung verwendet. Die Messprinzipien können sich untereinander nun bei der Erfassung von Daten beziehungsweise Objekten überprüfen. [PF:Web47]

Die grundlegenden Ziele sind zusammengefasst:

- **Erhöhung der Zuverlässigkeit**

Erkennen eines Ausfalles von Sensoren und ihn ausgleichen, sodass nicht das gesamte System ausfällt. [PF:Web49]

- **Förderung der Genauigkeit**

Messfehler anderer Sensoren sollen ausgeglichen werden. [PF:Web49]

- **Erhöhung der Detektionswahrscheinlichkeit**

Das Gesamtsystem erkennt Objekte beziehungsweise Hindernisse, welche einzelne Sensoren wegen Beeinträchtigungen durch Umweltbedingungen nicht wahrnehmen können. [PF:Web49]

- **Vergrößerung des Sichtbereiches**

Mehrere Sensoren vergrößern den Sichtbereich des Gesamtsystems. [PF:Web49]

- **zusätzlicher Informationsgewinn**

Weitere Sensoren liefern mehr Informationen als ein Einziger. [PF:Web49]

Ebenen der Sensorfusion

Es gibt unterschiedliche Zeitpunkte, wann die Fusion der unterschiedlichen Sensordaten geschehen kann.

Bei der *Low-Level-Fusion* (LLF) werden alle erfassten Daten zu einer einzigen Recheneinheit zusammengeführt, bevor die Verarbeitung stattfindet. Zum Beispiel werden für ein besseres Verständnis der Größe und Form des erkannten Objektes die Pixel, welche von Kameras erkannt wurden und Punktwolken der LiDAR Technologie zusammengeführt. Durch eine Recheneinheit gibt es eine Vielzahl an späteren Anwendungen, da diese Einheit über alle Daten verfügt. Dennoch mit dem Nachteil, dass

diese großen Datenmengen zu einer gesteigerten Komplexität bei Berechnungen führen, was zu enormen Kosten der Hardware führt. [PF:Web48]

Während *Mid-Level-Fusion* (MLF) die Objekte zuerst durch die Sensoren erkennt und darauf mithilfe eines Algorithmus fusioniert. Was bedeutet, dass beispielsweise ein LiDAR-Sensor und Kamera ein Objekt einzeln erkennen. Diese Ergebnisse werden dann zusammengeführt, wodurch eine bessere Schätzung über Geschwindigkeit und Position möglich wird. Hierbei handelt es sich, um den am einfachsten zu implementierenden Prozess, aber dennoch besteht das Risiko, dass wenn ein Sensor ausfällt, die gesamte Fusion nicht richtig funktioniert. [PF:Web48]

Zwischen der *High-Level-Fusion* (HLF) und der Mid-Level-Fusion herrschen Ähnlichkeiten, aber dennoch ein großer Unterschied, denn für die HLF werden für jeden Sensor Erkennung- und Tracking-Algorithmen implementiert und diese Ergebnisse dann zusammengeführt. Diese Methode hat somit dieselben Probleme wie MLF, denn wenn ein einzelner Sensor Fehler aufweist, kann das zu der Beeinträchtigung des Gesamt-systems führen. [PF:Web48]

Sensorfusion im autonomen Fahren

Die Fusion von Sensoren spielt besonders im autonomen Fahren eine große Rolle. Denn dieses Gebiet umfasst viele Einsatzmöglichkeiten die jeweils verschiedene Parameter, Betriebsbedingungen und Reichweiten haben. Zum Beispiel eignen sich Kameras zur Objektdentifizierung, sind aber sehr empfindlich was die Änderung des Wetters betrifft. Radarsensoren gleichen das aus, denn sie funktionieren bei so gut wie allen Wetterbedingungen. Dafür sind sie nicht in der Lage, exakte 3D-Karten zu entwerfen. Zur Erfassung der Umgebung kann aber LiDAR verwendet werden, dessen Sensoren aber eine höhere Preisklasse haben. Die Verwendung von mehreren Sensoren kann aber zu Redundanz der Daten führen, wenn Sensoren dasselbe Objekt gleichzeitig erkennen. [PF:Web48]

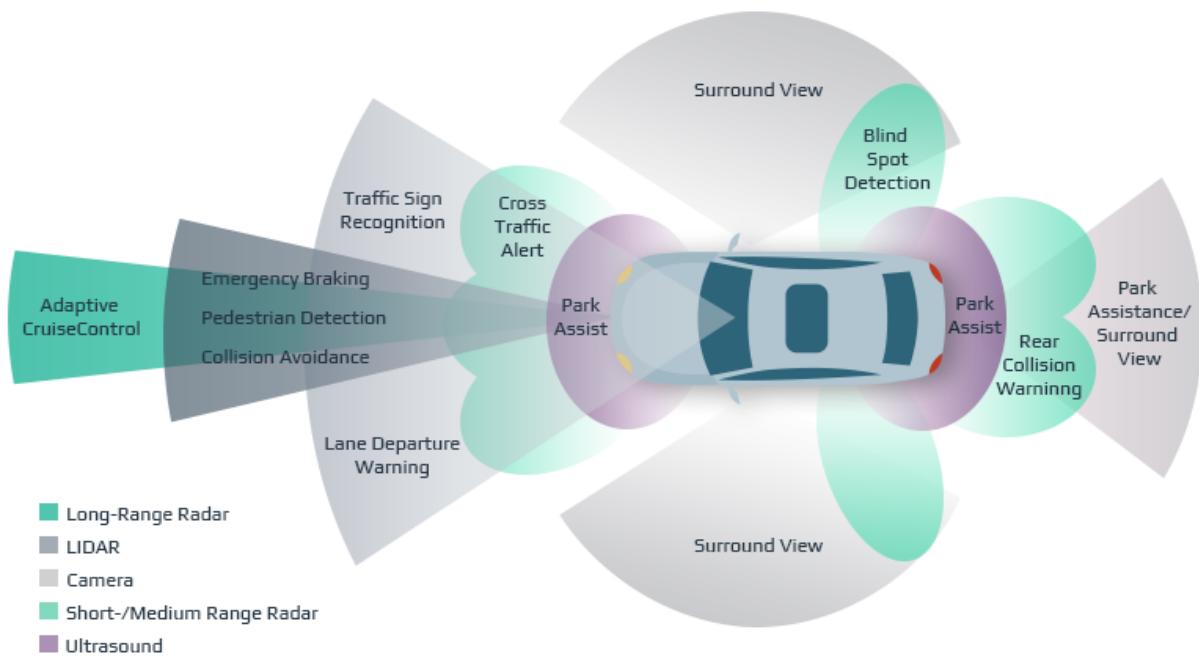


Abbildung 1.22: Beispielhafte Sensorfusion eines Autos
[PF:Web50]

Die Abbildung 1.22 zeigt hierbei ein Beispiel wie Radar, LiDAR, Kamera und Ultraschallsensoren an einem Fahrzeug angebracht werden können:

- Long³⁴-Range Radar wird hier für die anpassungsfähige Geschwindigkeitsregelung eingesetzt.
- Short³⁵-/ Medium³⁶ Range Radare werden für die Warnung vor seitlich kommenden Verkehr, dem Toten-Winkel und Auffahrunfällen verwendet.
- LiDAR findet Einsatz bei Notbremsungen, Fußgängererkennung und Kollisionsvermeidung.
- Kameras werden bei der Erkennung von Verkehrszeichen, Warnung bei Verlassen der Spur, Rundumüberblick über das Fahrzeug und als Parkassistent auf weitere Distanz eingesetzt.
- Ultraschall findet ebenfalls Gebrauch als Parkassistent aber auf niedriger Distanz als Kameras. [PF:Web50, PF:Web51]

³⁴Reichweite bis zu 250 Metern

³⁵Reichweite bis zu 100 Metern

³⁶Reichweite bis zu 50 Metern

1.4 Bezug auf das Projekt

1.4.1 Auswahl LiDAR-Sensor

Für die Hinderniserkennung wird ein entsprechender LiDAR-Sensor benötigt, welcher auf dem Modellauto befestigt werden soll. Nach einer Recherche kamen 3 Sensoren in Frage:

- WayPonDEV Slamtec RPLIDAR A2M12 [PF:Web40]
- WayPonDEV DTOF D300 [PF:Web41]
- Slamtec RPLIDAR A1 [PF:Web42]

Da es sich bei dem Projekt um ein Schulprojekt handelt, ist das Budget begrenzt. Das führt dazu, dass WayPonDEV Slamtec RPLIDAR A2M12 als Option wegfällt.

Für die finale Entscheidung wurden die übriggebliebenen Optionen mithilfe der Scoring Methode miteinander verglichen.

Bei der Scoring Methode handelt es sich um einen Bewertungsprozess, der mögliche Optionen anhand verschiedener Kriterien vergleicht. [PF:Web55]

Kriterien

- **Dimension**
Abmessungen beziehungsweise die Größe (Länge, Breite und Höhe) des Sensors, dies ist wichtig, denn der Sensor muss auf dem Modellauto Platz haben.
- **Gewicht**
Beschreibt die Masse des Sensors.
- **Messbare Distanz**
Spiegelt über welche Entfernung der Sensor messen kann.
- **Genauigkeit**
Zeigt wie präzise Objekt durch den LiDAR-Sensor wahrgenommen werden.
- **2D**
Zeigt, dass der Sensor dazu in der Lage ist, Daten zweidimensional zu erfassen. Diese Einschränkung spielt eine entscheidende Rolle im Verarbeitungsprozess der Daten.

- Preis**

Dient dazu, dass Budget im entsprechenden Rahmen eines Schulprojektes zu halten.

Einordnung der Punkte

Für die Einordnung der einzelnen Kriterien wird eine Bewertungsskala von 1 bis 5, welche in der Tabelle 1.3 näher erläutert wird, verwendet.

| Bewertung | Beschreibung |
|-----------|------------------|
| 1 | schlecht |
| 2 | nicht so gut |
| 3 | durchschnittlich |
| 4 | gut |
| 5 | sehr gut |

Tabelle 1.3: Bewertungsskala

Scoring Methode

Nach der Definition der Bewertungsskala und Bewertung, werden die Kriterien noch gewichtet und schlussendlich die Scoring Methode, wie in der Tabelle 1.4 zu sehen ist, für die übrigen Sensoren angewandt.

| Kriterium | Gewicht | WayPonDEV DTOF D300 | | Slamtec RPLIDAR A1 | |
|------------------|-------------|---------------------|----------------|--------------------|----------------|
| | | Bew.- Punkte | Bew. × Gewicht | Bew.- Punkte | Bew. × Gewicht |
| Dimension | 15% | 4 | 60 | 3 | 45 |
| Gewicht | 10% | 5 | 50 | 1 | 10 |
| messbare Distanz | 15% | 4 | 60 | 4 | 60 |
| Genauigkeit | 20% | 3 | 60 | 4 | 80 |
| 2D | 10% | 5 | 50 | 5 | 50 |
| Preis | 30% | 5 | 150 | 4 | 120 |
| Gesamt | 100% | 430 | | 365 | |

Tabelle 1.4: Bewertung der Lidar-Sensoren
[PF:Web41, PF:Web42]

Erkenntnis und Entscheidung

Bei Betrachtung der Scoring Methode fällt folgendes auf:

- beide Sensoren verfügen über die Fähigkeit die Daten zweidimensional zu erfassen
- in fast allen Kategorien ist der Sensor WayPonDEV DTOF D300 gegenüber dem Slamtec RPLIDAR A1 überlegen

Schlussendlich fiel die Wahl auf WayPonDEV DTOF D300, da dieser alle Kriterien, welche für dieses Projekt von Bedeutung sind, überdurchschnittlich erfüllt und auch bei der Scoring Methode besser abschneidet.



Abbildung 1.23: WayPonDEV DTOF D300
[PF:Web41]

Der auserwählte Sensor, welcher in Abbildung 1.23 zu sehen ist, wurde am Dach des Modellautos, wie in Grafik 1.24 ersichtlich ist, montiert. Die rote Markierung zeigt den montierten Sensor.



Abbildung 1.24: Montierter Sensor am Modellauto

1.4.2 Empfangen und Visualisierung der LiDAR-Daten

Für das Empfangen und die Visualisierung der Daten des Sensors, wurden Programme in der Programmiersprache Python verfasst, welche durch Kommentare innerhalb des Codes erklärt werden. Die Visualisierung erfolgt hierbei in 2D, da der Sensor nur über diese Fähigkeit verfügt.

Konstanten

Dieser Abschnitt zeigt die Konstanten, welche benötigt werden:

```
1 import platform #Schnittstelle zur Information der Plattform
2
3 LOCAL = False    #boolescher Wert ob Anwendung lokal oder nicht
4 HOST_INTERNAL = "jetson-orin"      #interner Hostname
5 HOST = "localhost" if LOCAL else HOST_INTERNAL #HOST basierend auf LOCAL
6 PORT = 8080        #Port
7
8 URL_LOCAL = "http://localhost:8000"          #lokale URL
9 URL_INTERNAL = f"http://{HOST_INTERNAL}:8000"  #interne URL
10 URL = URL_LOCAL if LOCAL else URL_INTERNAL   #URL basierend auf LOCAL
11
12 BLACK = (0, 0, 0)    #RGB-Wert der Farbe Schwarz
```

Listing 1.1: Konstanten/ constants.py

Funktionen

Des Weiteren werden insgesamt 3 Funktionen benötigt:

```

1 import colorsys      #Konvertierung zwischen Farbdarstellungen
2 import random        #Generierung von Zufallsvariablen
3
4 import numpy as np   #numerische Operationen
5 import requests       #HTTP-Anfragen
6
7 from constants import URL    #Konstante aus "constants.py"
8
9 #Berechnung der Dimension eines Vektors (Satz des Pythagoras)
10 def get_magnitude(x, y):
11     return np.sqrt(x ** 2 + y ** 2)
12
13
14 #Umwandlung Polarkoordinaten (Winkel und Abstand) in kartesische
15 #Koordinaten (x,y)
15 def polar_to_cartesian(angle, distance):
16     x = distance * np.cos(np.deg2rad(angle))
17     y = distance * np.sin(np.deg2rad(angle))
18     return x, y
19
20 #Konvertierung von HSV(Farbton, Farbkraft, Helligkeit) in RGB
21 def hsv2rgb(h, s, v):
22     return tuple(np.round(i * 255) for i in colorsys.hsv_to_rgb(h / 255, s
23                           / 255, v / 255))

```

Listing 1.2: Methoden/ functions.py

Pygame-Programm

Nach der Erstellung der Funktionen und Konstanten, wird ein Pygame³⁷ Programm erstellt, welches die durch den Sensor empfangenen Punkte visuell darstellt: [PF:Web56]

```

1 import socket      #Netzwerkkommunikation
2 import struct      #Packen/ Entpacken von Daten
3 import pygame      #GUI
4 import math        #mathematische Operationen
5 import functions
6 from constants import HOST_INTERNAL, PORT, BLACK      #Konstanten aus "
6   constants.py"
7
8 #Socket-Verbindung
9 client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

```

³⁷Python-Programmbibliothek für die Programmierung von Spielen

```
10 client.connect((HOST_INTERNAL, PORT))

11 #Initialisierung von Pygame
12 pygame.init()
13 pygame.display.set_caption('2D LiDAR Point Cloud Visualization')
14 screen = pygame.display.set_mode((800, 600), pygame.RESIZABLE)
15 clock = pygame.time.Clock()
16 font = pygame.font.SysFont('Consolas', 20)

17 #Definition von Konstanten und Initialisierung von Variablen
18 PACKET_SIZE = 8
19 MAX_POINTS = 360
20 lidar_data = []
21 running = True

22 while running:
23     #Durchgehen aller Events, welche von Pygame erfasst wurden
24     for event in pygame.event.get():
25         #wenn es sich um das Quit-Event handelt, wird die Schleife beendet
26         if event.type == pygame.QUIT:
27             running = False

28     #Entfernen bereits vorhandener Punkte
29     screen.fill(BLACK)

30     #Empfang der Daten via Socket-Verbindung
31     data = client.recv(PACKET_SIZE)

32     i = 0
33     #Verarbeiten der Datenpakete
34     while len(data) == PACKET_SIZE:
35         i += 1
36         #Lesen des Winkels als Gleitkommazahl
37         angle = struct.unpack('f', data[0:4])[0]
38         #Lesen der Distanz als unsigned Integer (u_int16)
39         dist = struct.unpack('H', data[4:6])[0] / 15

40         #Umwandlung der polaren Koordinaten in Kartesische
41         x, y = functions.polar_to_cartesian(90 - angle, dist)

42         #Punkte in Liste aufnehmen
43         lidar_data.append((x, y, angle))
44         #Empfangen neuer Daten um Schleife fortzusetzen
45         data = client.recv(PACKET_SIZE)

46         #wenn Anzahl Punkte mehr als maximale Anzahl ist, wird erste Punkt
47         #aus Liste entfernt
48         if len(lidar_data) > MAX_POINTS:
49             lidar_data.pop(0)

50     #Zeichnen der Punkte
51     for point in lidar_data:
52         screen.blit(pygame.image.load('point.png'), point)

53     #Aktualisieren des Bildschirms
54     pygame.display.flip()
55     clock.tick(60)

56     #Schlussbedingung
57     if len(lidar_data) == 0:
58         break

59 
```

```

60 #Punkte anzeigen
61 for point in lidar_data:
62     #Berechnung der Position jedes Punktes auf dem Bildschirm
63     center = (point[0] + screen.get_width() / 2, screen.get_height() /
64         2 - point[1])
65
66     #Punkte mit validen Koordinaten anzeigen
67     #Farbauswahl anhand von kartesischen Koordinaten
68     if math.isfinite(point[0]) and math.isfinite(point[1]) and 1 <=
69         point[2] <= 359:
70         pygame.draw.circle(
71             screen,
72             functions.hsv2rgb(functions.get_magnitude(point[0], point
73                 [1]), 255, 255),
74             center,
75             1.5
76         )
77
78     #Aktualisierung des Bildschirms
79     pygame.display.flip()
80     #Framerate von 60 Bilder pro Sekunde
81     clock.tick(60)
82
83 #Beenden von Pygame und Socket-Verbindung
84 pygame.quit()
85 client.close()

```

Listing 1.3: Anzeige der empfangenen Daten/ lidar_visualizer.py

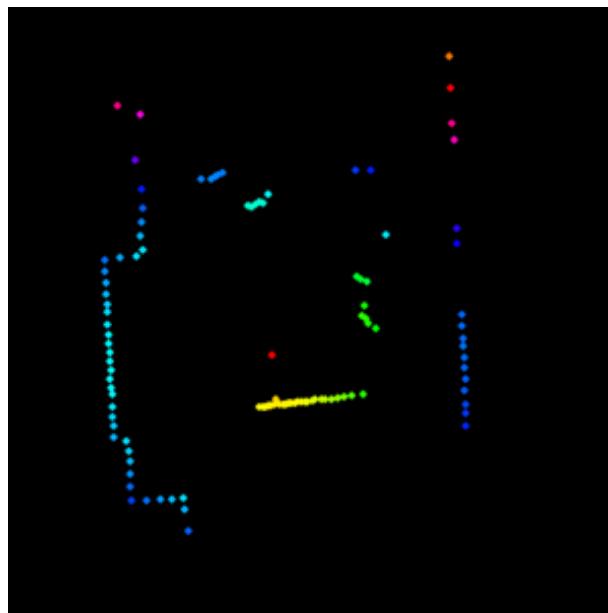


Abbildung 1.25: Anzeige der erkannten Punkten

Nach der Ausführung dieses Programms werden die durch den am Auto montierten LiDAR-Sensor erkannten Objekte beziehungsweise Hindernisse in Form von Punkten, angezeigt. Die Anzeige wird in Abbildung 1.25 veranschaulicht. Hierbei ist der Raum, in dem sich der Sensor befindet, abgebildet. Der rote Punkt repräsentiert den Sensor selbst.

Kapitel 2

Pathfinding

2.1 Allgemeines

2.1.1 Einleitung

Fährt man viel mit dem Auto oder öffentlichen Verkehrsmitteln, ist ein zuverlässiges Navigationssystem heutzutage so gut wie vorausgesetzt. Man gibt sein gewünschtes Ziel ein und schon werden die kürzesten Routen vom aktuellen Standort zum Zielort vorgeschlagen. Dieser Luxus wäre ohne *Pathfinding* undenkbar. Pathfinding, im Deutschen auch als Wegfindung bezeichnet, ist ein entscheidender Bestandteil vieler Anwendungen und Systeme, bei denen die Navigation von einem Startpunkt zu einem oder mehreren weiteren Punkten erforderlich ist. Sei es die Berechnung der schnellsten Route für den täglichen Arbeits- oder Schulweg oder die Pfadplanung für autonome Roboter in einer Fabrik. In seiner grundlegendsten Form handelt es sich um die Suche nach dem besten Pfad von einem Startpunkt zu einem Zielpunkt in einer gegebenen Menge von Punkten. Ein Beispiel für Pathfinding ist in *Abbildung 2.1* veranschaulicht. Der optimale Pfad ist grün gefärbt und verläuft von links nach rechts. [EZ:Web01]

2.1.2 Aufwand

Das Ziel von Pathfinding ist es, einen Weg zu finden, der den geringstmöglichen Aufwand erfordert. Dieser Aufwand kann in verschiedenen Kontexten unterschiedlich definiert sein, wie beispielsweise als die kumulative Entfernung oder Zeit zwischen den überquerten Knoten, oder, sowohl fiktive als auch finanzielle, Kosten. Unter fiktiven

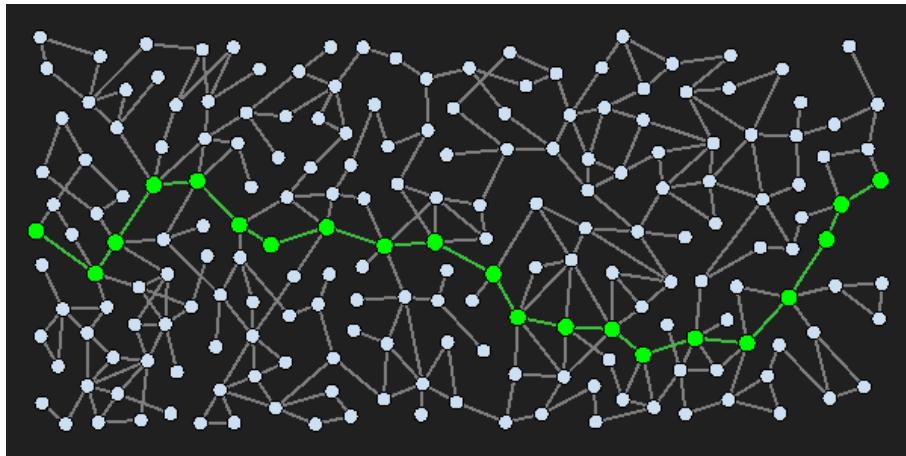


Abbildung 2.1: Beispiel für Pathfinding auf einem Graphen
[EZ:Web05]

Kosten kann man hier z.B. die Summe der Gewichtungen der Kanten, die der Pfad überquert, verstehen.

2.2 Graphentheorie

Die Graphentheorie ist eines der wichtigsten Konzepte im Bereich des Pathfinding. Ein Graph ist eine abstrakte, mathematische Struktur, die aus Knoten und Kanten besteht. In Bezug auf Pathfinding repräsentieren die Knoten die Standorte oder Punkte, zwischen denen man Wege finden möchte, und die Kanten stellen die theoretisch möglichen oder tatsächlich vorhandenen Verbindungen zwischen diesen Punkten dar. Bei Navigationssystemen für die reale Welt, wie Google Maps, Waze und anderen, repräsentieren Kanten z.B. Straßen und Knoten die Kreuzungen. [EZ:Web03, EZ:Web06, EZ:Web29]

2.2.1 Visualisierung

In den meisten Fällen werden Knoten als Kreise oder Punkte dargestellt, oft auch mit einer zusätzlichen Beschriftung oder Bezeichnung. Die Kanten werden meist als einfache Linien zwischen den Knoten dargestellt, ist der Graph jedoch ein gerichteter, sind es Pfeile anstatt Linien. Ist er gewichtet, sind die einzelnen Gewichtungen meist neben den dazugehörigen Kanten aufzufinden. [EZ:Web07]

2.2.2 Kantengewichtete Graphen

Ein Graph $G = (V, E)$ wird als kantengewichtet bezeichnet, wenn jeder Kante $e \in E$ eine Gewichtung $w(e)$ zugeordnet wird, wobei $w : E \rightarrow \mathbb{R}$ beziehungsweise $w : V \times V \rightarrow \mathbb{R}$ die Kantengewichtungsfunktion ist. Die Gewichtungen der Kanten können abhängig vom Anwendungsfall unterschiedlich interpretiert werden. Meistens stellen sie die euklidische Distanz zwischen zwei Knoten dar, oftmals aber auch die benötigte Zeit, um vom einen Knoten zum anderen zu gelangen. Letzteres ist zum Beispiel bei der Routenplanung im Straßenverkehr nützlicher, da stockender Verkehr und Staus berücksichtigt werden können. Es gibt auch Graphen, die knotengewichtet sind, diese finden aber nur selten Verwendung. Da es für knotengewichtete Graphen weitaus weniger Anwendungsfälle gibt, als für kantengewichtete, werden kantengewichtete Graphen meist nur als gewichtet bezeichnet. [EZ:Web04, EZ:Web10, EZ:Web22, EZ:Web32, EZ:Web35]

Verläuft in einem gewichteten Graphen zwischen zwei Knoten u und v keine Kante, gilt

$$w(u, v) = \infty$$

weil ein Wert von 0 eine Kante mit einer Gewichtung von 0 implizieren würde, solche Kanten aber durchaus sinnvoll sein können. Für ungewichtete Graphen gilt eine spezielle Kantengewichtungsfunktion, die folgendermaßen definiert ist:

$$w(u, v) = \begin{cases} 1 & \text{falls } u \text{ und } v \text{ verbunden sind,} \\ 0 & \text{sonst.} \end{cases}$$

Abbildung 2.2 zeigt ein einfaches Beispiel für einen Graphen. Er besteht aus fünf Knoten und fünf Kanten, die Verbindungen zwischen den Knoten darstellen. Alle Knoten sind mit einzigartigen Buchstaben beschriftet, damit man sie eindeutig identifizieren kann. Die Zahlen neben den einzelnen Kanten sind deren jeweiligen Gewichtungen, was bedeutet, dass der gezeigte Graph ein gewichteter ist. [EZ:Web42, EZ:Web43]

2.2.3 Gerichtete Graphen

Ein gerichteter Graph ist ein Graph, dessen Kanten nur in eine Richtung überquert werden dürfen. Wird eine ungerichtete Kante zwischen zwei Knoten benötigt, werden stattdessen zwei *gegenläufige*, gerichtete Kanten verwendet. Gegenläufig oder *antiparallel* nennt man zwei Kanten e_1, e_2 mit $e_1 = (a, b)$ und $e_2 = (b, a)$, wobei $a, b \in V$. Die

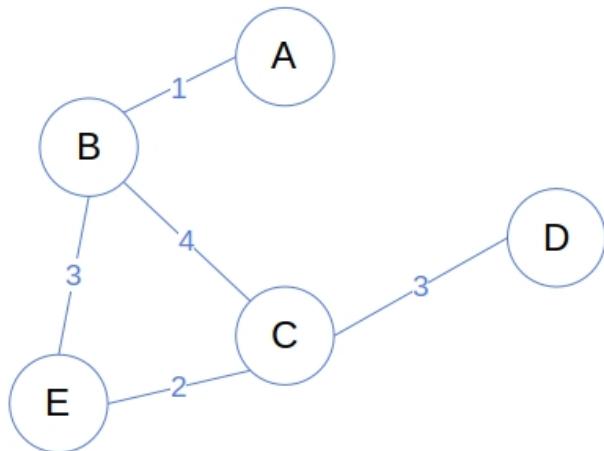


Abbildung 2.2: Ein gewichteter Graph
[EZ:Web04]

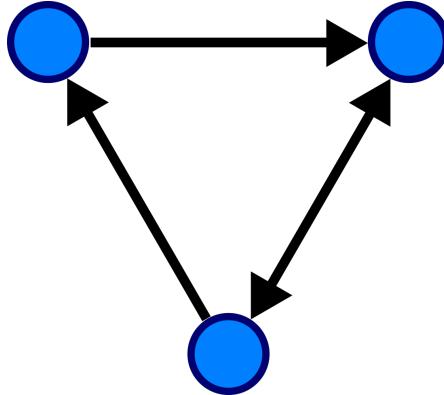


Abbildung 2.3: Ein Digraph
[EZ:Web20]

Kanten eines gerichteten Graphen nennt man gerichtete Kanten und sind geordnete Knotenpaare $(a, b) \in E$ wobei man Kanten eines ungerichteten Graphen als ungerichtet bezeichnet werden und ungeordnete Knotenpaare $\{a, b\} \in E$ sind. Ein gerichteter Graph wird häufig auch als *Digraph*¹ bezeichnet. Ein Beispiel eines Digraphen ist in Abbildung 2.3 zu sehen. [EZ:Web20]

2.2.4 Multigraphen

Multigraphen sind Graphen, in denen sowohl *Multikanten* als auch *Schlingen* vorkommen dürfen. Als Multikanten oder Mehrfachkanten bezeichnet man mehrere gleichar-

¹directed graph

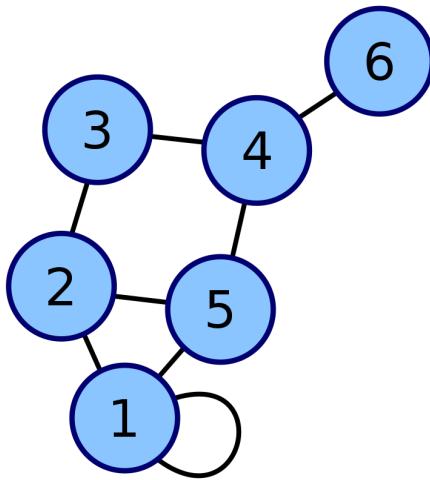


Abbildung 2.4: Ein Multigraph mit einer Schlinge
[EZ:Web27, EZ:Web28]

tige Kanten, die durch ein und dasselbe Knotenpaar verlaufen. Multikanten, die denselben Anfangs- und Endknoten haben, nennt man *parallel*. Sind zwei Multikanten, die durch dieselben zwei Knoten verlaufen, gerichtet, und zeigen diese in entgegengesetzte Richtungen, werden sie als *gegenläufig* oder *antiparallel* bezeichnet, *siehe Kapitel 2.2.3.* [EZ:Web29, EZ:Web39]

Schlingen oder *Schleifen* sind Kanten, die einen Knoten mit sich selbst verbinden. Ab hier wird in dieser Arbeit ausschließlich der Begriff *Schlinge* verwendet, um Verwechslungen mit Schleifen aus der Programmierung zu vermeiden. *Abbildung 2.4* veranschaulicht einen Multigraphen mit einer Schlinge. In den Abbildungen 2.5 und 2.6 sind gerichtete beziehungsweise ungerichtete Multigraphen visualisiert. [EZ:Web27, EZ:Web28, EZ:Web39]

2.2.5 Einfache Graphen

Im Unterschied zu Multigraphen bezeichnet man Graphen, die ungerichtet sind und weder Multikanten noch Schlingen aufweisen, als *einfach* oder *schlicht*. In *Abbildung 2.7* ist ein Beispiel eines einfachen Graphen zu sehen. [EZ:Web26]

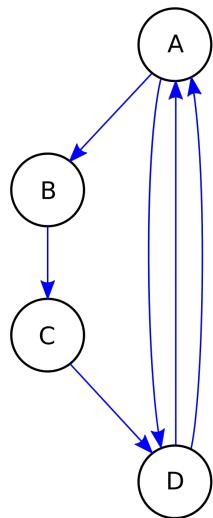


Abbildung 2.5: Ein gerichteter Graph mit Multikanten
[EZ:Web29]

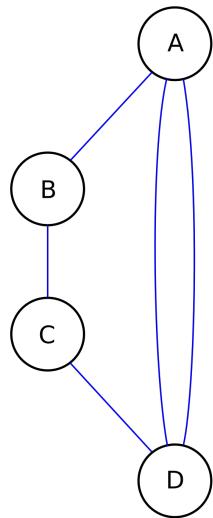


Abbildung 2.6: Ein ungerichteter Graph mit Multikanten
[EZ:Web29]

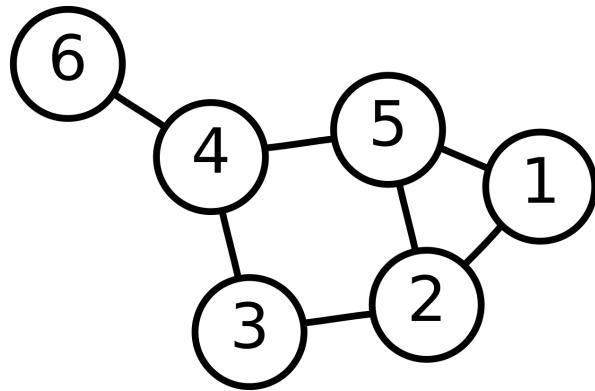


Abbildung 2.7: Ein einfacher Graph mit sechs Knoten und sieben Kanten
[EZ:Web25]

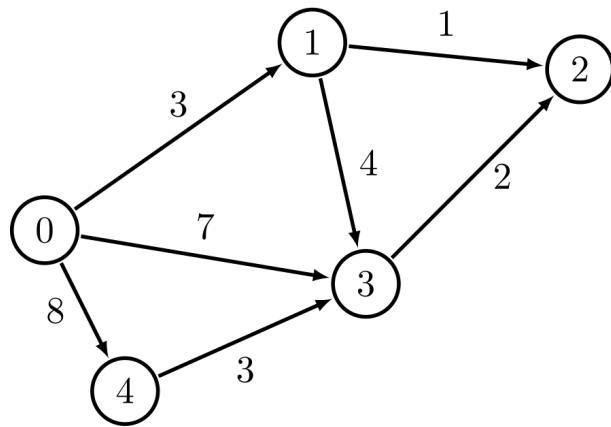


Abbildung 2.8: Ein Netzwerk
[EZ:Web10]

2.2.6 Netzwerke

Ist ein Graph sowohl gewichtet als auch gerichtet, ist er also ein gewichteter Digraph, spricht man von einem Netzwerk. Die Definition ist jedoch nicht einheitlich, zum Beispiel sind umgangssprachlich oft nur gewichtete Graphen oder gar Graphen im Allgemeinen gemeint, wenn von Netzwerken die Rede ist. In Abbildung 2.8 ist ein Netzwerk mit fünf nummerierten Knoten und sieben Kanten zu sehen. [EZ:Web09, EZ:Web25]

2.2.7 Symbolik

In den kommenden Abschnitten werden einige Symbole der Mengenlehre und Prädikatenlogik eingesetzt. Um klarzustellen, was diese bedeuten, sind die Definitionen der wichtigsten davon in Tabelle 2.1 auf Deutsch und auf Englisch aufzufinden.

| Symbol | Bedeutung | Meaning |
|------------|-----------------------------|-------------------------------|
| {...} | Menge | set |
| ... | Kardinalität (Mächtigkeit) | cardinality |
| \in | Element von | element of |
| \notin | kein Element von | not element of |
| \exists | es existiert mindestens ein | there exists at least one |
| $\exists!$ | es existiert genau ein | there exists one and only one |
| \forall | für alle | for all |

Tabelle 2.1: Wichtige Symbole der Mengenlehre
[EZ:Web23, EZ:Web24]

2.2.8 Zyklen

Ein Graph $G = (V, E)$ ist genau dann zyklisch, wenn seine Kanten mindestens einen *Zyklus* bilden. Ein Zyklus ist eine Teilmenge der Kantenmenge E eines Graphen, die einen Pfad formt, sodass der erste Knoten des Pfades dem letzten entspricht. Einen Zyklus in einem gerichteten Graphen bezeichnet man als gerichteten Zyklus und einen Zyklus in einem ungerichteten Graphen als ungerichteten Zyklus. Für einen ungerichteten Zyklus der *Länge* $k \in \mathbb{N}$ mit der Knotenfolge $(v_1, v_2, \dots, v_k, v_1)$ gilt somit $\forall i \in \{1, 2, \dots, k\} \exists e_i$ wobei $e_i = \{v_i, v_{i+1}\} \in E$ eine Kante ist, die v_i mit v_{i+1} verbindet und $v_{k+1} = v_1$. Somit sind v_k und v_{k+1} miteinander verbunden, was impliziert, dass v_k mit v_1 verbunden ist und sich somit ein Zyklus bildet. [EZ:Web11, EZ:Web12]

Obiges gilt auch für gerichtete Zyklen, jedoch ist die Länge dort oft nicht, wie bei ungerichteten Zyklen, als die Anzahl der Knoten oder Kanten, aus denen der Zyklus besteht, definiert, sondern als die Summe der Gewichtungen der im Zyklus überquerten Kanten. [EZ:Web16, EZ:Web17]

2.2.9 Kreise

Ein *Kreis* ist eine Sonderform eines Zyklus, bei dem der Pfad ein einfacher ist, also sind nicht nur die überquerten Kanten des Pfades einzigartig, sondern auch die überquerten Knoten. Somit müssen sich die Knoten v_1, v_2, \dots, v_k alle voneinander unterscheiden. Abbildung 2.9 zeigt ein simples Beispiel eines zyklischen Graphen, dessen Zyklus ein Kreis ist. [EZ:Web13]

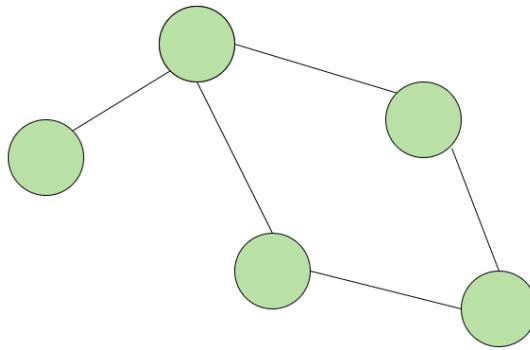


Abbildung 2.9: Ein zyklischer Graph mit einem Kreis
[EZ:Web12]

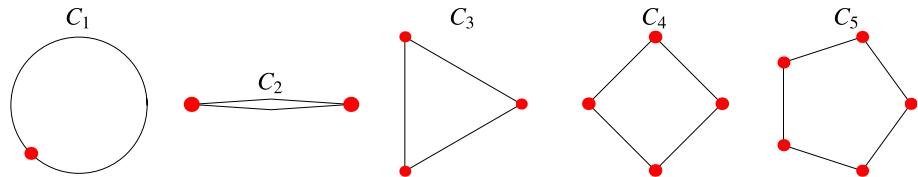


Abbildung 2.10: Die Kreisgraphen C_1 , C_2 , C_3 , C_4 und C_5
[EZ:Web14]

2.2.10 Kreisgraphen

Enthält ein Graph genau einen Zyklus, welcher gleichzeitig ein Kreis ist, und besteht dieser aus der gesamten Knotenmenge V des Graphen, bezeichnet man den Graphen als *Kreisgraph*. Für einen Kreisgraphen gilt immer

$$|V| = |E|$$

was bedeutet, dass die Anzahl der Knoten mit der der Kanten übereinstimmt. Die ersten fünf Kreisgraphen C_1 , C_2 , C_3 , C_4 und C_5 sind in *Abbildung 2.10* veranschaulicht. Der Knoten des Kreisgraphen C_1 ist durch eine Schlinge mit sich selbst verbunden. [EZ:Web14, EZ:Web15]

2.2.11 Vollständige Graphen

Ein vollständiger Graph ist ein einfacher Graph, in dem jeder Knoten eine Kante zu allen anderen im Graphen enthaltenen Knoten hat. Anders ausgedrückt: Jedes Paar von unterschiedlichen Knoten ist durch eine Kante verbunden. Für einen *vollständigen*

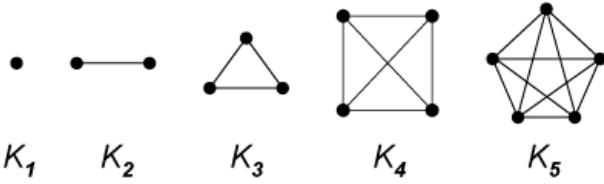


Abbildung 2.11: Die vollständigen Graphen K_1 , K_2 , K_3 , K_4 und K_5
[EZ:Web33]

Digraphen gilt ähnlich: Jedes Paar von unterschiedlichen Knoten ist durch *ein Paar von unterschiedlichen Kanten* verbunden, also mit einer Kante je Richtung. In Abbildung 2.11 sind die vollständigen Graphen K_1 , K_2 , K_3 , K_4 und K_5 zu sehen. [EZ:Web33, EZ:Web34]

2.2.12 Bipartite Graphen

Ein Graph ist *bipartit*, wenn eine der beiden folgenden Aussagen gilt:

- Der Graph ist azyklisch
- Für jeden Zyklus mit der Knotenfolge $(v_1, v_2, \dots, v_n, v_1) \in V$ gilt $n \equiv 0 \pmod{2}$. Es gilt also für keinen Zyklus $n \equiv 1 \pmod{2}$.

Ist ein Graph bipartit, kann man seine Knoten in zwei disjunkte Teilmengen aufteilen, sodass zwischen den Knoten innerhalb beider Teilmengen keine Kanten verlaufen. Damit das Ganze etwas greifbarer ist, ist die genannte Bedingung in Abbildung 2.12 mithilfe eines Beispiels für solch einen Graphen verdeutlicht. Der gezeigte Graph ist nicht *vollständig bipartit*, da nicht jeder Knoten der Teilmenge U eine Kante zu jedem Knoten der Teilmenge V hat. Ein Beispiel für einen Graphen, der diese Bedingung erfüllt und somit vollständig bipartit ist, ist in Abbildung 2.13 veranschaulicht. [EZ:Web12, EZ:Web19]

2.2.13 Grids

Neben Graphen werden für Pathfinding häufig auch Grids verwendet, da diese für einige Anwendungsfälle besser geeignet sind, da es keine vordefinierten Kanten gibt. Ein Grid kann als Sonderfall eines Graphen betrachtet werden, bei dem die Knoten in gleichmäßigen Abständen platziert sind und jeder Knoten Kanten zu allen Knoten hat, die ihn umgeben, sofern diese nicht von Hindernissen oder Ähnlichem blockiert

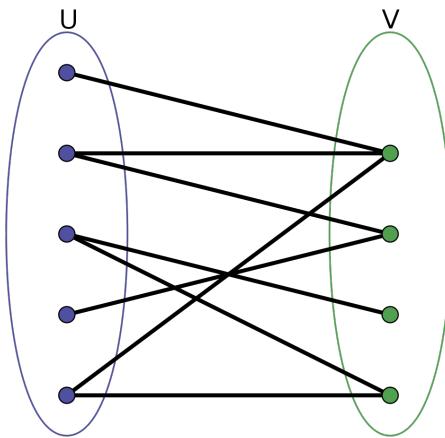


Abbildung 2.12: Ein einfacher, nicht vollständig bipartiter Graph mit Partitionsklassen U und V
[EZ:Web19]

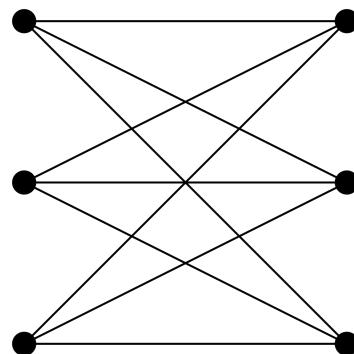


Abbildung 2.13: Ein einfacher, vollständig bipartiter Graph
[EZ:Web19]

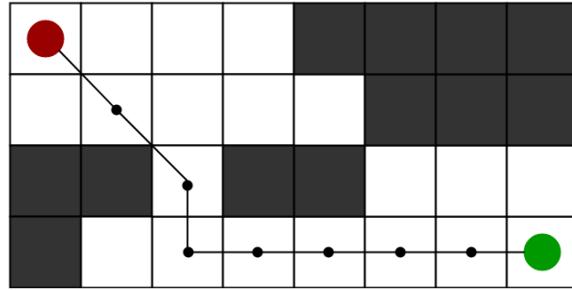


Abbildung 2.14: Beispiel für Pathfinding auf einem Grid
[EZ:Web02]

sind. Ein Beispiel für Pathfinding auf einem Grid ist in *Abbildung 2.14* zu sehen. Hier ist der rote Kreis der Startpunkt und der grüne der Zielpunkt. Die dunklen Zellen stellen Hindernisse dar, die der Pfad vermeiden muss. In diesem spezifischen Fall sind diagonale Schritte erlaubt, weshalb der kürzeste Weg zwei Diagonalen beinhaltet. Stellt man das Grid als Graph dar, existieren entweder keine Kanten zu den blockierten Knoten oder die Hindernisse werden erst gar nicht als Knoten dargestellt, sondern schlicht und ergreifend verworfen.

2.2.14 Darstellung

Adjazenzliste

Mithilfe einer *Adjazenzliste* kann ein Graph mitsamt dessen Knoten und Kanten dargestellt werden. Sie ist eine einfache Auflistung aller Nachbarknoten für jeden Knoten, wobei mit Nachbarknoten alle Knoten gemeint sind, zu denen ein Knoten eine Kante hat. Will man einen gewichteten Graphen als Adjazenzliste darstellen, speichert man gemeinsam mit jeder Kante ihre zugehörige Gewichtung ab. Adjazenzlisten eignen sich gut für gerichtete Graphen, da man mit ihnen nur die von Knoten *ausgehenden* Kanten beschreibt, nicht die eingehenden. Für ungerichtete Graphen ergeben sich somit zwei Möglichkeiten sie als Adjazenzliste abzuspeichern:

1. Man speichert jede ungerichtete Kante $\{v_i, v_j\}$ als zwei gegenläufige gerichtete Kanten ab: bei Knoten v_i als (v_i, v_j) und bei Knoten v_j als (v_j, v_i) . Dadurch ist es egal, auf welcher Seite man auf die Existenz einer Kante prüft. Ein Vorteil dieser Methode ist, dass man garantiert immer nur einen Kanten-Check durchführen muss und man deshalb auch nicht klarstellen muss, ob es sich um einen gerichteten oder einen ungerichteten Graphen handelt, solange man den Graphen nicht mehr verändert. Man kann bei Anwendung dieser Methode eine Art

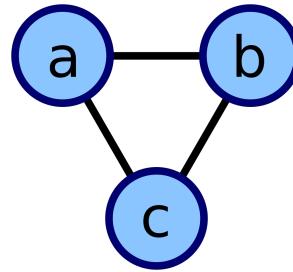


Abbildung 2.15: C_3 bzw. K_3 , der kleinste Dreiecksgraph
[EZ:Web31]

Mischung aus einem gerichteten und einem ungerichteten Graphen darstellen, da gerichtete Kanten nur bei ihrem Startknoten abgespeichert werden müssen. Der klare Nachteil ist der doppelte Speicherverbrauch für ungerichtete Kanten.

2. Man speichert jede ungerichtete Kante $\{v_i, v_j\}$ nur einmal ab, entweder bei v_i oder bei v_j , wodurch weniger Speicherplatz beansprucht wird. Prüft man jedoch von Knoten v_i ausgehend, ob zum Knoten v_j eine Kante (v_i, v_j) existiert und stellt fest, dass das nicht der Fall ist, so muss auch auf die Existenz der gegenläufigen Kante (v_j, v_i) geprüft werden. Geht man davon aus, dass man in durchschnittlich 50% der Fälle vom Startknoten und in 50% der Fälle vom Endknoten der zu überprüfenden Kante ausgehend prüft, so müssen in der Hälfte aller Fälle zwei Checks durchgeführt werden, was bedeutet, dass durchschnittlich $0.5 \cdot 1 + 0.5 \cdot 2 = 1.5$ Checks durchgeführt werden müssen, was ein klarer Nachteil ist. Damit die Existenz der gegenläufigen Kante (v_j, v_i) auch wirklich geprüft wird, falls (v_i, v_j) nicht existiert, muss klargestellt werden, dass es sich um einen ungerichteten Graphen handelt. Durch diesen Nachteil ergibt sich auch schon der nächste: Es gibt bei dieser Methode keine Möglichkeit, gerichtete Kanten darzustellen, sie kann also nur bei rein ungerichteten Graphen eingesetzt werden.

Tabelle 2.2 stellt die Adjazenzliste für den Graphen, der in Abbildung 2.15 zu sehen ist, dar. Dieser ist ungerichtet und zur Darstellung wird Methode 1 angewendet. [EZ:Web31, EZ:Web40]

| Knoten | adjazent mit |
|--------|--------------|
| a | b, c |
| b | a, c |
| c | a, b |

Tabelle 2.2: Adjazenzliste

| Case | Adjazenzliste | Adjazenzmatrix |
|---------|--------------------------|----------------------|
| Average | $\mathcal{O}(V + E)$ | $\mathcal{O}(V ^2)$ |
| Worst | $\mathcal{O}(V ^2)$ | $\mathcal{O}(V ^2)$ |

Tabelle 2.3: Die Platzkomplexitäten der Adjazenzliste und -matrix
[EZ:Web40]

Adjazenzmatrix

Adjazenzmatrizen bieten eine weitere Möglichkeit, einen Graphen $G = (V, E)$ darzustellen. Eine Adjazenzmatrix A ist eine quadratische $|V| \times |V|$ -Matrix. Sie speichert in jedem Element $A_{i,j}$ die Gewichtung der Kante zwischen den Knoten v_i und v_j . Wird sie an einem ungewichteten Graphen angewandt, wird eine 1 gespeichert, falls es eine Kante zwischen den beiden Knoten gibt. Verläuft keine Kante durch v_i und v_j , speichert sie den Wert 0 oder ∞ . Für gewichtete Graphen eignet sich ∞ besser, da 0 in manchen Fällen eine sinnvolle Gewichtung für Kanten sein kann. Adjazenzmatrizen haben den Vorteil, dass sie übersichtlich und, wie Adjazenzlisten, gut für gerichtete Graphen geeignet sind. Ihr großer Nachteil ist jedoch, dass es viele redundante Werte gibt, da Kanten, die nicht existieren, nicht abgespeichert werden müssten. Adjazenzmatrizen haben im Durchschnittsfall eine quadratische Platzkomplexität, wie in *Tabelle 2.3* angeführt. Aus diesem Grund sind Adjazenzmatrizen nur für eher kleine Graphen geeignet. In *Tabelle 2.4* ist die Adjazenzmatrix für den Graphen aus *Abbildung 2.15* zu sehen. [EZ:Web36, EZ:Web37, EZ:Web41, EZ:Web42, EZ:Web43]

| | a | b | c |
|---|---|---|---|
| a | 0 | 1 | 1 |
| b | 1 | 0 | 1 |
| c | 1 | 1 | 0 |

Tabelle 2.4: Adjazenzmatrix des Graphen aus *Abbildung 2.15*

2.2.15 Implementierung

Für fertige Implementierungen gibt es zahlreiche Libraries, wie z.B. *NetworkX* für Python oder *JGraphT* für Java. Um jedoch zu verstehen, wie diese intern funktionieren, wird das Konzept eines Graphen in dieser Arbeit von Grund auf neu implementiert: In *Listing 2.1* wird das generische Interface `Graph<T>` definiert, welches dazu dient, mehrere unterschiedliche Arten von Graphen darstellen zu können, zum Beispiel auch unendlich große. Das Interface schreibt genau eine Methode vor, die implementiert werden muss: `Map<T, Double> getNeighbors(T vertex)`. Mithilfe von dieser können alle Nachbarn eines gegebenen Knoten inklusive der Gewichtungen der Kanten zu diesen berechnet werden. Dadurch können auch unendlich große Graphen dargestellt werden, da die tatsächliche Struktur des Graphen nicht abgespeichert werden muss, sondern durch seine Adjazenzen impliziert werden kann. Zusätzlich sind `boolean hasEdge(T source, T destination)`, `int getDegree(T vertex)`, `double getEdgeWeight(T source, T destination)` und `double sumEdgeWeights(List<T> path)` defaultmäßig so implementiert, dass `getNeighbors` in den Methoden `hasEdge`, `getDegree` und `getEdgeWeight` zum Einsatz kommt und `getEdgeWeight` wiederum in `sumEdgeWeights` verwendet wird. Hierbei gibt `hasEdge` einen `boolean` zurück, der angibt, ob zwischen den beiden übergebenen Knoten in der angegebenen Reihenfolge eine Kante verläuft. Mithilfe von `int getDegree(T vertex)` kann man die Anzahl an Nachbarn des übergebenen Knotens erhalten. `getEdgeWeight` ist dafür zuständig, die Gewichtung der Kante zurückzugeben, die zwischen den beiden übergebenen Knoten verläuft, bzw. ∞ , falls die Knoten nicht (in der übergebenen Reihenfolge) miteinander verbunden sind. Die `sumEdgeWeights`-Methode ist dazu da, die Gesamtgewichtung eines Pfades zu berechnen, was für manche Pathfinding-Algorithmen entscheidend ist und eine wichtige Metrik für das Benchmarking darstellt.

```

1  public interface Graph<T> {
2
3      /**
4       * @param vertex the vertex to get the neighbors of
5       * @return a Map of the neighbors of the vertex and
6       * the weights of the edges between them
7       */
8      Map<T, Double> getNeighbors(T vertex);
9
10     /**
11      * @param source      vertex u of the edge (u, v) to be checked for
12      * @param destination vertex v of the edge (u, v) to be checked for
13      * @return the weight of the edge between the two vertices
14      */
15     default boolean hasEdge(T source, T destination) {
16         return getNeighbors(source).containsKey(destination);
17     }
18
19     /**
20      * @param vertex the vertex to get the degree of
21      * @return the degree of the vertex
22      */
23     default int getDegree(T vertex) {
24         return getNeighbors(vertex).size();
25     }
26
27     /**
28      * @param path the path to calculate the total weight of
29      * @return the accumulated weight of the path
30      */
31     default double sumEdgeWeights(List<T> path) {
32         double sum = 0;
33
34         for (int i = 0; i < path.size() - 1; i++) {
35             sum += getEdgeWeight(path.get(i), path.get(i + 1));
36         }
37
38         return sum;
39     }
40
41     default double getEdgeWeight(T source, T destination) {
42         return getNeighbors(source).getOrDefault(
43             destination,
44             Double.POSITIVE_INFINITY
45         );
46     }
47 }

```

Listing 2.1: Graph.java

Das Interface `ModifiableGraph<T>` aus *Listing 2.2* ist eine Erweiterung von `Graph<T>`, das einige zusätzliche Methoden vorschreibt. Mit diesen können Graph-Instanzen von Klassen, die dieses Interface implementieren, modifiziert werden und es können viele wichtige Informationen über sie ausgelesen werden. Beispielsweise kann mit `double calculateAverageDegree()` die durchschnittliche Nachbaranzahl pro Knoten berechnet werden.

```
1 public interface ModifiableGraph<T> extends Graph<T> {
2
3     /**
4      * Adds an unweighted edge between two vertices.
5      *
6      * @param source      the source vertex of the edge
7      * @param destination the destination vertex of the edge
8      */
9     void addEdge(T source, T destination);
10
11    /**
12     * Adds a weighted edge between two vertices with a weight.
13     *
14     * @param source      the source vertex of the edge
15     * @param destination the destination vertex of the edge
16     * @param weight      the weight of the edge
17     */
18    void addEdge(T source, T destination, double weight);
19
20    /**
21     * Adds a vertex to the graph.
22     *
23     * @param vertex the value of the vertex to be added
24     */
25    void addVertex(T vertex);
26
27    /**
28     * Removes an edge between two vertices.
29     *
30     * @param source      the source vertex of the edge to be removed
31     * @param destination the destination vertex of the edge to be removed
32     */
33    void removeEdge(T source, T destination);
34
35    /**
36     * Removes a vertex from the graph.
37     *
38     * @param vertex the value of the vertex to be removed
39     */
40    void removeVertex(T vertex);
41
42    /**
43     * @return the number of edges in the graph
44     */
```

```

45     int getEdgeCount();
46
47     /**
48      * @return the number of vertices in the graph
49      */
50     int getVertexCount();
51
52     /**
53      * @param vertex the vertex to be checked for
54      * @return the weight of the edge between the two vertices
55      */
56     boolean hasVertex(T vertex);
57
58     /**
59      * @return the average degree of the vertices in the graph
60      */
61     double calculateAverageDegree();
62
63     /**
64      * @return whether the graph is directed or not
65      */
66     boolean isDirected();
67
68     /**
69      * @return a map of all the vertices and their adjacencies
70      */
71     Map<T, Map<T, Double>> getAdjacencies();
72
73     /**
74      * @return a Set of all the vertices in the graph
75      */
76     Set<T> getVertices();
77
78     /**
79      * Adds a Collection of vertices to the graph.
80      */
81     default void addVertices(Collection<T> vertices) {
82         vertices.forEach(this::addVertex);
83     }
84 }

```

Listing 2.2: ModifiableGraph.java

Eine simple, konkrete Graph-Klasse könnte in Java in etwa wie in *Listing 2.3* aussehen. Sie implementiert das eben erwähnte `Graph<T>`-Interface und ist generisch, da Graphen eine Unzahl an Anwendungsfällen haben. Zum Beispiel können neben Straßennetzen und vielem mehr auch Freundschaften in einem sozialen Netzwerk dargestellt und analysiert werden. In diesem Fall könnte man für den Typparameter `T` beispielsweise eine `Person` verwenden, oder `String` für den Namen. Sind die Namen der

| Knoten | Adjazzenzen |
|--------|--------------------|
| A | B=1 |
| B | C=3 |
| C | E=7 |
| D | A=8, C=1 |
| E | A=5, B=3, C=2, D=5 |

Tabelle 2.5: Gewichtete Adjazenzliste des generierten Graphen aus *Listing 2.5*

Personen nicht eindeutig, können Klassen wie `integer`, `Long` oder `UUID` als Identifikator verwendet werden. [EZ:Web21, EZ:Web45, EZ:Web46]

In der Variable `Map<T, Map<T, Double>> adjacencies` werden sowohl die Knoten als auch die von ihnen ausgehenden Kanten gespeichert, indem jedem Knotenwert in einer Map eine weitere Map zugeordnet wird, die jedem Nachbarn des Knotens, zu dem die (innere) Map gehört, die Gewichtungen der jeweiligen Kanten als `Double` zuordnet. Der soeben beschriebene Ansatz, einen Graphen zu speichern, ist eine Form der in *Kapitel 2.2.14* beschriebenen Adjazenzliste, mit dem lediglichen Zusatz der Kanten-gewichtungen. Für eine Implementierung eines ungewichteten Graphen wäre eine `Map<T, List<T>>` völlig ausreichend, jedoch ist der Zweck dieser Klasse, möglichst viele Arten von Graphen darstellen zu können, daher auch der Name `FlexibleGraph<T>`.

Die Klasse ist mit einigen Lombok-Annotations ausgestattet, wie z.B. `@ToString` über der Klassendefinition für die automatische Generierung einer `toString`-Methode, die einen `String` zurückgibt, der den Namen der Klasse sowie sämtliche Instanzvariablennamen und -werte von dieser enthält. Außerdem ist sie mit `@Getter` und `@Setter` annotiert, damit für die `final` Variablen `boolean directed` und `Map<T, Map<T, Double>> adjacencies` automatisch Getter mit den Namen `isDirected` und `getAdjacencies` erstellt werden und für die `ToDoubleBiFunction<T, T> defaultWeightFunction` sowohl Getter als auch Setter erstellt werden. Bis auf den einzigen Unterschied, dass `ToDoubleBiFunction::<T, U>applyAsDouble` einen primitiven `double`, `BiFunction::<T, U, Double>apply` hingegen den Wrapper `Double` zurückgibt, sind die beiden Interfaces identisch. So wird dem Computer mit dem Einsatz von `ToDoubleBiFunction` ein wenig Arbeit erspart, da der Rechenaufwand von Boxing und Unboxing wegfällt.

Dem Konstruktor des Graphen wird ein `boolean directed` übergeben, der angibt, ob der Graph gerichtet ist, oder nicht. Ist er ungerichtet, wird für jede hinzugefügte Kante eine gegenläufige Kante, also mit `source` und `destination` vertauscht, abgespeichert. Es kommt also die erste der beiden in *Kapitel 2.2.14* genannten Möglichkeiten zur Darstellung für ungerichtete Graphen als Adjazenzliste zum Einsatz. Graphen wie diese können als eine Art gewichtete Adjazenzliste dargestellt werden. Ein Beispiel dafür ist in *Tabelle 2.5* zu sehen.

Wird der Parameter `double weight` der Methode `addEdge` weggelassen, wird für hinzugefügte Kanten die `defaultWeightFunction` evaluiertl. Diese gibt defaultmäßig immer 1 zurück, kann allerdings beliebig angepasst werden. Da die Klasse `ModifiableGraph<T>` implementiert, sind auch einige Hilfsmethoden vorhanden, um den Graphen zu modifizieren zu können. In `addVertex` wird `Map::computeIfAbsent` verwendet, weil

1. mit `Map::put` die von dem Knoten ausgehenden Kanten mit einer `new HashMap<>()` überschrieben werden würden,
2. mit `Map::putIfAbsent` unnötig eine `new HashMap<>()` erzeugt werden würde,

wenn bereits ein Knoten mit dem übergebenen Wert existiert. Da `Map::<K, V>put` und `Map::<K, V>putIfAbsent` als zweiten Parameter ein `v` erwarten, welches dem Value entspricht, `Map::<K, V>computeIfAbsent` aber eine `Function<K, V>`, die den Value zurückgibt, wird die `new HashMap<>()` mit `Map::computeIfAbsent` nur dann erzeugt, wenn der einzufügende Key noch nicht in der Map vorhanden ist. Aus diesem Grund sind die Methoden `addEdge`, `addVertex` und `addVertices` *idempotent*, was bedeutet, dass sich der Zustand des Graphen, wenn eine der genannten Methoden bereits einmal aufgerufen wurde, nach erneutem Aufrufen mit denselben Parametern nicht mehr verändert. Somit ermöglicht diese Implementierung weder mehrere Knoten mit demselben Wert noch Multikanten, eine Schlinge pro Knoten ist jedoch möglich. Die restlichen Methoden sind ebenfalls idempotent, jedoch ist diese Eigenschaft bei `get-` und `remove-`-Methoden weniger besonders. [EZ:Web30]

```
1 @RequiredArgsConstructor
2 @Getter
3 @Setter
4 @ToString
5 public class FlexibleGraph<T> implements ModifiableGraph<T> {
6
7     private final boolean directed;
8
9     private final Map<T, Map<T, Double>> adjacencies = new HashMap<>();
10
11    @ToString.Exclude
12    private ToDoubleBiFunction<T, T> defaultWeightFunction = (_, _) -> 1;
13
14    /**
15     * Undirected graph constructor.
16     */
17    public FlexibleGraph() {
18        this(false);
19    }
20
21    @Override
22    public void addEdge(T source, T destination) {
23        addEdge(
24            source,
25            destination,
26            defaultWeightFunction.applyAsDouble(source, destination)
27        );
28    }
29
30    @Override
31    public void addEdge(T source, T destination, double weight) {
32        addVertex(source);
33        addVertex(destination);
34        adjacencies.get(source).put(destination, weight);
35
36        if (!directed) {
37            adjacencies.get(destination).put(source, weight);
38        }
39    }
40
41    @Override
42    public void addVertex(T vertex) {
43        adjacencies.computeIfAbsent(vertex, _ -> new HashMap<>());
44    }
45
46    @Override
47    public void removeEdge(T source, T destination) {
48        adjacencies.get(source).remove(destination);
49
50        if (!directed) {
51            adjacencies.get(destination).remove(source);
```

```

52     }
53 }
54
55 @Override
56 public void removeVertex(T vertex) {
57     adjacencies.remove(vertex);
58 }
59
60 @Override
61 public int getEdgeCount() {
62     int count = adjacencies.values()
63         .stream()
64         .mapToInt(Map::size)
65         .sum();
66
67     return (directed) ? count : count / 2;
68 }
69
70 @Override
71 public int getVertexCount() {
72     return adjacencies.size();
73 }
74
75 @Override
76 public boolean hasVertex(T vertex) {
77     return adjacencies.containsKey(vertex);
78 }
79
80 @Override
81 public double calculateAverageDegree() {
82     return adjacencies.values()
83         .stream()
84         .mapToInt(Map::size)
85         .average()
86         .orElse(0);
87 }
88
89 @Override
90 public Set<T> getVertices() {
91     return adjacencies.keySet();
92 }
93
94 @Override
95 public Map<T, Double> getNeighbors(T vertex) {
96     return adjacencies.getOrDefault(vertex, Collections.emptyMap());
97 }
98
99 }

```

Listing 2.3: Implementierung einer flexiblen Graph-Klasse in Java

2.2.16 Erzeugung

Damit solche Graphen zufällig generiert werden können, was vor allem für das Benchmarking wichtig wird, gibt es die Klasse `Modifiable`, welche in *Listing 2.4* definiert ist. Die Bezeichnung dieser Klasse kann auf zwei unterschiedliche Arten interpretiert werden:

1. **ModifiableGraph Randomizer**: Die eigentlich beabsichtigte Bedeutung, da es die Aufgabe dieser Klasse ist, `ModifiableGraph`-Objekte zu randomisieren.
2. **Modifiable GraphRandomizer**: Die sich durch Zufall ergebene mögliche alternative Interpretation, die dennoch richtig ist, da die Klasse das Setzen vieler verschiedener Parameter ermöglicht.

Das Builder-Pattern wird hier angewendet, um ein übersichtlicheres Erzeugen von neuen `ModifiableGraphRandomizer`-Instanzen zu ermöglichen. Zur automatischen Generierung des Builders für diese Klasse wird die Lombok-Annotation `@Builder` verwendet, die eine `builder()`-Methode zur Verfügung stellt, die den Builder zurückgibt. Die Annotation `@Builder.Default` sorgt dafür, dass die den Variablen zugewiesenen Defaultwerte im Builder übernommen werden und dadurch nicht immer alle Instanzvariablen selbst gesetzt werden müssen. Zudem ist die Klasse mit `@Getter` und `@Setter` annotiert, wodurch automatisch für alle Instanzvariablen Getter und für alle `non-final` Instanzvariablen Setter erzeugt werden. [EZ:Web44]

Die `toDoubleBiFunction<T, T> weightFunction` dient dazu, die Gewichtung einer übergebenen Kante zu berechnen, bevor sie dieser zugewiesen wird. Defaultmäßig wird immer die Gewichtung 1 zugeteilt. In der Methode `randomizeUndirectedEdges` fängt die Laufvariable `int j` der inneren `for`-Schleife nicht bei 0, sondern bei `i + 1` an, weil der Graph ungerichtet ist und ansonsten jede Kante *zwei* Chancen bekäme, zu entstehen, anstatt nur einer. Somit wäre die Wahrscheinlichkeit, dass zwei Knoten miteinander verbunden werden, nicht mehr `edgeProbability`, sondern $1 - (1 - p)^2$, wobei $p = \text{edgeProbability}$. Man könnte annehmen, dass sich die Wahrscheinlichkeit verdoppelt, oder allgemeiner, dass die Wahrscheinlichkeit p_n , dass zwei Knoten nach n Versuchen miteinander verbunden wurden, np beträgt. Dies kann jedoch durch eine einfache *Reductio ad absurdum* widerlegt werden: Angenommen $p = 1$ und $n = 2$, so wäre $p_2 = 2p = 2$, also gäbe es eine 200%ige Chance, dass sich zwei Knoten miteinander verbinden, was unmöglich ist. Die tatsächliche Formel für p_n kann wie folgt hergeleitet werden: Ist p die Wahrscheinlichkeit, dass bei einem Versuch eine Kante zwischen zwei Knoten entsteht, so ist $1 - p$ die Gegenwahrscheinlichkeit, also die Wahrscheinlichkeit, dass keine Kante erzeugt wird. Somit ist $(1 - p)^n$ die Wahrscheinlichkeit, dass bei keinem von n Versuchen eine Kante erstellt wird. Um die Wahrscheinlichkeit zu errechnen, dass die zwei Knoten bei mindestens einem der

n Versuche miteinander verbunden werden, muss man ein weiteres Mal die Gegenwahrscheinlichkeit berechnen, indem man $(1 - p)^n$ von 1 subtrahiert. Man erhält die allgemeine Formel $p_n = 1 - (1 - p)^n$. Setzt man $n = 2$ in diese Formel ein, findet man heraus, dass $p_2 = 1 - (1 - p)^2$ gilt. Würde man also z.B. $p = 0.5$ mit dem fehlerhaften Code verwenden, bei dem zu Beginn der inneren `for`-Schleife `int j = 0` ausgeführt wird, anstatt `int j = i + 1`, so wäre die tatsächliche Kantenwahrscheinlichkeit $p_2 = 1 - (1 - 0.5)^2 = 1 - 0.5^2 = 1 - 0.25 = 0.75 = 75\%$, anstatt den eigentlich gewollten 50%. [EZ:Web38]

Im Gegensatz zu `randomizeUndirectedEdges` kommen bei `randomizeDirectedEdges` `for-each`-Schleifen zum Einsatz, da der Index für das zufällige Erzeugen von gerichteten Kanten nicht benötigt wird, weil die Reihenfolge der Knotenpaare bei gerichteten Graphen einen Unterschied macht, weshalb keine Kante mehr als eine Chance bekommt, zu entstehen. Damit Schlingen vermieden werden, muss nur überprüft werden, ob `source != destination`, ansonsten wird keine Kante erzeugt. Aus demselben Grund beginnt `j` in `randomizeUndirectedEdges` bei `i + 1`, anstatt bei `i`.

```

1  @Builder
2  @Getter
3  @Setter
4  public class ModifiableGraphRandomizer<T> {
5
6      private static final Random RANDOM = new Random();
7
8      /**
9       * The vertices the randomized graph will contain
10      */
11     @Builder.Default
12     private List<T> vertices = Collections.emptyList();
13
14     /**
15      * The probability of an edge being created between two vertices
16      */
17     @Builder.Default
18     private double edgeProbability = 0.5;
19
20     /**
21      * The function used to calculate the weight of a given edge
22      */
23     @Builder.Default
24     private ToDoubleBiFunction<T, T> weightFunction = (_, _) -> 1;
25
26     /**
27      * Generates random edges between the vertices of a directed graph
28      *
29      * @return a new directed graph with randomized edges
30      */
31     public ModifiableGraph<T> randomizeDirectedEdges() {
32         var graph = new FlexibleGraph<T>(true);

```

```
33     for (T source : vertices) {
34         for (T destination : vertices) {
35             if (source != destination && RANDOM.nextDouble() <
36                 edgeProbability) {
37                 graph.addEdge(
38                     source,
39                     destination,
40                     calculateWeight(source, destination)
41                 );
42             }
43         }
44     }
45
46     return graph;
47 }
48
49 /**
50 * Generates random edges between the vertices of an undirected graph
51 *
52 * @return a new undirected graph with randomized edges
53 */
54 public ModifiableGraph<T> randomizeUndirectedEdges() {
55     var graph = new FlexibleGraph<T>(false);
56
57     for (int i = 0; i < vertices.size(); i++) {
58         for (int j = i + 1; j < vertices.size(); j++) {
59             if (RANDOM.nextDouble() < edgeProbability) {
60                 T source = vertices.get(i);
61                 T destination = vertices.get(j);
62
63                 graph.addEdge(
64                     source,
65                     destination,
66                     calculateWeight(source, destination)
67                 );
68             }
69         }
70     }
71
72     return graph;
73 }
74
75     private double calculateWeight(T source, T destination) {
76         return weightFunction.applyAsDouble(source, destination);
77     }
78 }
79 }
```

Listing 2.4: ModifiableGraphRandomizer

Ein Anwendungsbeispiel der `ModifiableGraphRandomizer`-Klasse ist in *Listing 2.5* demonstriert. Der in *Tabelle 2.5* als gewichtete Adjazenzliste dargestellte Graph wurde mit diesem Codeausschnitt generiert.

```

1 var random = new Random();
2
3 var graphRandomizer = ModifiableGraphRandomizer.<Character>builder()
4     .vertices(List.of('A', 'B', 'C', 'D', 'E'))
5     .weightFunction((_, _) -> 1 + random.nextInt(10))
6     .build();
7
8 var graph = graphRandomizer.randomizeDirectedEdges();
```

Listing 2.5: Beispielanwendung der Klasse aus *Listing 2.4*

2.3 Algorithmen

2.3.1 Bewertung

Pathfinding-Algorithmen können anhand verschiedenster Kriterien bewertet werden. Ein Algorithmus, der in einem Szenario gut funktioniert, muss nicht unbedingt in einem anderen Kontext die beste Wahl sein. Die Auswahl des für den Anwendungsfall richtigen Pathfinding-Algorithmus hängt vor allem von den folgenden drei Faktoren ab:

1. **Laufzeit:** Die Zeitdauer, bis ein bzw. der kürzeste Pfad gefunden wurde.
2. **Besuchte Knoten:** Die Anzahl an Knoten, die besucht werden mussten, damit der Pfad gefunden werden konnte.
3. **Speicherbedarf:** Der Speicher, der im Prozess des Pathfinding eingenommen wird.
4. **Skalierbarkeit:** Die Fähigkeit eines Algorithmus, effizient mit großen und komplexen Graphen umzugehen.

Diese sollte man immer im Hinterkopf behalten, wenn man einen Algorithmus entwickelt oder anwendet. Nicht nur bei Pathfinding. Insbesondere die Laufzeit und die Anzahl besuchter Knoten werden im Laufe dieser Arbeit, vor allem beim Benchmarking, noch eine wichtige Rolle spielen.

2.3.2 Hilfsklassen

EndCondition

Die Klasse `EndCondition<T>` aus *Listing 2.6* dient als Endbedingung für Pathfinding-Algorithmen. Die Verwendung einer Endbedingung ist oft vorteilhaft gegenüber der Angabe eines spezifischen Endknotens, da dieser im Vorhinein oft noch nicht bekannt ist. Will man zum Beispiel den kürzesten Pfad zur nächsten Tankstelle finden, so kann man als Endbedingung etwa `vertex.getPointOfInterest() == PointOfInterest.GAS_STATION` verwenden und muss somit nicht wissen, welche Tankstelle die nächste ist, da der Algorithmus diese findet und gleichzeitig den kürzesten Pfad dorthin berechnet. Ist der Endknoten jedoch bereits vor der Berechnung des Pfades bekannt, kann die `EndCondition` auch mit der Angabe eines bestimmten Knotens erzeugt werden. Aus diesem Grund hat die Klasse zwei Instanzvariablen:

1. `T endVertex`, die den spezifischen Endknoten repräsentiert, und
2. `Predicate<T> condition`, die die Bedingung darstellt, die der Knoten erfüllen muss, damit die Suche beendet wird.

Um die Initialisierung zu vereinfachen besitzt die Klasse zusätzlich zwei statische Methoden, welche als Konstruktoren agieren:

1. `endAt(T vertex)` wird verwendet, um eine `EndCondition<T>` zu erzeugen, die angibt, dass die Suche bei einem bestimmten Knoten enden soll. Die Bedingung wird somit auf `vertex::equals` gesetzt. Damit die Information des Endknotens nicht verloren geht und in Algorithmen wie der bidirektionalen Bestensuche aus *Kapitel 2.3.6* wiederverwendet werden kann, wird `this.vertex` auf den übergebenen `vertex` gesetzt.
2. `endIf(Predicate<T> condition)` wird verwendet, um eine `EndCondition<T>` zu erstellen, die angibt, dass die Suche am ersten gefundenen Knoten enden soll, der die übergebene Bedingung erfüllt. Der Endknoten wird somit auf `null` gesetzt, da er nicht bekannt ist. Demzufolge gibt die Methode `condition()` ein `Optional<T>` zurück, welches leer ist, wenn die `EndCondition` auf diese Art erzeugt wurde.

```
1 @AllArgsConstructor(access = lombok.AccessLevel.PRIVATE)
2 public class EndCondition<T> {
3
4     private final T vertex;
5     private final Predicate<T> condition;
6
7     public static <T> EndCondition<T> endAt(T vertex) {
8         return new EndCondition<>(vertex, vertex::equals);
9     }
10
11    public static <T> EndCondition<T> endIf(Predicate<T> condition) {
12        return new EndCondition<>(null, condition);
13    }
14
15    public Optional<T> vertex() {
16        return Optional.ofNullable(vertex);
17    }
18
19    public Predicate<T> condition() {
20        return condition;
21    }
22
23 }
```

Listing 2.6: EndCondition.java

SearchAlgorithm

Um die Implementierung mehrerer verschiedener Algorithmen zu vereinfachen und zu flexibilisieren, gibt es das generische Interface `SearchAlgorithm<T>`, dessen Definition in *Listing 2.7* zu sehen ist. Es schreibt die Methode `getVisitedVertexCount` vor, welche einen `int` zurückgibt, der angibt, wie viele Knoten bei der letzten Ausführung des Algorithmus besucht werden mussten, um den Pfad zu finden. Diese Metrik ist für das Benchmarking am Ende gemeinsam mit und im Verhältnis zu der Laufzeit ausschlaggebend.

Zusätzlich schreibt es die Methode `findAnyPath` vor, welche als Parameter ein `T start`, eine `EndCondition<T> endCondition` sowie einen `Graph<T> graph` erwartet. Der Parameter `start` stellt den Knoten dar, von dem der Pfad ausgehen soll, `endCondition` die Bedingung, die erfüllt werden muss, damit der Pfad als gefunden gilt und `graph` den Graphen, in dem irgendein Pfad zwischen `start` und dem die `endCondition` erfüllenden Knoten gefunden werden soll. Die Methode `findAnyPath` gibt eine `List<T>` zurück welche den gefundenen Pfad als Knotenfolge repräsentiert. Im Fall, dass es zwischen Start- und Endknoten keinen Pfad gibt, wird von implementierenden Klassen `Collections.emptyList()` zurückgegeben.

```

1 public interface SearchAlgorithm<T> {
2
3     /**
4      * @return the number of visited vertices during the last search.
5      */
6     int getVisitedVertexCount();
7
8     /**
9      * Finds any path between two given vertices in a graph.
10     *
11     * @param start      the vertex the path starts at
12     * @param endCondition the condition that has to
13     *                      be met for the path to end
14     * @param graph       the graph in which a path is to be found
15     * @return any path between the two vertices,
16     * or {@link Collections#emptyList()} if none exists.
17     */
18     List<T> findAnyPath(T start,
19                           EndCondition<T> endCondition,
20                           Graph<T> graph);
21 }

```

Listing 2.7: SearchAlgorithm.java

PathfindingAlgorithm

Das in *Listing 2.8* definierte Interface `PathfindingAlgorithm<T>` erbt von dem soeben erwähnten `searchAlgorithm<T>`-Interface und schreibt eine zusätzliche Methode namens `findShortestPath` vor, welche dieselben Parameter annimmt und denselben Rückgabetypen besitzt, wie `findAnyPath`. Defaultmäßig ist `findAnyPath` hier so implementiert, dass `findShortestPath` mit den übergebenen Parametern aufgerufen wird und der Rückgabewert dieses Aufrufs zurückgegeben wird, da die meisten Pathfinding-Algorithmen ausschließlich diese implementieren. Die beiden Methoden ähneln sich sehr, `findShortestPath` ist allerdings dafür zuständig, nicht nur irgendeinen Pfad zwischen zwei Knoten zu berechnen, sondern den kürzesten. Die Methode `findAnyPath` ist Teil dieses Interfaces, da manche Algorithmen auch auf Arten implementiert werden können, welche nicht unbedingt den optimalen Pfad finden und dadurch performanter sind. Somit kann man also sagen, dass bei sinnhafter Implementierung die Laufzeit von `findAnyPath` im Durchschnitt \leq der Laufzeit von `findShortestPath` ist.

```

1  public interface PathfindingAlgorithm<T> extends SearchAlgorithm<T> {
2
3      /**
4      * Finds the shortest path between two given vertices in a graph.
5      *
6      * @param start          the vertex the path starts at
7      * @param endCondition   the condition that has to
8      *                      be met for the path to end
9      * @param graph          the graph in which a path is to be found
10     * @return the shortest path between the two vertices,
11     * or {@link Collections#emptyList()} if no path exists.
12     */
13    List<T> findShortestPath(T start,
14                             EndCondition<T> endCondition,
15                             Graph<T> graph);
16
17    default List<T> findAnyPath(T start,
18                                EndCondition<T> endCondition,
19                                Graph<T> graph) {
20        return findShortestPath(start, endCondition, graph);
21    }
22
23 }
```

Listing 2.8: PathfindingAlgorithm.java

PathTracer

Die Klasse `PathTracer<T>` aus *Listing 2.9* ist eine Hilfsklasse, mit der aus einer gegebenen Map von Knotenvorgängern ein Pfad rekonstruiert werden kann. Sie hat nur eine Instanzvariable, die `Map<T, T> predecessors` und zwei Methoden, `safeTrace` und `unsafeTrace`.

Die `safeTrace`-Methode verwendet intern ein `LinkedHashSet<T>`, und das aus zwei Gründen:

1. `Set`, um mehrfach vorkommende Knoten effizient zu erkennen und dadurch festzustellen, ob man in einem Zyklus gefangen ist.
2. `LinkedHashSet`, damit die Einfügereihenfolge beibehalten wird und der Pfad am Ende fehlerfrei rekonstruiert werden kann.

Zu Beginn wird der aktuelle Knoten in das Set eingefügt, der zu diesem Zeitpunkt noch der Endknoten des Pfades ist. In einer `while`-Schleife fügt man so lange den Vorgänger des aktuellen Knotens zum Set hinzu und setzt danach den aktuellen Knoten auf den Vorgänger, bis man am Startknoten angelangt ist. Ist ein Knoten bereits im Set vorhanden, wird eine `cycleException` geworfen, siehe *Listing 2.10*. Da der Pfad in umgekehrter Reihenfolge in das Set eingefügt wird, wird der Pfad am Ende umgedreht und als `List<T>` zurückgegeben, damit einzeln auf Elemente an beliebigen Indizes zugegriffen werden kann.

Die Methode `unsafeTrace` bietet eine unsicherere, allerdings um einiges effizientere Alternative zu `safeTrace`, da als Datenstruktur anstatt eines `LinkedHashSet`, welches für seine Ineffizienz berühmt-berüchtigt ist, eine `ArrayList` verwendet wird. Es wird jedoch nicht auf Duplikate überprüft und somit könnte man sich in einem Zyklus verfangen, daher das *unsafe* im Namen. Da viele Pathfinding-Algorithmen so implementiert sind, dass Zyklen vermieden werden, kann `unsafeTrace` in den meisten Fällen ohne Bedenken eingesetzt werden.

```

1  public record PathTracer<T>(Map<T, T> predecessors) {
2
3      /**
4       * This method is less safe than {@link #safeTrace(T, T)} because
5       * it is possible to get stuck in an infinite loop. However, it is
6       * more efficient because it makes use of {@link ArrayList}.
7       */
8      public List<T> unsafeTrace(T start, T end) {
9          var path = new ArrayList<T>();
10         var current = end;
11         path.add(current);
12
13         while (!current.equals(start)) {
14             current = predecessors.get(current);
15             path.addFirst(current);
16         }
17
18         return path;
19     }
20
21     /**
22      * This method is safer than {@link #unsafeTrace(T, T)}
23      * because it checks for cycles in the path. However, it is
24      * less efficient because it makes use of {@link LinkedHashSet}.
25      *
26      * @throws CycleException if a cycle is detected in the path
27      */
28     public List<T> safeTrace(T start, T end) {
29         var path = new LinkedHashSet<T>();
30         var current = end;
31         path.add(current);
32
33         while (!current.equals(start)) {
34             current = predecessors.get(current);
35
36             if (!path.add(current)) {
37                 throw new CycleException(
38                     "The end node is unreachable because of a cycle."
39                 );
40             }
41         }
42
43         return path.reversed()
44             .stream()
45             .toList();
46     }
47
48 }
```

Listing 2.9: PathTracer.java

CycleException

Die `RuntimeException CycleException` aus *Listing 2.10* sollte dann geworfen werden, wenn das Programm in einem Zyklus feststeckt und sich nicht mehr befreien kann. Das kann durchaus passieren: Enthält ein Pfad einen Zyklus und wird dieser Pfad von einem `PathTracer` rekonstruiert, ist diese Fehlermeldung garantiert.

```
1 public class CycleException extends RuntimeException {  
2  
3     public CycleException() {  
4         super();  
5     }  
6  
7     public CycleException(String message) {  
8         super(message);  
9     }  
10 }  
11 }
```

Listing 2.10: `CycleException.java`

2.3.3 Breitensuche

Die Breitensuche, abgekürzt BFS², ist ein Suchalgorithmus für Graphen, der unter Umständen auch dazu verwendet werden kann, den kürzesten Pfad zwischen zwei Knoten zu finden. Damit der Algorithmus in allen Fällen den Pfad mit der geringsten Gesamtgewichtung findet, muss der Graph, auf dem er angewandt wird, folgende Kriterien erfüllen:

1. Er ist ungewichtet oder alle Kanten haben die gleiche Gewichtung
2. Er ist nicht zyklisch

Ist das erste Kriterium nicht erfüllt, findet BFS den Pfad mit der geringsten Knotenzahl, unabhängig von deren individuellen Gewichtungen. Das zweite Kriterium ist nur dann wichtig, wenn man keine Liste von Knoten führt, bei denen man bereits war, da man ansonsten in eine Endlosschleife geraten könnte. Speichert man jedoch die bereits behandelten Knoten ab, ist es völlig egal, ob der Graph zyklisch ist, oder nicht.

Die Breitensuche hat ihren Namen von ihrer Funktionsweise: Für jeden Knoten in einer Queue, welche anfangs nur den Startknoten enthält, wird überprüft, ob er mit dem Zielknoten übereinstimmt. Außerdem werden all seine Nachbarn, die noch nicht besucht wurden und noch nicht in der Queue sind, in diese hinzugefügt. Da Queues auf dem FIFO³-Prinzip basieren, wird also in die Breite gesucht.

Eine Implementierung von BFS ist in *Listing 2.11* zu sehen. Die Streaming-API von Java kommt hier zum Einsatz, um den Code einfach und verständlich zu gestalten. Als Queue wird in diesem Beispiel die Klasse `ArrayDeque` verwendet, wobei Deque für *Double-ended queue steht*. Deques können somit sowohl als Queue als auch als Stack dienen. In der `ArrayList<T> visited` werden alle bereits besuchten Knoten abgespeichert, damit man keine Knoten öfter als einmal besucht und man nicht in Endlosschleifen gerät. In der `HashMap<T, T> predecessors` wird für jeden Knoten abgespeichert, bei welchem Knoten man war, als er in die Queue eingefügt wurde, damit man am Ende den Pfad wiederherstellen kann. Die `while`-Schleife wird so lange ausgeführt, bis keine Elemente mehr in der Queue vorhanden sind. Anfangs ist nur der Startknoten in der Queue, woraufhin all seine Nachbarknoten an das Ende dieser hinzugefügt werden. Bei der nächsten Iteration werden dann die Nachbarn der Nachbarn hinzugefügt und so weiter, bis man am Zielknoten angelangt ist. Gibt es keine unbesuchten Knoten mehr, bricht die Schleife ab und es wird `collections.emptyList()` zurückgegeben. Wurde der Zielknoten jedoch erreicht, wird der Pfad mithilfe der `unsafeTrace`-Methode der `PathTracer`-Klasse wiederhergestellt und zurückgegeben, *siehe Listing*

²Breadth-First Search

³First In - First Out

2.9. Es wird `unsafeTrace` bevorzugt, da es viel performanter ist als `safeTrace` und nicht auf Zyklen überprüft werden muss. Diese Aufgabe wird bei der Breitensuche schon von der `ArrayList<T>` `visited` übernommen.

```

1  @Getter
2  public class BreadthFirstSearch<T> implements PathfindingAlgorithm<T> {
3
4      private int visitedVertexCount;
5
6      @Override
7      public List<T> findShortestPath(T start,
8                                      EndCondition<T> endCondition,
9                                      Graph<T> graph) {
10
11         var queue = new ArrayDeque<T>();
12         var visited = new ArrayList<T>();
13         var predecessors = new HashMap<T, T>();
14         queue.add(start);
15         visitedVertexCount = 0;
16
17         while (!queue.isEmpty()) {
18             T current = queue.poll();
19             visited.add(current);
20             visitedVertexCount++;
21
22             if (endCondition.condition().test(current)) {
23                 var pathTracer = new PathTracer<>(predecessors);
24                 return pathTracer.unsafeTrace(start, current);
25             }
26
27             graph.getNeighbors(current)
28                 .keySet()
29                 .stream()
30                 .filter(neighbor -> !visited.contains(neighbor))
31                 .filter(neighbor -> !queue.contains(neighbor))
32                 .forEach(neighbor -> {
33                     queue.offer(neighbor);
34                     predecessors.put(neighbor, current);
35                 });
36
37         }
38
39         return Collections.emptyList();
40     }

```

Listing 2.11: BFS-Algorithmus in Java

2.3.4 Tiefensuche

Die Tiefensuche, abgekürzt DFS⁴, ist ein weiterer Suchalgorithmus für Graphen, welcher auch für Pathfinding eingesetzt werden kann. Mit dem standardmäßigen DFS-Algorithmus findet man jedoch nicht den kürzesten Pfad, sondern irgendeinen. Vorteil ist, dass der Algorithmus schnell ist, er hat also eine vergleichsweise kurze Laufzeit, da er nicht darauf ausgelegt ist, den optimalen Pfad zu finden. Der offensichtliche Nachteil ist dadurch aber, dass der gefundene Pfad sehr lange sein kann und im Großteil der Fälle nicht der kürzeste ist.

Die Tiefensuche hat ihren Namen ebenfalls von ihrer Funktionsweise, die sehr ähnlich zur Breitensuche ist: Für jeden Knoten in einem Stack, welcher anfangs nur den Startknoten enthält, wird überprüft, ob er mit dem Zielknoten übereinstimmt. Es werden all seine Nachbarn, die noch nicht im Stack sind, in diesen hinzugefügt. Da Stacks auf dem LIFO⁵-Prinzip basieren, wird also in die Tiefe gesucht. Der einzige logische Unterschied zu BFS ist also die Verwendung eines Stacks anstatt einer Queue.

Iterativ

Eine iterative Implementierung von DFS ist in *Listing 2.12* in der Methode `findAnyPath` zu sehen. Die Streaming-API wird hier erneut angewendet. Wie bei BFS wird hier eine `ArrayDeque` als Datenstruktur verwendet, mit dem Unterschied, dass sie hier als Stack verwendet wird, anstatt als Queue. Es wird nicht die Klasse `stack` verwendet, da diese thread-safe und aufgrund ihrer `synchronized`-Methoden sehr langsam ist. Da die gezeigte Implementierung single-threaded ist, stellt die Verwendung von Klassen und Methoden, welche nicht thread-safe sind, kein Problem dar. Auch hier wird, wie bei der `findShortestPath`-Methode von BFS, in jeder Iteration der `while`-Schleife der `findAnyPath`-Methode überprüft, ob der Zielknoten erreicht wurde. Bricht die Schleife ab, wird `Collections.emptyList()` zurückgegeben.

Um garantiert den kürzesten Pfad zu finden, kann eine modifizierte Version von DFS angewendet werden, welche in `findShortestPath` implementiert ist. Diese ist jedoch extrem ineffizient, da alle Pfade durchsucht werden und erst dann der mit der geringsten Gewichtung zurückgegeben werden kann. Hier wird keine Liste von bereits besuchten Knoten geführt, da Knoten mehrmals besucht werden müssen, falls der Algorithmus einen Pfad gewählt hat, der nicht optimal ist, dieser Pfad jedoch Knoten des kürzesten Pfades enthält. Außerdem könnte der Zielknoten dann nur einmal besucht werden, was sinnlos ist, da es dann nur einen Pfad geben könnte, der mit hoher Wahr-

⁴Depth-First Search

⁵Last In - First Out

scheinlichkeit nicht der kürzeste wäre. Es reicht nicht, für den Zielknoten eine Ausnahme zu machen und ihn nicht in die Liste von besuchten Knoten aufzunehmen, da es für andere Knoten ebenso wichtig ist, häufiger besucht werden zu können, falls sie Teil des optimalen Pfades sind. Eine elegante Lösung für dieses Problem ist nicht zu überprüfen, ob ein Knoten bereits besucht wurde, sondern ob er Teil des aktuellen Pfades ist. Dies kann mithilfe der `predecessors`-Map in Kombination mit der `PathTracer`-Klasse erzielt werden, indem man von jedem Knoten aus den Pfad zum Startknoten rekonstruiert. Mit der Zeile `.filter(neighor -> !path.contains(neighor))` werden dann alle Nachbarknoten herausgefiltert, die bereits im aktuellen Pfad enthalten sind. Danach werden mit `.filter(neighor -> !stack.contains(neighor))` noch alle Nachbarn verworfen, die schon im Stack sind und nur noch darauf warten, besucht zu werden. Die `ArrayList<List<T>> paths` wird dazu verwendet, alle gefundenen Pfade vom Start- zum Zielknoten zu speichern. Aus diesen wird zum Schluss der kürzeste bestimmt und zurückgegeben.

```
1  @Getter
2  public class DepthFirstSearch<T> implements PathfindingAlgorithm<T> {
3
4      private int visitedVertexCount;
5
6      @Override
7      public List<T> findAnyPath(T start,
8                                  EndCondition<T> endCondition,
9                                  Graph<T> graph) {
10         var predecessors = new HashMap<T, T>();
11         var stack = new ArrayDeque<T>();
12         var visited = new ArrayList<T>();
13         stack.push(start);
14         visitedVertexCount = 0;
15
16         while (!stack.isEmpty()) {
17             T current = stack.pop();
18             visited.add(current);
19             visitedVertexCount++;
20
21             if (endCondition.condition().test(current)) {
22                 var pathTracer = new PathTracer<>(predecessors);
23                 return pathTracer.unsafeTrace(start, current);
24             }
25
26             graph.getNeighbors(current)
27                 .keySet()
28                 .stream()
29                 .filter(neighbor -> !visited.contains(neighbor))
30                 .filter(neighbor -> !stack.contains(neighbor))
31                 .forEach(neighbor -> {
32                     stack.push(neighbor);
33                     predecessors.put(neighbor, current);
34                 });
35         }
36
37         return Collections.emptyList();
38     }
39
40     @Override
41     public List<T> findShortestPath(T start,
42                                     EndCondition<T> endCondition,
43                                     Graph<T> graph) {
44         var predecessors = new HashMap<T, T>();
45         var stack = new ArrayDeque<T>();
46         var paths = new ArrayList<List<T>>();
47         var pathTracer = new PathTracer<>(predecessors);
48         stack.push(start);
49         visitedVertexCount = 0;
50
51         while (!stack.isEmpty()) {
```

```
52     T current = stack.pop();
53     var path = pathTracer.unsafeTrace(start, current);
54     visitedVertexCount++;
55
56     if (endCondition.condition().test(current)) {
57         paths.add(path);
58         continue;
59     }
60
61     graph.getNeighbors(current)
62         .keySet()
63         .stream()
64         .filter(neighbor -> !path.contains(neighbor))
65         .filter(neighbor -> !stack.contains(neighbor))
66         .forEach(neighbor -> {
67             stack.push(neighbor);
68             predecessors.put(neighbor, current);
69         });
70
71
72     return paths.stream()
73         .min(Comparator.comparingDouble(graph::sumEdgeWeights))
74         .orElse(Collections.emptyList());
75 }
76
77 }
```

Listing 2.12: Iterativer DFS-Algorithmus in Java

Rekursiv

Die Tiefensuche kann nicht nur iterativ implementiert werden, sondern auch rekursiv, siehe Listing 2.13. Rekursive Implementierungen haben den Vorteil, dass sie viel sauberer und einfacher zu verstehen sind, da keine Variablen wie `stack`, `predecessors`, `visited` (für `findAnyPath`), `paths` (für `findShortestPath`) oder andere benötigt werden. Bei rekursiven DFS-Implementierungen macht man sich den internen Callstack zunutze und verwendet ihn als Stack für die traversierten Knoten und Pfade. So ergibt sich der Nachteil, dass bei der Anwendung in gigantischen Graphen Stackoverflows auftreten können. Außerdem ist die rekursive Variante um einiges langsamer und ineffizienter, als die iterative.

Man muss hier den zurückgelegten Pfad nicht bei jedem Schritt rekonstruieren, sondern kann ihn einfach beim nächsttieferen Methodenaufruf übergeben und immer den aktuellen Knoten ans Ende anfügen. Wichtig ist hierbei jedoch, dass jeder Rekursionsbranch ein eigenes Listenobjekt erhält und somit keine Änderungen in anderen Branches bewirken kann. Darum kümmert sich die Methode `append`: Diese fügt nicht nur ein Element an das Ende einer gegebenen Liste an, sondern sorgt auch dafür, dass die übergebene Liste unverändert bleibt und eine Kopie von dieser erzeugt und mit dem hinzugefügten Element zurückgegeben wird. Somit frisst die rekursive Implementierung Unmengen an Speicherplatz, da für jeden Methodenaufruf eine Kopie des gesamten Pfades erzeugt wird.

Die Zeile `.filter(neighbor -> !path.contains(neighbor))` sorgt hier, wie bei der iterativen Variante, dafür, dass ausschließlich Knoten behandelt werden, die nicht bereits Teil des zurückgelegten Pfades sind. Danach wird von jedem Nachbarn des aktuellen Startknotens ausgehend irgendein bzw. der kürzeste Pfad zum Zielknoten gesucht und anschließend alle leeren Pfade, also alle `Collections.emptyList()`, entfernt, und die übrigen weiterverarbeitet. Bei `findAnyPath` wird zum Schluss nur noch `Stream::findAny` aufgerufen, wohingegen bei `findShortestPath` stattdessen `Stream::min` mit dem Comparator `Comparator.comparingDouble(graph::sumEdgeWeights)` aufgerufen und somit der optimale Pfad zurückgegeben wird.


```
51     visitedVertexCount++;
52
53     if (endCondition.condition().test(start)) {
54         return path;
55     }
56
57     return graph.getNeighbors(start)
58         .keySet()
59         .stream()
60         .filter(neighbor -> !path.contains(neighbor))
61         .map(neighbor -> findShortestPath(
62             neighbor,
63             endCondition,
64             graph,
65             append(path, neighbor)
66         ))
67         .filter(list -> !list.isEmpty())
68         .min(Comparator.comparingDouble(graph::sumEdgeWeights))
69         .orElse(Collections.emptyList());
70 }
71
72 private List<T> append(List<T> list, T element) {
73     var appended = new ArrayList<>(list);
74     appended.add(element);
75     return appended;
76 }
77
78 }
```

Listing 2.13: Rekursiver DFS-Algorithmus in Java

2.3.5 Bestensuche

Bestensuchen wählen immer den zum Zeitpunkt der Wahl vielversprechendsten Knoten, welcher anhand einer bestimmten Bewertungsfunktion

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

bestimmt wird, die die Gesamtkosten vom Start- zum Zielknoten durch den Knoten n berechnet oder schätzt. $g(n)$ entspricht hier den (tatsächlichen) Kosten vom Startknoten zum Knoten n und $h(n)$ ist die sogenannte *Heuristik*, also die geschätzten Kosten vom Knoten n zum Zielknoten.

Eine solche Implementierung ist in der Klasse `AbstractBestFirstSearch` aus *Listing 2.15* zu sehen, welche das Interface `BestFirstSearch` aus *Listing 2.14* implementiert, jedoch trotzdem abstrakt ist, da sie zwei von `BestFirstSearch` vorgeschriebene Methoden noch nicht implementiert.

Diese sind einerseits `double g(T vertex, Map<T, Double> distances)` und andererseits `double h(T vertex, EndCondition<T> endCondition)`, welche den Kostenfunktionen $g(n)$ und $h(n)$ entsprechen. Diese werden zur Berechnung der Prioritäten der Knoten verwendet.

```

1  public interface BestFirstSearch<T> extends PathfindingAlgorithm<T> {
2
3      Map<T, Double> getDistances();
4
5      Map<T, T> getPredecessors();
6
7      T getCurrent();
8
9      void closeCurrent();
10
11     boolean nextOpen();
12
13     boolean hasVisited(T vertex);
14
15     boolean hasOpen();
16
17     void initializeDataStructures(T start);
18
19     Map<T, Double> expand(EndCondition<T> endCondition, Graph<T> graph);
20
21     double g(T vertex, Map<T, Double> distances);
22
23     double h(T vertex, EndCondition<T> endCondition);
24
25     default boolean nextUnvisited() {
26         while (nextOpen()) {
27             if (!hasVisited(getCurrent())) {
28                 return true;
29             }
30         }
31
32         return false;
33     }
34
35     default double g(T vertex) {
36         return g(vertex, getDistances());
37     }
38
39 }
```

Listing 2.14: BestFirstSearch.java

Die Klasse `AbstractBestFirstSearch` besitzt einige Instanzvariablen:

1. Die `Map<T, Double> distances`, die die Distanzen jedes Knotens zum Startknoten speichert und bei Dijkstra und A* für die Evaluierung von $g(n)$ benötigt wird.
2. Die `Map<T, T> predecessors`, die die Vorgänger aller Knoten im Pfad speichert.

3. Das `Set<T> closed`, in dem alle bereits besuchten Knoten abgespeichert werden.
4. Den `FibonacciHeap<T> open`, in dem die zu besuchenden Knoten gemeinsam mit ihrer Priorität gespeichert werden.
5. Das `T current`, welches den aktuell besuchten Knoten abspeichert. Es ist eine Instanzvariable, damit auch von außen darauf zugegriffen kann, was in *Kapitel 2.3.6* noch wichtig wird.

Zu Beginn der `findShortestPath`-Methode werden die Datenstrukturen initialisiert und die Distanz des Startknotens zu sich selbst auf 0 gesetzt, wonach er zur Queue bzw. zum Heap hinzugefügt wird. In einer `while`-Schleife wird, solange die Queue nicht leer ist, für jeden Nachbarn des aktuellen Knotens überprüft, ob der Weg über den aktuellen Knoten kürzer ist als der bisher bekannte Weg zu diesem Nachbarn. Ist dies der Fall, wird die Distanz aktualisiert, der Vorgänger des Nachbars auf den aktuellen Knoten gesetzt und der Nachbar in die Queue eingefügt. Bei jedem Schleifendurchlauf wird der erste Knoten in der Queue ausgewählt, welcher dem Knoten n mit dem kleinsten $f(n)$ entspricht. Erfüllt dieser Knoten die Endbedingung, wird die Schleife beendet und der kürzeste Pfad vom Startknoten zum aktuellen Knoten mithilfe eines `PathTracer` kalkuliert und zurückgegeben. Bricht die Schleife ab, wird `Collections.emptyList()` zurückgegeben, da kein Pfad gefunden wurde. Damit dieser Algorithmus funktioniert, ist vorausgesetzt, dass keine Kanten negativ gewichtet sind und die eingesetzte Heuristik *zulässig* ist, mehr dazu in *Kapitel 2.3.5*.

```
1  @Getter
2  public abstract class AbstractBestFirstSearch<T>
3      implements BestFirstSearch<T> {
4
5      private final Map<T, Double> distances = new HashMap<>();
6      private final Map<T, T> predecessors = new HashMap<>();
7      private final Set<T> closed = new HashSet<>();
8      private FibonacciHeap<T> open;
9      private T current;
10
11     @Override
12     public int getVisitedVertexCount() {
13         return closed.size();
14     }
15
16     @Override
17     public List<T> findShortestPath(T start,
18                                     EndCondition<T> endCondition,
19                                     Graph<T> graph) {
20         initializeDataStructures(start);
21
22         while (nextUnvisited()) {
23             if (endCondition.condition().test(current)) {
24                 var pathTracer = new PathTracer<>(predecessors);
25                 return pathTracer.unsafeTrace(start, current);
26             }
27
28             closeCurrent();
29             expand(endCondition, graph);
30         }
31
32         return Collections.emptyList();
33     }
34
35     @Override
36     public boolean nextOpen() {
37         if (hasOpen()) {
38             current = open.dequeueMin().getValue();
39             return true;
40         }
41
42         current = null;
43         return false;
44     }
45
46     @Override
47     public void closeCurrent() {
48         closed.add(current);
49     }
50
51     @Override
```

```
52     public boolean hasVisited(T vertex) {
53         return closed.contains(vertex);
54     }
55
56     @Override
57     public boolean hasOpen() {
58         return !open.isEmpty();
59     }
60
61     @Override
62     public void initializeDataStructures(T start) {
63         predecessors.clear();
64         distances.clear();
65         distances.put(start, 0.0);
66         closed.clear();
67         open = new FibonacciHeap<>();
68         open.enqueue(start, 0.0);
69     }
70
71     @Override
72     public Map<T, Double> expand(EndCondition<T> endCondition,
73                                   Graph<T> graph) {
74         return graph.getNeighbors(current)
75             .entrySet()
76             .stream()
77             .filter(entry -> !hasVisited(entry.getKey()))
78             .filter(entry -> {
79                 T neighbor = entry.getKey();
80                 double weight = entry.getValue();
81                 double g = g(neighbor);
82                 double tentativeG = g(current) + weight;
83
84                 if (tentativeG < g) {
85                     distances.put(neighbor, tentativeG);
86                     predecessors.put(neighbor, current);
87                     double heuristic = h(neighbor, endCondition);
88                     open.enqueue(neighbor, tentativeG + heuristic);
89                     return true;
90                 }
91
92                 return false;
93             })
94             .collect(Collectors.toMap(
95                 Map.Entry::getKey,
96                 Map.Entry::getValue
97             ));
98     }
99
100 }
```

Listing 2.15: AbstractBestFirstSearch.java

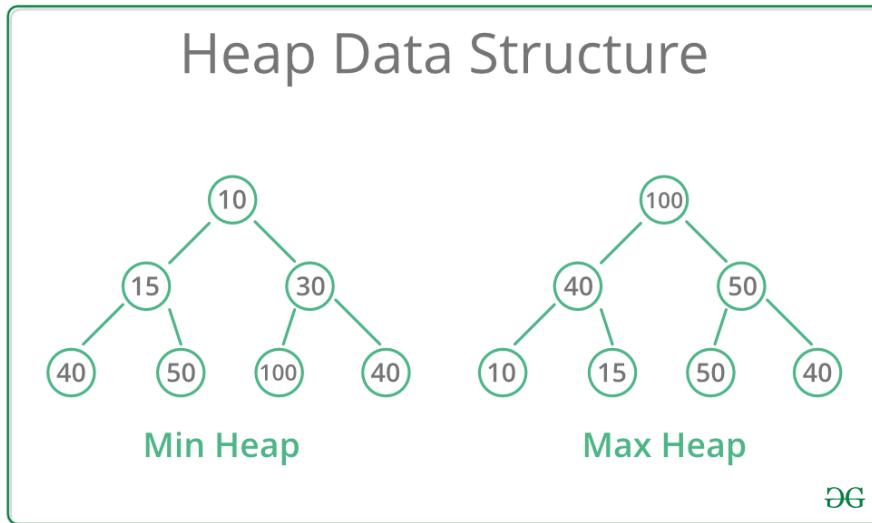


Abbildung 2.16: Beispiele für Min- und Max-Heap
[EZ:Web60]

Fibonacci-Heap

Der Fibonacci-Heap übernimmt hier die Aufgabe einer Priority-Queue, er speichert also Elemente so, dass möglichst effizient auf das Element mit der höchsten Priorität zugegriffen werden kann. Anstelle des Fibonacci-Heaps könnte hier auch die von Java vorgegebene `PriorityQueue`-Klasse angewendet werden, welche intern als binärer Min-Heap implementiert ist. Min- und Max-Heaps haben eine zum Binärbaum ähnliche Struktur, speichern in ihrer Wurzel jedoch immer das kleinste bzw. größte Element des Heaps, wobei die Kind-Elemente dieser immer größer bzw. kleiner sein müssen als die Wurzel. Diese Regel gilt auch für alle Sub-Bäume. In *Abbildung 2.16* ist jeweils ein Beispiel für einen Min- und einen Max-Heap zu sehen.

Da Min-Heaps im Zusammenhang mit Pathfinding jedoch eine schlechtere Performance aufweisen, wird hier die von Keith Schwarz implementierte `FibonacciHeap`-Klasse verwendet. Die Struktur eines solchen Heaps ist in *Abbildung 2.17* visualisiert. Ein Fibonacci-Heap besteht aus einer Liste mehrerer Bäume. Dabei kann jeder Baum auch nur aus einem einzigen Element bestehen. Werden Operationen wie `decreaseKey` oder `insert` bzw. `enqueue` nur selten aufgerufen, so wäre eine andere Datenstruktur wie z.B. ein binärer Heap wahrscheinlich effizienter. Bei Bestensuchen werden jedoch bei einem sehr großen Teil der Iterationen neue Knoten mittels `enqueue` zur Open-List hinzugefügt, weshalb der Fibonacci-Heap hier so effizient ist. Die Zeitkomplexitäten von Min- und Fibonacci-Heap werden in *Tabelle 2.6* gegenübergestellt. [EZ:Web53, EZ:Web56].

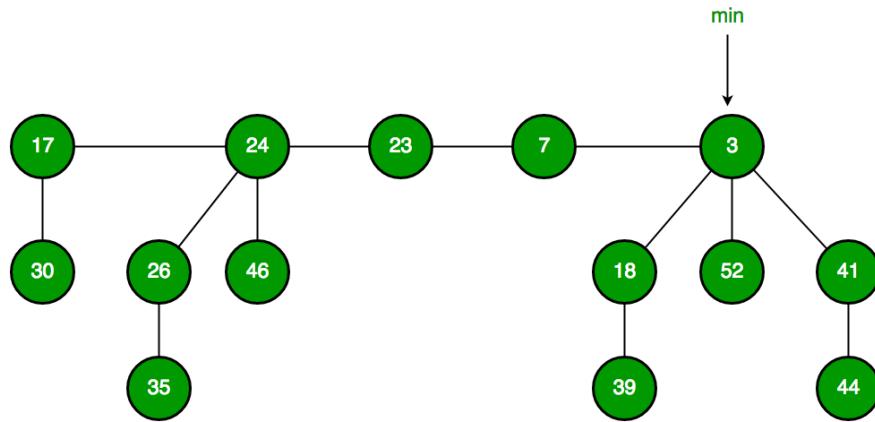


Abbildung 2.17: Struktur eines Fibonacci-Heaps
[EZ:Web54]

| Operation | Min-Heap | Fibonacci-Heap |
|-------------|------------------------------------|-----------------------|
| insert | $\mathcal{O}(\log n)$ | $\mathcal{O}(1)$ |
| getMin | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |
| extractMin | $\mathcal{O}(\log n)$ | $\mathcal{O}(\log n)$ |
| decreaseKey | $\mathcal{O}(\log n)$ | $\mathcal{O}(1)$ |
| remove | $\mathcal{O}(\log n)$ | $\mathcal{O}(\log n)$ |
| merge | $\mathcal{O}(m \cdot \log(n + m))$ | $\mathcal{O}(1)$ |

Tabelle 2.6: Vergleich der Zeitkomplexitäten des Min- und Fibonacci-Heaps
[EZ:Web55]

Würde man in *Listing 2.15* anstatt eines Fibonacci-Heaps eine Java-**PriorityQueue** verwenden, so könnte die Initialisierung wie in *Listing 2.16* aussehen. Der übergebene **Comparator** sorgt dafür, dass die Priorität eines eingefügten Knotens n anhand von $f(n)$, also von der Summe von $g(n)$ und $h(n)$, bestimmt wird. Gilt $f(n_1) = f(n_2)$ für zwei Knoten n_1 und n_2 , so hängt die Priorität ausschließlich von $h(n)$ ab.

```

1 open = new PriorityQueue<>(
2     Comparator.<T>comparingDouble(
3         vertex -> g(vertex) + h(vertex, endCondition)
4     )
5     .thenComparingDouble(
6         vertex -> h(vertex, endCondition)
7     )
8 );

```

Listing 2.16: Priority-Queue mit Comparator für Best-first search

Mithilfe der Closed-List kann festgestellt werden, ob ein Knoten bereits besucht wurde und die aktuelle Iteration übersprungen werden, wenn das der Fall ist. Theoretisch wä-

re diese aber gar nicht unbedingt notwendig, würde man nämlich beim Aktualisieren der Priorität eines Knotens dafür sorgen, dass er an seine designierte Position verschoben wird, anstatt ihn neu einzufügen, und würde man eine *monotone* Heuristik verwenden (*siehe Kapitel 2.3.5*), so wäre garantiert, dass kein Knoten öfter als einmal besucht wird, da kein Knoten mehrfach in der Open-List vorkommen könnte, und gleichzeitig würde man sogar Speicherplatz sparen. Das könnte in etwa wie in *Listing 2.17* aussehen, wobei `entries` eine `Map<T, FibonacciHeap.Entry<T>>` ist, welche jedem Knoten eine Referenz auf seinen Eintrag im Fibonacci-Heap zuordnet.

```

1 if (entries.containsKey(neighbor)) {
2     open.decreaseKey(entries.get(neighbor), tentativeG + heuristic);
3 } else {
4     entries.put(neighbor, open.enqueue(neighbor, tentativeG + heuristic));
5 }
```

Listing 2.17: Aktualisierung der Prioritäten ohne Knoten mehrfach einzufügen

Die `decreaseKey`-Operation hat hier die Aufgabe, die Priorität auf einen niedrigeren Wert zu ändern, und diese somit zu erhöhen. Den Wert selbst zu erhöhen wäre nicht möglich, da sonst eine Exception geworfen werden würde, *siehe Listing 2.18*. Da `decreaseKey` performancemäßig jedoch ziemlich teuer ist, wird hier ein einfaches `open.enqueue(neighbor, tentativeG + heuristic)` bevorzugt, obwohl die Priorität so nur von $f(n)$ abhängt und Knoten mit demselben $f(n)$ untereinander keine bestimmte Ordnung haben. Stattdessen wird bei jeder Iteration überprüft, ob der aktuelle Knoten bereits besucht wurde und diese übersprungen, falls das der Fall ist. Zusätzlich werden mit der Zeile `.filter(entry -> !hasVisited(entry.getKey()))` nur die Nachbarn herausgefiltert, die noch nicht besucht wurden, um die Anzahl zu überspringender Iterationen möglichst gering zu halten.

```

1 public void decreaseKey(Entry<T> entry, double newPriority) {
2     checkPriority(newPriority);
3     if (newPriority > entry.mPriority)
4         throw new IllegalArgumentException("New priority exceeds old.");
5
6     /* Forward this to a helper function. */
7     decreaseKeyUnchecked(entry, newPriority);
8 }
```

Listing 2.18: `decreaseKey`-Methode aus `FibonacciHeap.java`

Hierbei stellt `checkPriority` sicher, dass die Priorität nicht auf `Double.NaN` gesetzt werden kann, *siehe Listing 2.19*.

```

1 private void checkPriority(double priority) {
2     if (Double.isNaN(priority))
3         throw new IllegalArgumentException(priority + " is invalid.");
4 }
```

Listing 2.19: `checkPriority`-Methode aus `FibonacciHeap.java`

Dijkstra-Algorithmus

What is the shortest way to travel from Rotterdam to Groningen, in general: from given city to given city. It is the algorithm for the shortest path, which I designed in about twenty minutes. One morning I was shopping in Amsterdam with my young fiancée, and tired, we sat down on the café terrace to drink a cup of coffee and I was just thinking about whether I could do this, and I then designed the algorithm for the shortest path. As I said, it was a twenty-minute invention. In fact, it was published in '59, three years later. The publication is still readable, it is, in fact, quite nice. One of the reasons that it is so nice was that I designed it without pencil and paper. I learned later that one of the advantages of designing without pencil and paper is that you are almost forced to avoid all avoidable complexities. Eventually, that algorithm became to my great amazement, one of the cornerstones of my fame. [EZ:Web47, Edsger W. Dijkstra, 2001]

Der Algorithmus von Dijkstra wurde im Jahr 1956 von Edsger W. Dijkstra entworfen und drei Jahre später veröffentlicht. Er findet in vielen verschiedenen Bereichen Anwendung, und das nicht nur bei der Navigation im Straßenverkehr, sondern zum Beispiel auch bei Routing-Protokollen für Computernetzwerke. Die zwei bekanntesten sind OSPF⁶ und IS-IS⁷, welche auf dem Dijkstra-Algorithmus aufbauen. [EZ:Web48, EZ:Web49]

Genau genommen ist der Dijkstra-Algorithmus ein Sonderfall von A* und somit auch der allgemeinen Bestensuchen. Dijkstra ist hierbei ein Sonderfall, da immer $h(n) = 0$ und somit $f(n) = g(n)$ gilt, es wird also nur die Distanz zum Startknoten zur Bewertung miteinbezogen, die Entfernung zum Endknoten spielt somit keine Rolle. Ein weiteres wichtiges Detail des Dijkstra-Algorithmus ist, dass er normalerweise nicht nur den kürzesten Pfad zwischen zwei Knoten findet, sondern zu allen Knoten im Graphen von einem bestimmten Startpunkt aus. Um seine Effizienz zu verbessern, kann er jedoch modifiziert werden, sodass die Suche beendet wird, sobald er am Zielknoten angelangt ist bzw. die Endbedingung erreicht wurde. [EZ:Web08, EZ:Web51, EZ:Web52]

Die konkrete Implementierung des Dijkstra-Algorithmus ist in *Listing 2.20* veranschaulicht. Die Klasse `Dijkstra` erbt von `AbstractBestFirstSearch` und implementiert die Methode `g(T vertex, Map<T, Double> distances)` so, dass immer die Distanz des übergebenen Knotens zum Startpunkt zurückgegeben wird, bzw. ∞ , falls diese noch nicht bekannt ist. Die Methode `h(T vertex, EndCondition<T> endCondition)` zur Berechnung der Heuristik gibt hier immer den Wert 0 zurück.

⁶Open Shortest Path First

⁷Intermediate System to Intermediate System

Wie bereits in *Kapitel 2.3.5* erwähnt, wäre es mit dem Einsatz einer monotonen Heuristik nicht unbedingt erforderlich, eine Liste von bereits besuchten Knoten zu führen, um sicherzustellen, dass der Algorithmus korrekt funktioniert und keine Knoten mehr als einmal besucht werden. Dieses Kriterium wird vom Dijkstra-Algorithmus immer erfüllt, da dieser eine implizite Heuristik von $h(n) = 0$ verwendet, welche somit monoton ist. Dass nun kein Knoten mehrfach besucht wird, wird von zwei Eigenschaften gewährleistet, wobei die zweite von der ersten impliziert wird:

1. **Priorisierung kürzerer Pfade:** Der verwendete Fibonacci-Heap stellt aufgrund seiner Priority-Queue-Eigenschaften sicher, dass immer der Knoten mit dem aktuell kürzesten bekannten Pfad vom Startknoten aus besucht wird. So werden alle kürzeren Pfade vor längeren erkundet und es ist ausgeschlossen, dass ein Knoten auf einem längeren Umweg erneut besucht wird, da der Pfad dazu kürzer sein müsste. [EZ:Web50]
2. **Monotonie der Distanzen:** Da die Entfernung vom Startknoten beim Verfolgen eines Pfades immer um die Gewichtungen der traversierten Kanten erhöht wird, kann die Distanz vom Startknoten zum aktuellen Knoten niemals kleiner sein als die Distanz zum Vorgängerknoten. Auch der Vorgängerknoten wird daher, obwohl er in Digraphen manchmal bzw. in ungerichteten Graphen immer Nachbar des aktuellen Knotens ist, nie mehr als einmal besucht werden, da er bereits auf einem kürzeren Pfad erreicht wurde.

```

1 public class Dijkstra<T> extends AbstractBestFirstSearch<T> {
2
3     @Override
4     public double g(T vertex, Map<T, Double> distances) {
5         return distances.getOrDefault(vertex, Double.POSITIVE_INFINITY);
6     }
7
8     @Override
9     public double h(T vertex, EndCondition<T> endCondition) {
10        return 0;
11    }
12
13 }
```

Listing 2.20: Dijkstra-Algorithmus in Java

A*-Algorithmus

A^* ist ebenfalls ein Algorithmus aus der Klasse der BeFS⁸-Algorithmen. Im Gegensatz zum Dijkstra-Algorithmus, bei dem immer $h(n) = 0$ gilt, spielt die Heuristik $h(n)$ hier jedoch eine wichtige Rolle.

Eine Implementierung von A^* ist in *Listing 2.21* zu sehen. Die Vererbungshierarchie von `Astar` ist von der von `ArrayList` inspiriert. Die konkrete Klasse `Astar` erbt von der abstrakten Klasse `AbstractBestFirstSearch` und implementiert somit die zwei Kostenfunktionen g und h . Die Methode `g(T vertex, Map<T, Double> distances)` ist wie bei Dijkstra implementiert und `h(T vertex, EndCondition<T> endCondition)` gibt den Wert der an den Konstruktor von `Astar` übergebenen und auf die Parameter angewendeten `Heuristic<T> heuristic` zurück.

```

1  @RequiredArgsConstructor
2  public class AStar<T> extends AbstractBestFirstSearch<T> {
3
4      private final Heuristic<T> heuristic;
5
6      @Override
7      public final double g(T vertex, Map<T, Double> distances) {
8          return distances.getOrDefault(vertex, Double.POSITIVE_INFINITY);
9      }
10
11     @Override
12     public final double h(T vertex, EndCondition<T> endCondition) {
13         return heuristic.applyAsDouble(vertex, endCondition);
14     }
15
16 }
```

Listing 2.21: A*-Algorithmus in Java

Das Interface `Heuristic` aus *Listing 2.22* ist ein `FunctionalInterface`, welches von `ToDoubleBiFunction<T, EndCondition<T>>` erbt. Es wird ein `FunctionalInterface` verwendet, damit die `h`-Methode und somit auch die `Astar`-Klasse nicht abstrakt sein muss, und `h` dadurch nicht jedes Mal bei der Verwendung einer neuen Heuristik neu implementiert werden muss, sondern einfach als Konstruktorparameter übergeben werden kann. [EZ:Web08, EZ:Web18]

```

1  @FunctionalInterface
2  public interface Heuristic<T>
3      extends ToDoubleBiFunction<T, EndCondition<T>> {
4
5 }
```

Listing 2.22: Heuristic.java

⁸Best-first search (nicht zu verwechseln mit Breadth-first search)

Damit A* garantiert den optimalen Pfad finden kann, muss die Heuristik *zulässig* (englisch *admissible*) sein. Das ist sie dann und nur dann, wenn die von ihr geschätzte Distanz $h(n)$ vom aktuellen Knoten zum Endknoten nie die tatsächliche Mindestdistanz überschreitet. Ist die Zulässigkeit der verwendeten Heuristik nicht gegeben, so ist es besser, einen anderen Algorithmus zu verwenden, wie z.B. Dijkstra, da dieser eine monotone Heuristik von $h(n) = 0$ verwendet.

Damit eine Heuristik als *monoton* gilt, muss zusätzlich

$$\forall n \in V : h(n) \leq h(n') + w(n, n')$$

gelten. Der Wert der Heuristik muss also für jeden Knoten n höchstens so groß sein wie das Maximum der Heuristik-Werte all seiner Nachbarn n' plus dem Kantengewicht zwischen n und n' . Anders ausgedrückt darf der Wert der Heuristik für keinen Knoten n zu seinen Nachbarknoten n' um mehr als die Gewichtung der Kante zwischen diesen sinken. Monotone bzw. *konsistente* Heuristiken sind immer zulässig, umgekehrt ist dies jedoch nicht unbedingt der Fall. Verwendet A* eine monotone Heuristik, ist sichergestellt, dass jeder Knoten nur einmal besucht werden muss, da ein Knoten so erst dann besucht wird, wenn der kürzeste Pfad zu ihm gefunden wurde. [EZ:Web18, EZ:Web57]

Nodes expanded in a [bidirectional](#) search vs. those expanded in a [unidirectional](#) search.

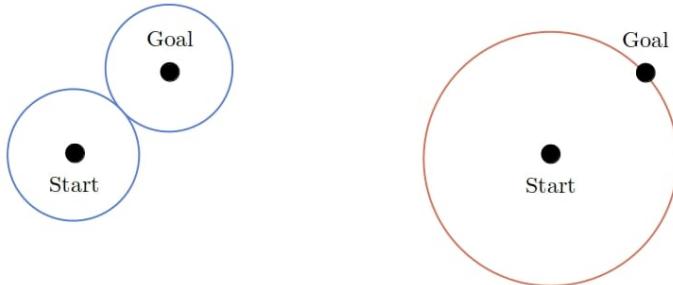


Abbildung 2.18: Die Idee hinter der bidirektonalen Suche
[EZ:Web62]

2.3.6 Bidirektionale Bestensuche

Die bidirektionale Bestensuche, welche auch als *Meet In The Middle*-Algorithmus bezeichnet wird, ist eine Erweiterung der normalen, unidirektionalen Bestensuche. Anstatt einer einfachen Suche, die terminiert, sobald sie einen bzw. den Knoten gefunden hat, der die Endbedingung erfüllt, werden hier zwei Suchen gleichzeitig verwendet. Gleichzeitig jedoch nicht im Sinne von parallel, sondern zwischen den beiden Suchen abwechselnd. Die erste ist hierbei die *Vorwärtssuche*, die wie gewohnt von Anfang bis Ende sucht. Die zweite, die *Rückwärtssuche*, beginnt ihre Suche nun aber beim Endknoten und sucht in Richtung Startknoten. Das bedeutet, dass für das Funktionieren solch einer bidirektionalen Suche die Endbedingung dieser via `EndCondition::endAt` erzeugt werden muss, da der Endknoten bei einer Erzeugung via `EndCondition::endIf` nicht im Vorhinein bekannt wäre. Treffen die beiden Suchen aufeinander, wird der Pfad vom Startknoten zum überlappenden Knoten berechnet, woraufhin dieser Pfad mit dem vom überlappenden Knoten zum Endpunkt verschmolzen wird. Deshalb wird die bidirektionale Suche auch als *Meet In The Middle*-Algorithmus bezeichnet, da die beiden Suchen in der Mitte des gesamten Pfades aufeinandertreffen, wenn der Graph symmetrisch ist. In *Abbildung 2.18* ist der konzeptionelle Unterschied zwischen bi- und unidirektionalen Suchen visualisiert.

Es stellt sich die Frage, welche Vorteile diese bidirektionale Art der Erkundung bietet, sogar ohne Multithreading einzusetzen. Diese kommen erst richtig zum Vorschein, wenn man den Algorithmus auf Graphen anwendet, bei denen die Anzahl erkundeter Knoten exponentiell wächst, wenn man auf ihnen eine Breitensuche ausführt. Ein Beispiel für solch einen Graphen ist der des 15-Puzzles, welches in *Kapitel 2.4.2* näher erklärt wird. Dort ist die Anzahl der Knoten bei einer Erkundungstiefe von n für kleine n annähernd 4^n , wobei die tatsächliche Anzahl 4^n aber nie überschreitet. Sucht man von nur einer Seite aus, so müsste man für einen Endknoten, der nur 20 Schritte entfernt liegt, schon ca. $4^{20} \approx 10^{12}$ Knoten erkunden. Sucht man nun aber vom Start- zum

Endknoten und umgekehrt, so muss jede der beiden Suchen nur mehr ca. $4^{10} \approx 10^6$ Knoten erkunden, also insgesamt ca. $2 \cdot 4^{10} = 2^{21} \approx 2 \cdot 10^6$. Generell kann man also sagen, dass die bidirektionale Bestensuche die Anzahl der zu erkundenden Knoten von m auf ungefähr $2\sqrt{m}$, bzw. von k^n auf ungefähr $2k^{n/2}$ reduziert, wobei k die Basis der Exponentialfunktion k^n ist, die die Entwicklung der Anzahl der besuchten Knoten beschreibt, wenn man auf einem Graphen eine Breitensuche mit einer maximalen Tiefe von n durchführt.

Ein Nachteil dieser dualen Suche ist, dass gerichtete Kanten für die Rückwärtssuche umgedreht werden müssen, da ansonsten Pfade gefunden werden könnten, die vom Start- zum Zielknoten gar nicht traversierbar sind. Auf unendlichen Graphen ist es jedoch oft unmöglich, jeden Knoten zu finden, der von sich ausgehend eine Kante zu einem bestimmten anderen Knoten hat. Dieses Problem tritt allerdings ausschließlich bei Digraphen auf, da gerichtete Kanten nur von einer Seite aus traversiert werden können, ungerichtete aber von beiden. Somit kennt man bei ungerichteten Graphen, wenn man alle Nachbarn eines bestimmten Knoten kennt, automatisch alle Knoten, die ebenfalls eine von sich ausgehende Kante zu diesem haben.

Eine Implementierung der bidirektionalen Bestensuche ist in *Listing 2.23* zu sehen. Wichtig anzumerken ist hierbei, dass diese Implementierung nicht die Optimalität gefundener Pfade garantiert. Genau deswegen implementiert `BidiBestFirstSearch` das Interface `SearchAlgorithm` anstatt `PathfindingAlgorithm`. Algorithmen wie BHPA⁹ oder BHFFA¹⁰ bzw. BHFFA2¹¹ sind dazu in der Lage, kürzeste Pfade zu finden, sind hingegen um einiges komplexer. Natürlich könnten andere Pathfinding-Algorithmen als nur Bestensuchen ebenso für bidirektionale Suchen verwendet werden, jedoch kann mit diesen nicht garantiert werden, dass nur optimale Pfade gefunden werden. Das liegt daran, dass ausschließlich Bestensuchen die dafür benötigte Kostenfunktion $g(n)$ besitzen, die immer die aktuell geringste bekannte Distanz zum Startknoten zurückgibt. [EZ:Web58, EZ:Web62, EZ:Web63, EZ:Web64]

⁹Bidirectional Heuristic Path Algorithm

¹⁰Bidirectional Heuristic Front-to-Front Algorithm

¹¹BHFFA Again

```
1  public record BidiBestFirstSearch<T>(BestFirstSearch<T> forwardSearch,
2                                         BestFirstSearch<T> backwardSearch)
3   implements SearchAlgorithm<T> {
4
5     public BidiBestFirstSearch {
6       if (Objects.requireNonNull(forwardSearch) == Objects.requireNonNull(
7           backwardSearch)) {
8         throw new IllegalArgumentException(
9             "The forward and backward searches " +
10            "must not be the same instance.");
11      }
12    }
13
14   /**
15    * Constructor to create a bidirectional best-first search using A*
16    * with the given heuristic as the search algorithm for both directions
17    */
18   public BidiBestFirstSearch(Heuristic<T> heuristic) {
19     this(new AStar<>(heuristic), new AStar<>(heuristic));
20   }
21
22   @Override
23   public int getVisitedVertexCount() {
24     return forwardSearch.getVisitedVertexCount()
25       + backwardSearch.getVisitedVertexCount();
26   }
27
28   @Override
29   public List<T> findAnyPath(T start,
30                             EndCondition<T> forwardEndCondition,
31                             Graph<T> graph) {
32     T end = forwardEndCondition.vertex()
33       .orElseThrow(() -> new IllegalArgumentException(
34           "The end condition must specify a vertex."));
35   }
36
37   var backwardEndCondition = EndCondition.endAt(start);
38   forwardSearch.initializeDataStructures(start);
39   backwardSearch.initializeDataStructures(end);
40
41   while (nextUnvisited()) {
42     forwardSearch.closeCurrent();
43     backwardSearch.closeCurrent();
44
45     if (forwardSearch.hasVisited(backwardSearch.getCurrent())) {
46       return mergePaths(start, backwardSearch.getCurrent(), end);
47     }
48
49     if (backwardSearch.hasVisited(forwardSearch.getCurrent())) {
50       return mergePaths(start, forwardSearch.getCurrent(), end);
51     }
52   }
53 }
```

```

51     }
52
53     forwardSearch.expand(forwardEndCondition, graph);
54     backwardSearch.expand(backwardEndCondition, graph);
55 }
56
57 return Collections.emptyList();
58 }

59
60 private boolean nextUnvisited() {
61     return forwardSearch.nextUnvisited()
62         && backwardSearch.nextUnvisited();
63 }

64 /**
65 * Merges the vertices from the start to the common vertex
66 * and from the end to the common vertex into a single path.
67 * The common vertex is included only once in the result.
68 *
69 * @return the merged path
70 */
71 private List<T> mergePaths(T start,
72                             T common,
73                             T end) {
74     var forwardPredecessors = forwardSearch.getPredecessors();
75     var forwardTracer = new PathTracer<>(forwardPredecessors);
76     var startToCommon = forwardTracer.unsafeTrace(start, common);

77     var backwardPredecessors = backwardSearch.getPredecessors();
78     var backwardTracer = new PathTracer<>(backwardPredecessors);
79     var endToCommon = backwardTracer.unsafeTrace(end, common);
80     return mergePaths(startToCommon, endToCommon);
81 }
82

83 /**
84 * Merges the 2 paths by reversing the 2nd one,
85 * removing the common vertex from it and concatenating them
86 *
87 * @param startToCommon the path from the start to the common vertex
88 * @param endToCommon   the path from the end to the common vertex
89 * @return the merged path
90 */
91 private List<T> mergePaths(List<T> startToCommon, List<T> endToCommon)
92 {
93     var commonToEnd = endToCommon.reversed();
94     startToCommon.addAll(commonToEnd.subList(1, commonToEnd.size()));
95     return startToCommon;
96 }
97
98 }
99

```

Listing 2.23: BidiBestFirstSearch.java

2.4 Vergleich und Evaluation der Algorithmen

2.4.1 Hilfsklassen

Pathfinder

Damit man nicht bei jeder Pfadberechnung denselben Graphen und Algorithmus als Parameter übergeben muss, gibt es die generische Klasse `Pathfinder<T>`. Sie besitzt zwei Instanzvariablen, den `Graph<T> graph` und den `PathfindingAlgorithm<T> algorithm`. Die Implementierung folgt somit dem Strategy-Pattern. Will man die Methoden `findAnyPath` oder `findShortestPath` am an den Konstruktor von `Pathfinder` übergebenen Graphen anwenden, müssen jetzt nur mehr Start- und Zielknoten übergeben werden, da immer der gespeicherte Graph verwendet wird. Sowohl der `Graph <T> graph` als auch der `PathfindingAlgorithm<T> algorithm` haben Getter und Setter, beide Variablen können also im Nachhinein noch verändert werden, falls nötig. [EZ:Web65]

```

1  @AllArgsConstructor
2  @Getter
3  @Setter
4  public class Pathfinder<T> {
5
6      private Graph<T> graph;
7      private PathfindingAlgorithm<T> algorithm;
8
9      public List<T> findAnyPath(T start,
10          EndCondition<T> endCondition) {
11          return algorithm.findAnyPath(start, endCondition, graph);
12      }
13
14      public List<T> findShortestPath(T start,
15          EndCondition<T> endCondition) {
16          return algorithm.findShortestPath(start, endCondition, graph);
17      }
18  }

```

Listing 2.24: Pathfinder.java

Da die Klasse `Pathfinder<T>` generisch ist, reicht es aus, den Typ `T` nur ein einziges Mal zu spezifizieren, in diesem Fall bei der Erzeugung des Graphen, *siehe Listing 2.25*. Der Typ `T` des `Pathfinder<T>` kann hier durch Typinferenz bestimmt werden. Dasselbe gilt für den `PathfindingAlgorithm<T>`, hier `BreadthFirstSearch<T>`, *siehe Kapitel 2.3.3*, da er erst bei der Übergabe der Konstruktorparameter erzeugt wird. Als Beispiel wird hier der Graph aus *Abbildung 2.19* verwendet. Das Print-Statement am

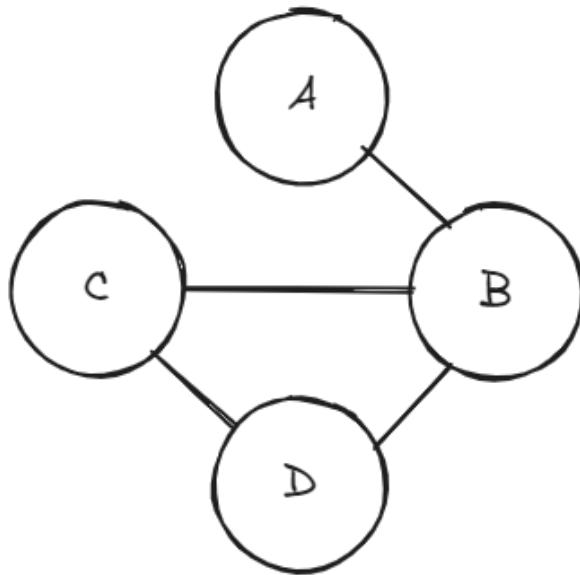


Abbildung 2.19: Graph mit zwei unterschiedlich langen Pfaden von A zu D

Ende gibt wie erwartet den kürzesten Pfad von A zu D aus: [A, B, D].

```
1 var graph = new FlexibleGraph<Character>();  
2 graph.addEdge('A', 'B');  
3 graph.addEdge('B', 'C');  
4 graph.addEdge('C', 'D');  
5 graph.addEdge('B', 'D');  
6  
7 var finder = new Pathfinder<>(graph, new BreadthFirstSearch<>());  
8 System.out.println(finder.findShortestPath('A', EndCondition.endAt('D')));
```

Listing 2.25: Typinferenz

2.4.2 Benchmarking

Mithilfe der Java-Library *OpenCSV* werden die für die folgenden Benchmarks relevanten Daten in eine CSV¹²-Datei geschrieben und in Python mit der Library *pandas* in einen `DataFrame` konvertiert. Anschließend werden Teile davon mittels *seaborn* visualisiert.

ModifiableGraph

Die für *Abbildung 2.20* und *Abbildung 2.21* verwendeten Daten wurden aufgezeichnet, indem die `findShortestPath`-Methode von jedem der Algorithmen BFS, Dijkstra und A* und die `findAnyPath`-Methode von Bidi A* auf 1000 verschiedenen Graphen mit jeweils 1000 Knoten und durchschnittlich 1000 zufälligen Kanten auf einer *Intel® Core™ i5-8400 CPU @ 2.80 GHz* ausgeführt wurde. Es wird für Bidi A* `findAnyPath` aufgerufen, da dieser Algorithmus nur diese Methode für die Pfadberechnung besitzt, da er nicht garantieren kann, dass seine gefundenen Pfade die kürzesten sind. Da die Tiefensuche nicht für die Berechnung kürzester Pfade geeignet ist und somit eine Ewigkeit braucht, um zu terminieren, kommt diese nicht in diesem Benchmark vor. Die rekursive Variante ist noch einmal um Größenordnungen langsamer, daher wird diese erst recht nicht miteinbezogen.

In *Abbildung 2.20* sind die Korrelationen zwischen den verschiedenen aufgezeichneten Spalten in einer Heatmap visualisiert. Anstatt der defaultmäßigen Pearson- werden hier die Spearman-Korrelationskoeffizienten berechnet, da die Kriterien für das Erkennen von Korrelationen so um einiges lockerer sind. Diese müssen bei Spearman nicht linear sein, sondern nur monoton. Da kein in dieser Arbeit beschriebener Pathfinding-Algorithmus eine lineare Zeitkomplexität hat, ist das von großem Vorteil. Die Spalten haben folgende Bedeutungen:

- **Path Present:** Ob ein Pfad gefunden wurde, oder nicht
- **Path Length:** Die Anzahl der Knoten im Pfad, der gefunden wurde
- **Path Weight:** Die Summe der Gewichtungen der Kanten, die der gefundene Pfad überquert
- **Duration (µs):** Die Zeit in Mikrosekunden, die es gedauert hat, bis der Algorithmus den Pfad gefunden hat

¹²Comma-separated values

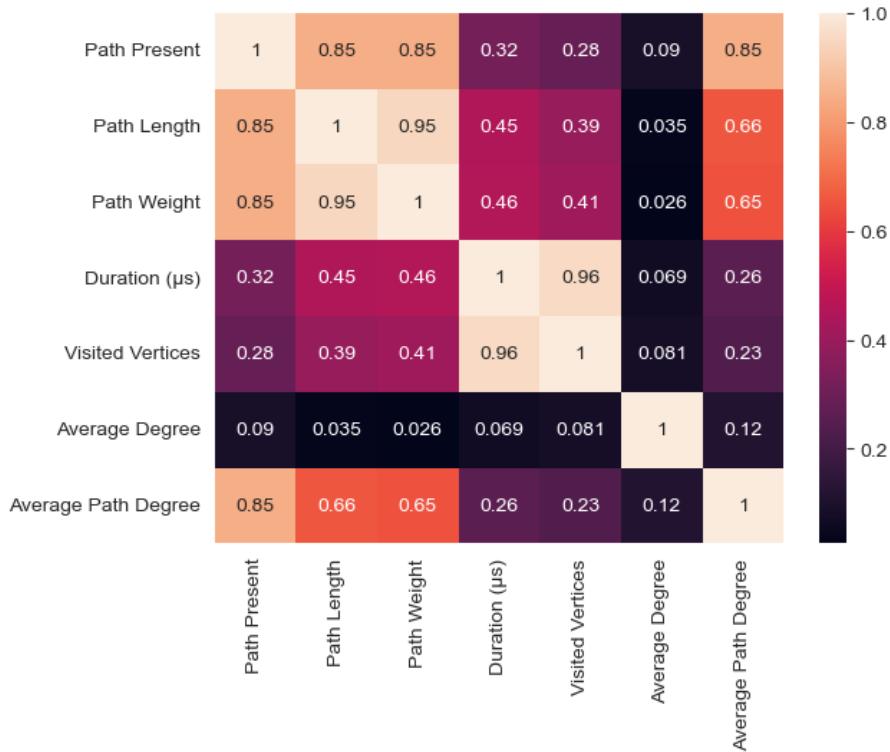


Abbildung 2.20: Die Korrelationen zwischen den Spalten

- **Visited Vertices:** Die Anzahl an Knoten, die vom jeweiligen Algorithmus besucht wurden, um den Pfad zu finden
- **Average Degree:** Die durchschnittliche Nachbaranzahl jedes Knotens des Graphen, in dem der Algorithmus ausgeführt wurde
- **Average Path Degree:** Die durchschnittliche Nachbaranzahl jedes Knotens des Pfades, der gefunden wurde

Man kann aus der Grafik einige interessante Zusammenhänge erkennen. Zum Beispiel korreliert die Zeitdauer, bis ein Pfad gefunden wurde, mit einem Koeffizienten von 0.96 extrem stark mit der Anzahl an besuchten Knoten. Daraus folgt, dass das Traversieren von Knoten den größten Teil der Zeit einnimmt, die ein Algorithmus benötigt, um einen Pfad zu berechnen. Die nahezu lineare Korrelation dieser zwei Spalten ist in *Abbildung 2.21(e)* klar erkennbar. Der Pearson-Korrelationskoeffizient wäre hier beispielsweise nur 0.43.

Auf den ersten Blick könnte man denken, dass der Zeitaufwand von Bidi A* am schlechtesten ist, da dieser dort am schnellsten ansteigt, wenn sich die Anzahl besuchter Knoten erhöht. Durch *Abbildung 2.21(f)* wird jedoch klar, dass genau das Gegenteil

der Fall ist. Für diesen Plot wurden nämlich die Anzahlen besuchter Knoten jedes Algorithmus durch die höchste Anzahl an besuchten Knoten des jeweiligen Algorithmus dividiert. Somit kann man besser erkennen, welcher Algorithmus wie lange gebraucht hat, um einen gewissen Anteil seiner höchsten besuchten Knotenanzahl zu erkunden. Der Zeitaufwand von Bidi A* steigt also zwar am schnellsten, allerdings besucht dieser Algorithmus im Durchschnitt viel weniger Knoten, als andere Algorithmen.

Aus *Abbildung 2.21(a)* geht hervor, dass BFS davon ausgeht, dass die Gewichtung jeder Kante 1 beträgt. Somit optimiert BFS nicht die Gewichtung des Pfades, sondern die Anzahl der Knoten, die er beinhaltet. Dieser Fakt wird in *Abbildung 2.21(b)* nochmals verdeutlicht, da die durchschnittliche Gewichtung von Pfaden, die von BFS gefunden wurden, größer ist. Die kleinen, schwarzen Balken an der Spitze der großen sind Fehlerbalken und stellen das 95%-Konfidenzintervall dar.

Außerdem sind Länge und Gewichtung von Pfaden, die von Bidi A* gefunden wurden, im Durchschnitt größer, als von A* und Dijkstra gefundene. Dieser Fakt hängt allerdings stark von der Implementierung der bidirektionalen Suche ab. Die hier verwendete findet zwar immer einen vergleichsweise kurzen Pfad, jedoch nicht unbedingt den kürzesten.

In *Abbildung 2.21(c)* wird der Zeitaufwand von Algorithmen gegenübergestellt, wenn ein Pfad gefunden vs. nicht gefunden wurde. Bidi A* merkt eindeutig am frühesten, wenn zwei Knoten nicht durch einen Pfad miteinander verbunden sind. Dieser Algorithmus ist im Durchschnitt auch am schnellsten, wenn ein Pfad zwischen den Knoten existiert. Hierbei ist es aber nochmals wichtig anzumerken, dass die verwendete Implementierung von Bidi A* nicht immer den kürzesten Pfad findet. Wäre der Algorithmus dazu in der Lage, würde er etwas schlechter, aber wahrscheinlich trotzdem noch am besten abschneiden.

Abbildung 2.21(d) zeigt den Zusammenhang zwischen den Längen der gefundenen Pfade und der durchschnittlichen Anzahl besuchter Knoten. Der Bereich rund um die Linien stellt hier wieder das 95%-Konfidenzintervall dar. Man kann sehen, dass Bidi A* im Durchschnitt die wenigsten Knoten besuchen muss und die Anzahl besuchter Knoten am langsamsten ansteigt, wenn sich die Pfadlänge erhöht.

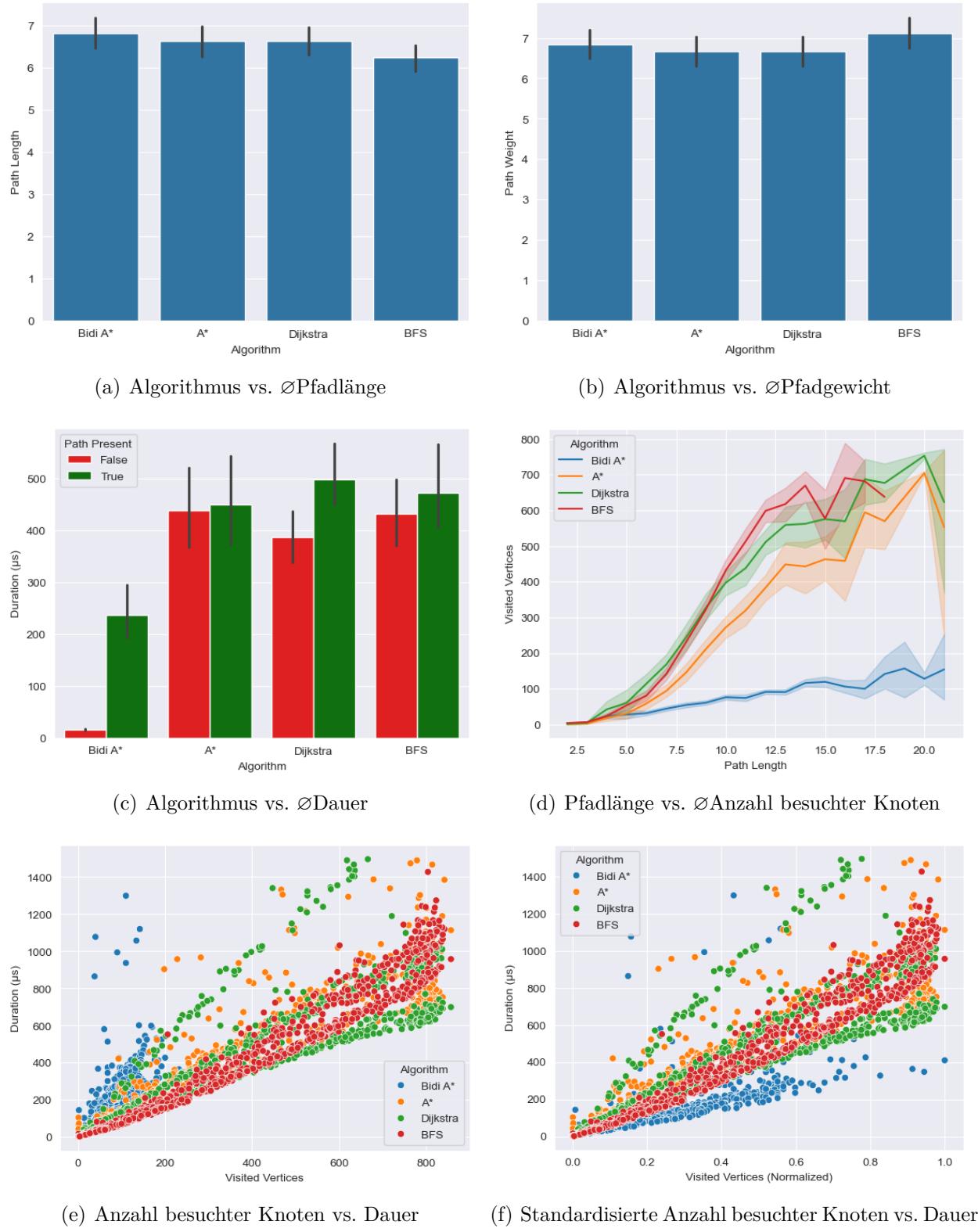
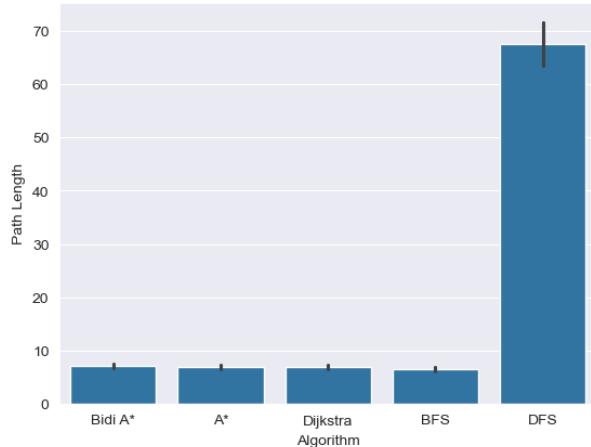


Abbildung 2.21: Vergleich der iterativen Algorithmen exklusive DFS

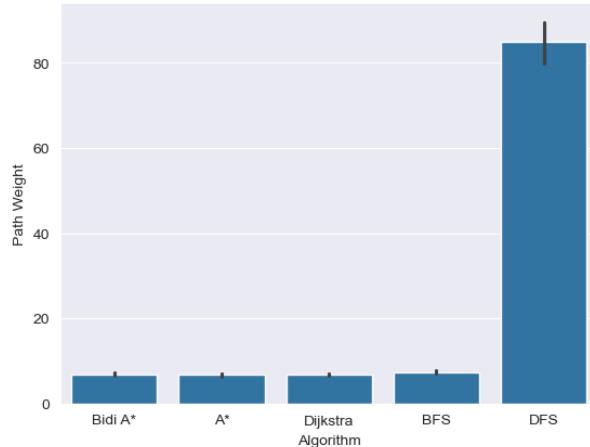
Im Benchmark der Methode `findAnyPath` ist die iterative Variante der Tiefensuche inkludiert. Sogar für Finden irgendeines Pfades auf nur zehn zufälligen Graphen benötigt die rekursive Variante mehr als acht Stunden, während alle anderen Algorithmen, sogar die iterative Tiefensuche, für dieselbe Aufgabe auf über 1000 Graphen nur wenige Millisekunden benötigen. Als CPU wurde dieselbe verwendet, wie für den `findShortestPath`-Benchmark. Die Anzahlen an Graphen und Knoten sowie die durchschnittliche Anzahl an Kanten pro Graph sind ebenfalls gleich, wie beim vorherigen Benchmark.

Abbildung 2.22(a) und *Abbildung 2.22(b)* sind kaum unterscheidbar, da der Balken von DFS so hoch ist, dass alle anderen an Detail verlieren. Durch *Abbildung 2.22(c)* und *Abbildung 2.22(f)* sieht man aber, dass DFS trotzdem fast dieselbe Performance aufweist, wie alle anderen Algorithmen. Eine Ausnahme ist hier jedoch Bidi A*. Dieser Algorithmus ist performancemäßig so gut wie nicht zu übertreffen.

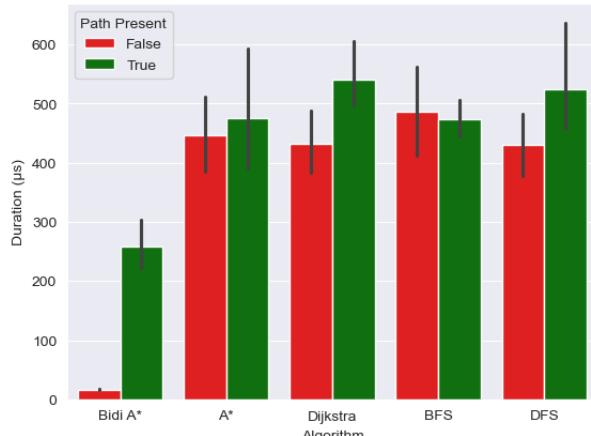
Es sieht in *Abbildung 2.22(d)* so aus, als würde die Anzahl der besuchten Knoten von DFS gegen ungefähr 400 konvergieren, wenn die Pfadlänge gegen ∞ strebt. Dieser Grenzwert ist abhängig von der Kantenwahrscheinlichkeit und den Knotenzahlungen in den verwendeten Graphen, sofern beide dieser Kennzahlen konstant sind.



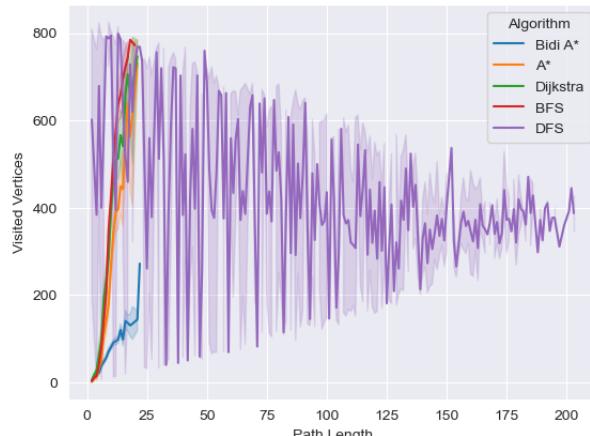
(a) Algorithmus vs. ØPfadlänge



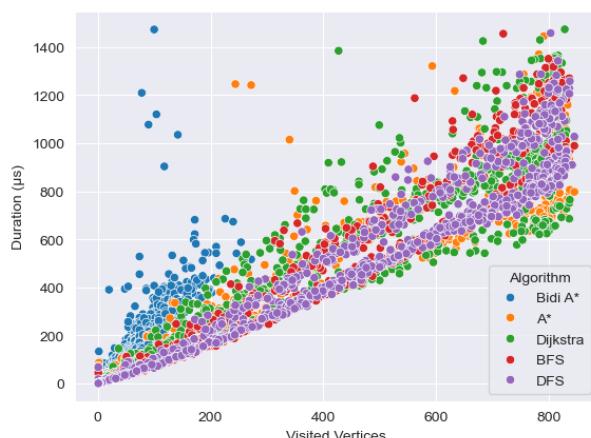
(b) Algorithmus vs. ØPfadgewicht



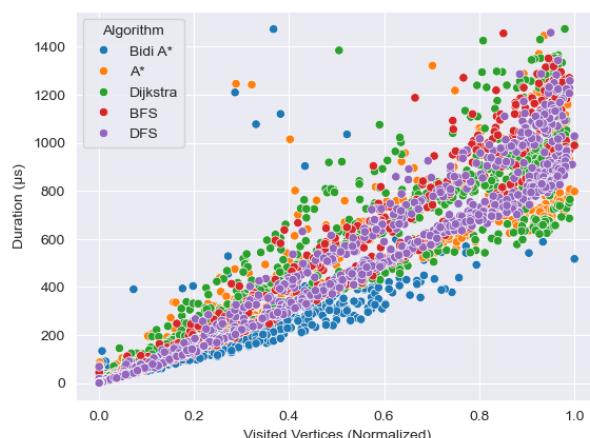
(c) Algorithmus vs. ØDauer



(d) Algorithmus vs. ØAnzahl besuchter Knoten



(e) Anzahl besuchter Knoten vs. Dauer



(f) Standardisierte Anzahl besuchter Knoten vs. Dauer

Abbildung 2.22: Vergleich der iterativen Algorithmen inklusive DFS

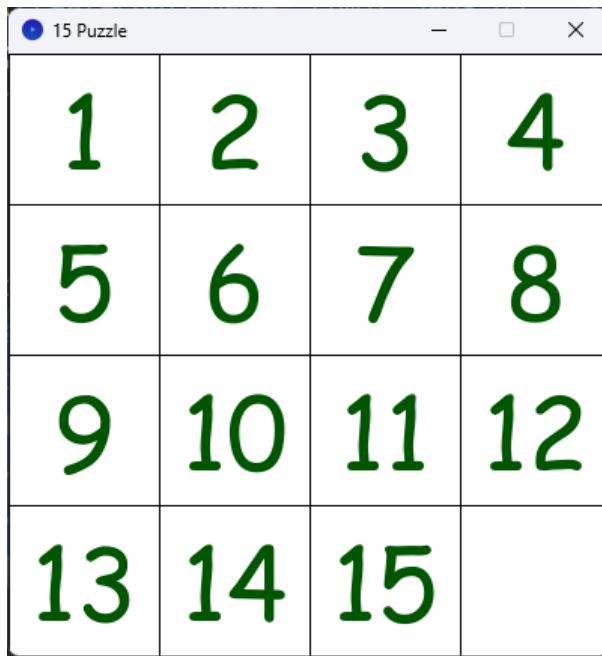


Abbildung 2.23: Der gelöste Zustand des 15-Puzzles

15-Puzzle

Das 15-Puzzle ist ein klassisches Beispiel für kontraintuitive, dennoch nützliche Anwendungsfälle von Pathfinding. Es besteht aus einem Raster, welches aus vier Spalten und vier Zeilen besteht. Auf diesem sind 15 nummerierte Tiles platziert, und um das Puzzle zu lösen, müssen diese wie in *Abbildung 2.23* von 1 bis 15 aufsteigend angeordnet werden, wenn man von oben nach unten zeilenweise von links nach rechts liest.

Tiles können nur verschoben werden, indem sie ihren Platz mit dem leeren Tile tauschen. Dadurch ergeben sich $16! = 2.0922789888 \cdot 10^{13}$ verschiedene mögliche Zustände. Von diesen mehr als 20 Billionen Zuständen sind jedoch nur exakt die Hälfte lösbar, also $\frac{16!}{2} = 1.0461394944 \cdot 10^{13}$. Einer von diesen ist in *Abbildung 2.24* zu sehen.

Interpretiert man nun jeden lösbarren Zustand als Knoten eines Graphen, so wird klar, dass seine Speicherung nicht so einfach funktionieren kann. Nimmt man an, dass jeder Knoten nur 32 Bit an Speicherplatz benötigt, so bräuchte man schon $32 \cdot \frac{16!}{2}$ Bit = $3.34764638208 \cdot 10^{14}$ Bit = $4.1845579776 \cdot 10^{13}$ Byte \approx 41 Terabyte, um diesen Graphen auf einmal im RAM zu halten, was schlachtweg unmöglich ist. Zieht man nun zusätzlich in Betracht, dass jedes der 16 Tiles (15 nummerierte + 1 leeres) als 32-Bit `int` dargestellt wird, so erhöht sich der Speicherverbrauch nochmal auf das 16-fache, also $16 \cdot 41.845579776$ Terabyte \approx 670 Terabyte, und das, ohne Objektreferenzen miteinzubeziehen. Wegen Fällen wie solchen ist `Graph` ein Interface: damit die `getNeighbors-`

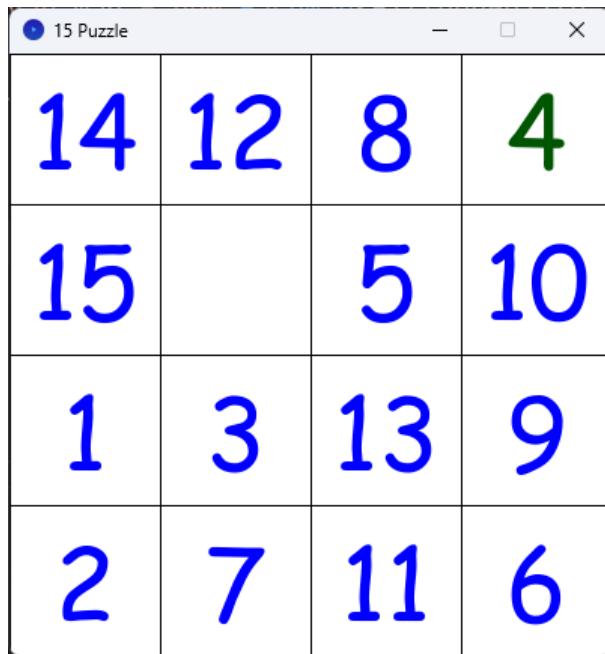


Abbildung 2.24: Ein ungelöster Zustand des 15-Puzzles

Methode beliebig implementiert werden kann. Für das 15-Puzzle existiert eine eigene `FifteenPuzzleGraph`-Klasse, die die Methode so implementiert, dass alle Zustände zurückgegeben werden, die von dem übergebenen Zustand aus innerhalb eines Zuges erreicht werden können, *siehe Listing 2.26*. So muss der Graph nie als Ganzes abgespeichert werden, dafür müssen seine Nachbarn bei jedem `getNeighbors`-Aufruf neu berechnet werden. `Direction` ist hierbei ein Enum mit den 4 Werten `UP`, `DOWN`, `LEFT` und `RIGHT`.

```
1 public class FifteenPuzzleGraph implements Graph<FifteenPuzzle> {
2
3     @Override
4     public Map<FifteenPuzzle, Double> getNeighbors(FifteenPuzzle puzzle) {
5         var map = new HashMap<FifteenPuzzle, Double>();
6
7         for (var direction : Direction.values()) {
8             var boardCopy = new FifteenPuzzleBoard(puzzle.board());
9             boolean moved = boardCopy.move(direction);
10
11            if (moved) {
12                var puzzleCopy = new FifteenPuzzle(boardCopy);
13                map.put(puzzleCopy, 1.0);
14            }
15        }
16
17        return map;
18    }
19
20    @Override
21    public double getEdgeWeight(FifteenPuzzle source,
22                                FifteenPuzzle destination) {
23        return 1;
24    }
25
26    @Override
27    public double sumEdgeWeights(List<FifteenPuzzle> path) {
28        return (path.isEmpty()) ? 0 : path.size() - 1;
29    }
30}
31}
```

Listing 2.26: Repräsentation des 15-Puzzles als Graph

Die abstrakte Klasse `MemoizedGraph` aus *Listing 2.27*, welche `Graph` implementiert, sorgt dafür, dass bereits berechnete Nachbarn eines Knotens nicht bei jedem Zugriff erneut berechnet werden müssen, sondern memoisiert bzw. zwischengespeichert werden, um Rechenzeit zu ersparen. Da die Berechnung der Nachbarzustände beim 15-Puzzle performancemäßig jedoch ziemlich billig ist, hat Memoization hier sogar einen negativen Einfluss auf die Performance, weshalb `FifteenPuzzleGraph` direkt `Graph` implementiert, ohne einen Vererbungs-Umweg über `MemoizedGraph` zu nehmen. [EZ:Web59]

```

1  public abstract class MemoizedGraph<T> implements Graph<T> {
2
3      private final Map<T, Map<T, Double>> adjacencyCache = new HashMap<>();
4
5      @Override
6      public Map<T, Double> getNeighbors(T vertex) {
7          return adjacencyCache.computeIfAbsent(
8              vertex,
9              this::calculateNeighbors
10         );
11     }
12
13     protected abstract Map<T, Double> calculateNeighbors(T vertex);
14
15 }
```

Listing 2.27: `MemoizedGraph.java`

Bei den Benchmarks für das 15-Puzzle wird die Zeit nicht mehr in Mikrosekunden, sondern in Sekunden gemessen. Das liegt daran, dass das Finden von möglichst kurzen Lösungswegen beim 15-Puzzle eine äußerst ressourcenintensive Aufgabe ist. Als einzige Algorithmen kommen für dieses Problem A* und Bidi A* infrage, da alle anderen letztendlich in einem `OutOfMemoryError` resultieren würden. Sogar bei A* tritt dieses Problem in den meisten Fällen auf, da schlichtweg zu viele Knoten besucht und gespeichert werden müssen. Aus diesem Grund kommt hier ausschließlich Bidi A* zum Einsatz. Als Heuristik wird zuerst für jedes der 15 Tiles die minimale Anzahl an Zügen berechnet, die benötigt werden, um dieses in die gewünschte Position zu bringen. Die minimale Anzahl an Zügen entspricht hier ganz einfach der Manhattan-Distanz von der aktuellen zur gewollten Position. Zum Schluss werden diese 15 Zahlen aufsummiert. Der Code, der genau das tut, ist in *Listing 2.28* zu sehen. Da hier die *minimale* Anzahl an Zügen berechnet wird, ist diese Heuristik zulässig, da sie die tatsächliche Anzahl an Zügen nie überschätzt.

```

1 public int getLeastMoveCountTo(FifteenPuzzle desired) {
2     return getLeastMoveCountsTo(desired)
3         .values()
4         .stream()
5         .mapToInt(Integer::intValue)
6         .sum();
7 }
8
9 public Map<Integer, Integer> getLeastMoveCountsTo(FifteenPuzzle desired) {
10    var moves = HashMap.<Integer, Integer>newHashMap(board.area() - 1);
11    var positions = getPositions();
12    var desiredPositions = desired.getPositions();
13
14    for (int i = 1; i < board.area(); i++) {
15        var position = positions.get(i);
16        var desiredPosition = desiredPositions.get(i);
17        int moveCount = position.manhattanDistance(desiredPosition);
18        moves.put(i, moveCount);
19    }
20
21    return moves;
22 }
```

Listing 2.28: Berechnung der minimalen Anzahl an benötigten Zügen

Für die Heuristik des Puzzles werden diese Methoden in der das Interface `Heuristic` implementierenden Klasse `FifteenPuzzleHeuristic` angewendet, um die Heuristik für ein übergebenes Puzzle bei gegebenem Endzustand zu evaluieren.

```

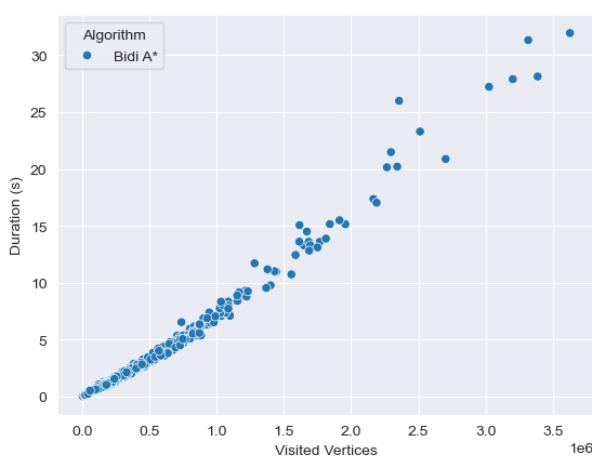
1 public class FifteenPuzzleHeuristic implements Heuristic<FifteenPuzzle> {
2
3     @Override
4     public double applyAsDouble(FifteenPuzzle puzzle,
5                                 EndCondition<FifteenPuzzle> endCondition) {
6         var vertex = endCondition.vertex().orElseThrow();
7         return puzzle.getLeastMoveCountTo(vertex);
8     }
9
10 }
```

Listing 2.29: FifteenPuzzleHeuristic.java

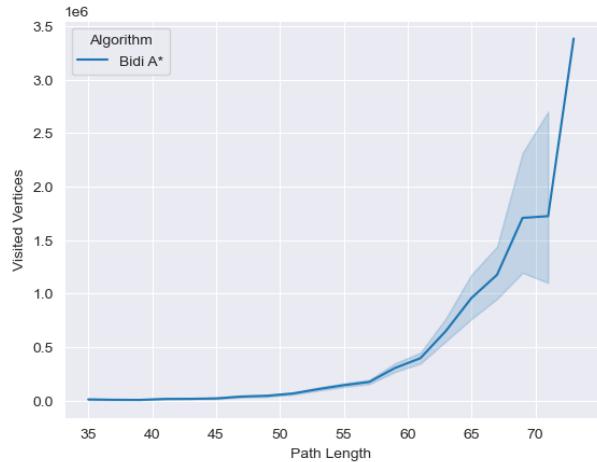
Für das Benchmarking wurde Bidi A* mit der soeben beschriebenen Heuristik auf 1000 zufälligen Anfangszuständen des 15-Puzzles ausgeführt. Es wurde dieselbe CPU verwendet, wie für die vorherigen Benchmarks.

In Abbildung 2.25(a) kann man deutlich erkennen, dass der Zusammenhang zwischen Anzahl besuchter Knoten und der Zeitdauer nahezu linear ist. Der Korrelationskoeffizient nach Spearman ist hier größer als 0.996 und sogar der nach Bravais-Pearson liegt über 0.99. Fast alle Dauern befinden sich im Intervall [0; 30], wobei die überwiegende Mehrheit unter 10 liegt.

Weiters interessant ist auch, dass man in Abbildung 2.25(b) klar den exponentiell ansteigenden Verlauf der Kurve erkennen kann, welche die durchschnittliche Anzahl besuchter Knoten für jede vorkommende Pfadlänge beschreibt. Der Grund dafür ist in Kapitel 2.3.6 näher erklärt. Kurz gefasst ist der Grund dafür die annähernd exponentiell ansteigende Anzahl an Knoten im Graphen des 15-Puzzles, da es für die meisten Konfigurationen des Puzzles vier Nachbarn gibt, die wiederum vier Nachbarn haben (wobei nur drei davon neu sind, deshalb *annähernd* exponentiell) und so weiter.



(a) Anzahl besuchter Knoten vs. Dauer



(b) Pfadlänge vs. Anzahl besuchter Knoten

Abbildung 2.25: Ergebnisse von Bidi A* am 15-Puzzle

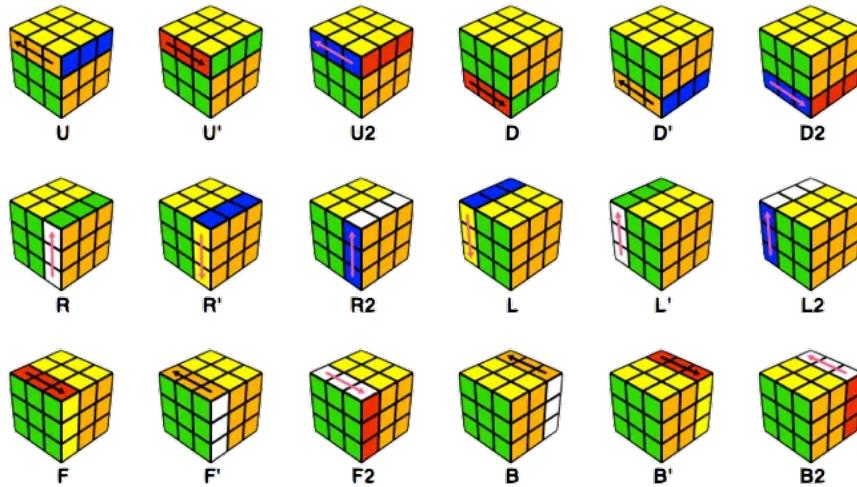


Abbildung 2.26: Die 18 möglichen Züge beim Rubik's Cube
[EZ:Web61]

Rubik's Cube

Ein ähnliches Problem, welches sich ebenfalls mit Pathfinding lösen lässt, ist der Zauberwürfel bzw. Rubik's Cube. Bei diesem kann ebenfalls jeder lösbarer Zustand als Knoten eines Graphen interpretiert und das Rätsel auf eine ähnliche Weise gelöst werden. Beim Rubik's Cube gibt es im Gegensatz zu den maximal 4 Nachbarzuständen des 15-Puzzles immer 18, siehe Abbildung 2.26. Das macht das Problem zu einem viel komplexeren. Zusätzlich beträgt die Anzahl seiner möglichen Zustände nicht nur mehr als 10^{13} , sondern über $4 \cdot 10^{19}$. Da für die Lösung dieses Problems weitaus mehr Konzepte eingesetzt werden müssen, als nur Pathfinding, gibt es in dieser Arbeit keine Benchmarks für Pathfinding auf Rubik's Cubes.

2.4.3 Zusammenfassung der Ergebnisse

Tabelle 2.7 gibt eine Übersicht über die Stärken und Schwächen der beschriebenen Algorithmen. Es gibt keinen Algorithmus, der in allem gut ist. Jeder hat seine eigenen spezifischen Anwendungsfälle, für die er am besten geeignet ist. Deshalb ist auch bei jedem der genannten Algorithmen die *bedingte* Optimalität ein Vorteil. Obwohl jeder hier aufgezählte Algorithmus diesen gemeinsamen Vorteil hat, heißt es noch lange nicht, dass alle Pathfinding-Algorithmen optimal sein können. Es gibt sehr wohl welche, die niemals optimal sind, außer durch Zufall.

Zusammenfassend kann man sagen, dass für die Findung des kürzesten Pfades A* die beste Wahl ist, wenn eine zulässige Heuristik verfügbar ist, z.B. wenn die Kantengewichte die geografische Distanz zwischen den Knoten darstellen. Ist der Einsatz einer zulässigen Heuristik nicht möglich, ist vermutlich Dijkstra die nächstbeste Wahl.

Will man bloß irgendeinen Pfad zwischen zwei Knoten finden, ist Bidi A* wohl am besten. Findet man keine Bibliothek die Bidi A* bereits implementiert hat und will man sich nicht die Mühe machen, den Algorithmus von Grund auf neu zu implementieren, da die Implementierung etwas komplexer ist, als die von anderen Algorithmen, so kann auch die Tiefensuche eingesetzt werden.

| Algorithmus | Vorteile | Nachteile |
|-------------|--|---|
| BFS | + Bedingt optimal + Simple Implementierung | – Nur dann optimal, wenn Graph ungewichtet oder alle Gewichtungen gleich – Ineffizient |
| DFS | + Bedingt optimal + Simple Implementierung + Kann schnell und effizient einen Pfad zwischen zwei Knoten finden | – Nur dann optimal, wenn Implementierung geeignet – Kann nur langsam und ineffizient den kürzesten Pfad zwischen zwei Knoten finden |
| Dijkstra | + Bedingt optimal | – Nur dann optimal, wenn keine Gewichtungen traversierter Kanten negativ – Ineffizient |
| A* | + Bedingt optimal + Sehr effizient | – Nur dann optimal, wenn Heuristik zulässig und keine Gewichtungen traversierter Kanten negativ |
| BidiBeFS | + Bedingt optimal + Sehr effizient + Hohe Skalierbarkeit | – Nur dann optimal, wenn Heuristik zulässig und Implementierung geeignet – Komplexe Implementierung – Adjazzenzen müssen für die Rückwärtssuche invertiert werden |

Tabelle 2.7: Die Vor- und Nachteile der verschiedenen Pathfinding-Algorithmen



Abbildung 2.27: Das von LiCAR verwendete Modellauto

2.5 Projektbezug

Das Projekt *LiCAR* hat die Aufgabe, das Modellauto aus *Abbildung 2.27* selbstständig zwischen zwei Punkten navigieren zu lassen. Dazu muss zuerst ein Agent in einer Testumgebung trainiert werden. Bestenfalls in einer simulierten, damit keinerlei Schäden angerichtet werden können. Diese simulierte Testumgebung wird LiCAR von *CARLA*, einer Verkehrs-Simulationssoftware, zur Verfügung gestellt. Für die effiziente Navigation dieses Agents ist Pathfinding zwingend notwendig. Über die Python-API von CARLA kann man die Topologie einer Map erhalten. Aus dieser kann mithilfe von NetworkX ein Graph erzeugt werden, siehe *Listing 2.30*. In der echten Welt könnte dieser mithilfe einer zweidimensionalen Punktwolke erstellt werden, die direkt aus dem am Dach des Modellautos montierten LiDAR-Sensor ausgelesen wird. Die rohen LiDAR-Daten können dann z.B. mittels SLAM¹³ zu einem Graphen konvertiert werden, auf dem sowohl in dieser Arbeit genannte als auch andere Pathfinding-Algorithmen angewandt werden können.

```
1 import carla
2 import math
3 import networkx as nx
4
5 client = carla.Client('localhost', 2000)
6 world = client.get_world()
7 topology = world.get_map().get_topology()
8
9 graph = nx.Graph()
10 graph.add_edges_from(topology)
```

Listing 2.30: Erzeugung eines Graphen aus einer CARLA-Map

¹³Simultaneous Localization and Mapping

NetworkX bietet auch eine Menge an vorgefertigten Methoden für Pfadberechnungen, wobei die für LiCAR mit Abstand wichtigste `shortest_path` ist. In *Listing 2.31* kann man sehen, wie diese Methode in der Praxis angewandt werden könnte. Ähnlicher Code wird intern von LiCAR verwendet.

```

1 nodes = list(graph)
2 start = nodes[0]
3 end = nodes[-1]
4
5 path = nx.shortest_path(
6     graph,
7     source=start,
8     target=end,
9     weight=lambda u, v, attributes: math.hypot(
10         u.transform.location.x - v.transform.location.x,
11         u.transform.location.y - v.transform.location.y
12     )
13     method='dijkstra'
14 )

```

Listing 2.31: Berechnung des kürzesten Pfades mittels NetworkX

Standardmäßig verwendet `shortest_path` den Dijkstra-Algorithmus, der optionale Parameter `method` kann aber anstatt auf `'dijkstra'` auch auf `'bellman-ford'` gesetzt werden, damit der Bellman-Ford-Algorithmus benutzt wird. Der Hauptvorteil von Bellman-Ford gegenüber Dijkstra besteht darin, dass dieser mit negativen Kantengewichtungen umgehen kann, er ist aber dafür zeitaufwendiger. Die Lambda-Expression, auf die `weight` gesetzt wird, berechnet die euklidische Distanz zwischen zwei CARLA-Waypoints. Das ist notwendig, da `add_edges_from` keine Gewichtungen setzt und ansonsten davon ausgegangen wird, dass jede Kante eine Gewichtung von 1 besitzt. Man bekommt die drei Parameter `u`, `v` und `attributes` übergeben, wobei `u` und `v` den Anfangs- bzw. Endknoten der Kante darstellen, deren Gewichtung berechnet werden soll. Dabei ist `attributes` ein `dict`, welches Attribute der Kante enthält.

Kapitel 3

Model Based Reinforcement Learning

3.1 Artificial Intelligence, AI

In diesem Kapitel wird eine Übersicht über das Thema Künstliche Intelligenz (Artificial Intelligence, AI) gegeben. In diesem Rahmen wird auf die Grundlagen und verschiedenen Anwendungsbereiche der AI eingegangen.

Durch Machine Learning und AI können Fahrzeuge ihre Umgebung in Echtzeit erkennen und darauf reagieren. AI kann potenzielle Probleme frühzeitig erkennen und diese vermeiden. Im Wesentlichen verbessert AI im autonomen Fahren Sicherheit, Effizienz und das gesamte Fahrerlebnis.

3.1.1 Einführung in AI

Im Kern zielt AI darauf ab, menschliche Intelligenz in Maschinen zu replizieren, um ihnen die Ausführung von Aufgaben zu ermöglichen, die in der Regel menschliche Kognition erfordern. Dieses Bestreben wird dadurch angetrieben, Systeme zu schaffen, die in der Lage sind, ähnlich wie Menschen, zu lernen, zu schlussfolgern, Probleme zu lösen und Entscheidungen zu treffen. AI umfasst eine reiche Vielfalt von Technologien, Methoden und Algorithmen, die Maschinen mit menschenähnlichen Fähigkeiten ausstatten.

Grundlagen der Künstlichen Intelligenz

Die Funktion einer AI ist es, menschliches Handeln so gut wie möglich nachzuahmen und dieses zu optimieren. Um dies zu ermöglichen, muss man die Verfügung von einigen Funktionen sicherstellen. Eine AI, welche Handlungen wie eine Person oder besser ausführen können soll, muss die folgenden Vier Fähigkeiten beherrschen.

Menschliches Handeln

In einem von Alan Turing erfundenen Prozess, der die operative Intelligenz misst, werden von einer Maschine einige schriftliche Fragen beantwortet. Wenn ein Mensch nicht erkennen kann, ob diese von einem Menschen oder Computer beantwortet wurden, besteht die Maschine den sogenannten "Turing-Test". Um solch einen Test zu bestehen, muss eine Maschine über die folgenden Fähigkeiten verfügen:

- natürliche Sprachverarbeitung, um erfolgreich in einer vorgegebenen Sprache zu kommunizieren;
- Wissensrepräsentation, um das zu speichern, was sie weiß oder hört;
- automatisches Schlussfolgern, um die gespeicherten Informationen zu verwenden, um Fragen zu beantworten und neue Schlussfolgerungen zu ziehen;
- maschinelles Lernen, um sich an neue Umstände anzupassen und Muster zu erkennen und zu extrapolieren.

Obwohl der Turing-Test Intelligenz prüft, ist keine visuelle oder physische Interaktion zwischen dem Menschen und der Maschine erforderlich. Der sogenannte "total Turing-Test" beinhaltet auch ein Video Signal, welches die Wahrnehmungsfähigkeit der Maschine überprüft. Um solch einen Test zu bestehen sind weitere Funktionen notwendig:

[MM:Book01]

Menschliches Denkvermögen

Um einer AI beizubringen, so zu Handeln und zu Denken wie ein Mensch, muss es zuerst möglich sein Faktoren und Kriterien zu bestimmen um mit deren Hilfe zu überprüfen, wie erfolgreich die AI ihr erwünschtes Ziel erreicht. Menschliches Denkvermögen

ist jedoch schwer zu messen und zu überprüfen. Doch durch kognitive Wissenschaften kann der Denkprozess von verschiedenen AI Models mit denen von Menschen verglichen werden. Man kann mit der Hilfe von psychologischen Tests die Ähnlichkeit der Verhalten messen, indem man das Verhalten einer Person in einem bestimmten Umfeld observiert und einer AI dasselbe Simulierte Umfeld oder die Entscheidung gibt.

Rationales Denken

Das rationale Denken ist wesentlich einfacher zu überprüfen als Faktoren, welche spezifischer auf Menschen zutreffen. Rationales Denken kann durch Logik und Ethik überprüft werden und wurde erstmals von griechischen Philosophen eingeführt. Um Logik und Rationalität zu überprüfen, muss der AI Agent den Zustand in dem es ist, und den vorherigen Zustand vergleichen. Dies nennt man einen "Rational Agent". Ein solcher Agent kann zum Beispiel auf ein selbst fahrendes Auto angewendet werden, welcher den momentanen Zustand optimiert, in dem er Sensordaten und seine Position im Umfeld Vergleicht und dadurch verbessert.

Rationales handeln

All die oben angeführten Punkte sind für rationales Handeln erforderlich. Um auf den "Rational Agent" zurück zu kommen, kann man diesen als Beispiel nehmen. Wenn ein AI Modell den Turing Test, welcher für Rationales Handeln ein elementarer Baustein ist, besteht, sind menschliches Handeln und Denken schon einigermaßen vorhanden. Diese Fähigkeiten ermöglichen es einem Agenten schon einigermaßen gute und nachvollziehbare Entscheidungen zu treffen. Ein Rational Agent trifft bestimmte Entscheidungen Anhand von Überprüfungen des Umfelds und der eigenen Aktionen. Dies passt mir der Hilfe von mathematischen Funktionen und Regeln, welche helfen sollten intelligentes Verhalten zu erzeugen.

- Computer Vision dient dazu, Objekte wahrzunehmen,
- Robotik dient dazu, Objekte zu manipulieren sich zu bewegen.

[MM:Book03]

Technische Grundlagen der Künstlichen Intelligenz

Um ein AI-System funktionsfähig zu machen, sind mehrere Schlüsselkomponenten erforderlich. Dazu gehören:

- Agent: Der Agent ist das zentrale Element eines AI-Systems. Er ist die Entität, die Entscheidungen trifft und Handlungen ausführt. Je nach Anwendung kann der Agent ein Computerprogramm, ein Roboter, oder eine andere Form von AI-Entität sein. Der Agent interagiert mit seiner Umgebung und versucht, Aufgaben zu lösen oder Ziele zu erreichen.
- Umgebung (Enviroment): Die Umgebung representiert, das Umfeld in dem der Agent agieren kann. Dieser erhält Informationen über den Zustand der Umgebung, um dann mithilfe von einer Reward Function Entscheidungen treffen zu können.
- Wahrnehmung (Observation): Die Fähigkeit des Agenten, die Umgebung wahrzunehmen, ist von entscheidender Bedeutung. Dies umfasst die Verarbeitung und Interpretation von Daten der Sensoren, um den Zustand der Umgebung zu verstehen. Hier kommen Technologien wie Computer Vision und Spracherkennung ins Spiel.
- Aktionen (Actions): Aktionen sind die Entscheidungen, die der Agent treffen kann, um seine Ziele zu erreichen. Die Auswahl der richtigen Aktionen ist entscheidend für den Erfolg des AI-Systems.
- Lernalgorithmen (Learning Algorithms): Lernalgorithmen sind die mathematischen Verfahren, die es dem Agenten ermöglichen, aus Erfahrung und Daten zu lernen. Diese Algorithmen analysieren Feedback aus der Umgebung (Enviroment), passen die Handlungen (Actions) des Agenten an und verbessern seine Leistung im Laufe der Zeit. Beispiele für Lernalgorithmen sind neuronale Netzwerke, Q-Learning und Deep Reinforcement Learning.
- Training: Beim Trainingsprozess werden gesammelte Daten von Umgebungen ähnlich jener, in der der Agent dann eingesetzt werden soll verwendet, um dem Agenten die richtigen Handlungen in verschiedenen Situationen beizubringen. Dieser Prozess kann sowohl überwacht (supervised) oder unüberwacht (unsupervised) sein.
- Daten-Quelle (Data-Source): Daten spielen eine elementare Rolle in der Entwicklung von einer AI. AI-Systeme basieren auf Daten, aus denen sie lernen. Dazu ist ein kontinuierlicher Datenfluss erforderlich, welcher von einem Sensor, von Bildern, Videos oder einer vorgefertigten Datenbank stammen kann.

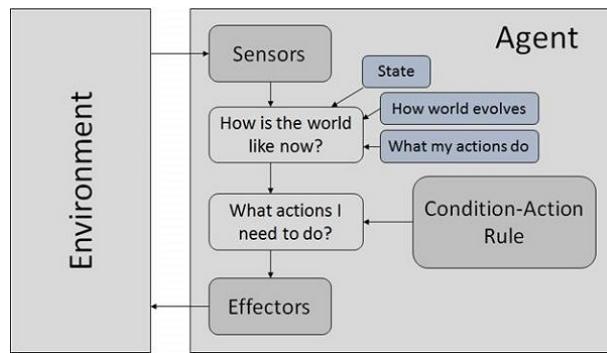


Abbildung 3.1: Agent interagiert mit Umgebung
[MM:Web01]

Wichtige AI-Konzepte und Terminologie

AI Konzepte können verwendet werden, um ein AI Modell auf einen bestimmten Nutzen zu spezialisieren. Denn für Modelle, welche für bestimmte Bereiche eingesetzt werden sollen, müssen Optimierungen und andere Ansätze durchgeführt werden. Einige Konzepte und deren Nutzen werden folgend erklärt.

Für manche Konzepte werden einige Terminologien welche AI betreffen wichtig sein, welche zuvor definiert werden.

Begriffe:

- **Big Data:** Big Data bezieht sich auf umfangreiche und komplexe Datensätze, da für Machine Learning eine große Menge an Daten gebraucht wird, um bessere Ergebnisse zu bekommen.
- **Artificial Neurons:** Artificial Neurons sind die grundlegenden Bausteine von Neural Networks. Sie nehmen Eingabewerte, wenden Gewichtungen und Thresholds an und leiten das Ergebnis durch eine Activation Function, um eine Ausgabe zu erzeugen. Sie imitieren die Funktionsweise von biologischen Neuronen.

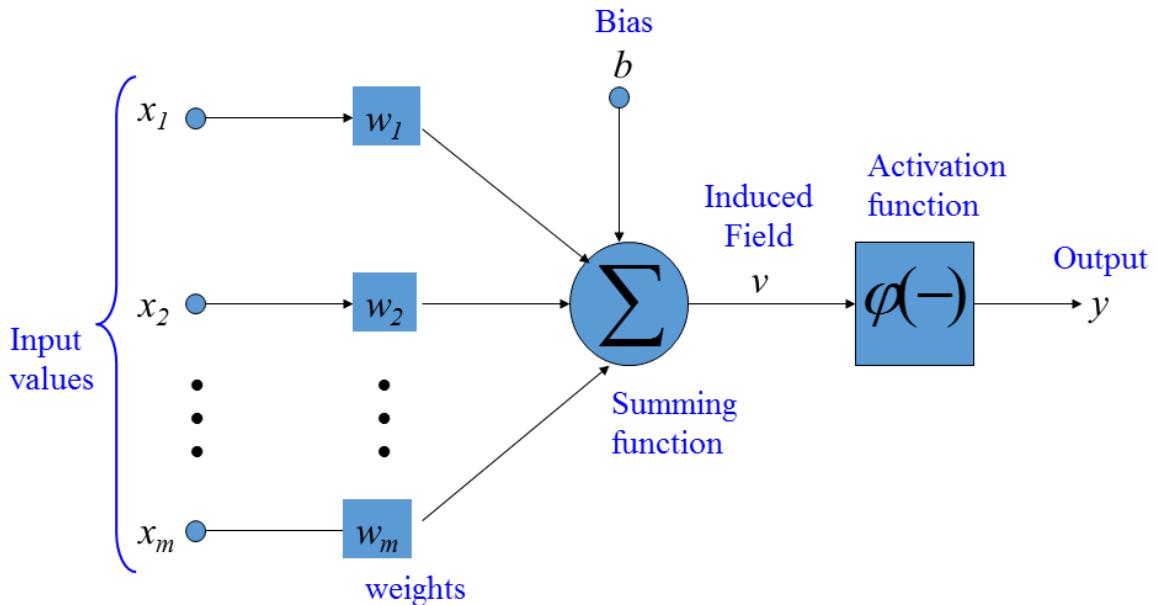


Abbildung 3.2: Artificial Neuron
[MM:Web08]

Wie in der Abbildung 3.2 zu sehen ist, hat ein Artificial Neuron, Inputs, welche auf Perceptrons genannt werden, jedes dieser Inputs hat eine Gewichtung welches Weight genannt wird.

Diese Weights werden mit den Inputs multipliziert und dann summiert:

$$\text{sum} = \sum_{i=0}^n x_i * w_i$$

auf diese Summe wird dann eine Activation Function angewandt, welche die Summe der Inputs mit den Weights in einen Output verwandelt. Auf diese Activation Functions wird noch später genauer eingegangen.

Dieser Output kann dann als:

$$y_k = \phi\left(\sum_{j=0}^m w_{kj}x_j\right)$$

ϕ ist dabei die Threshold Function

[MM:Web31]

- **Layers:** In einem Neural Network, wie in der Abbildung 3.3, beziehen sich Layers auf die verschiedenen Ebenen von miteinander verbundenen artificial Neurons. Es gibt Input-, Hidden- und Output Layers. Informationen fließen von dem Input

Layer über die Hidden Layers zum Output Layer, wo die Output Values ausgegeben werden.

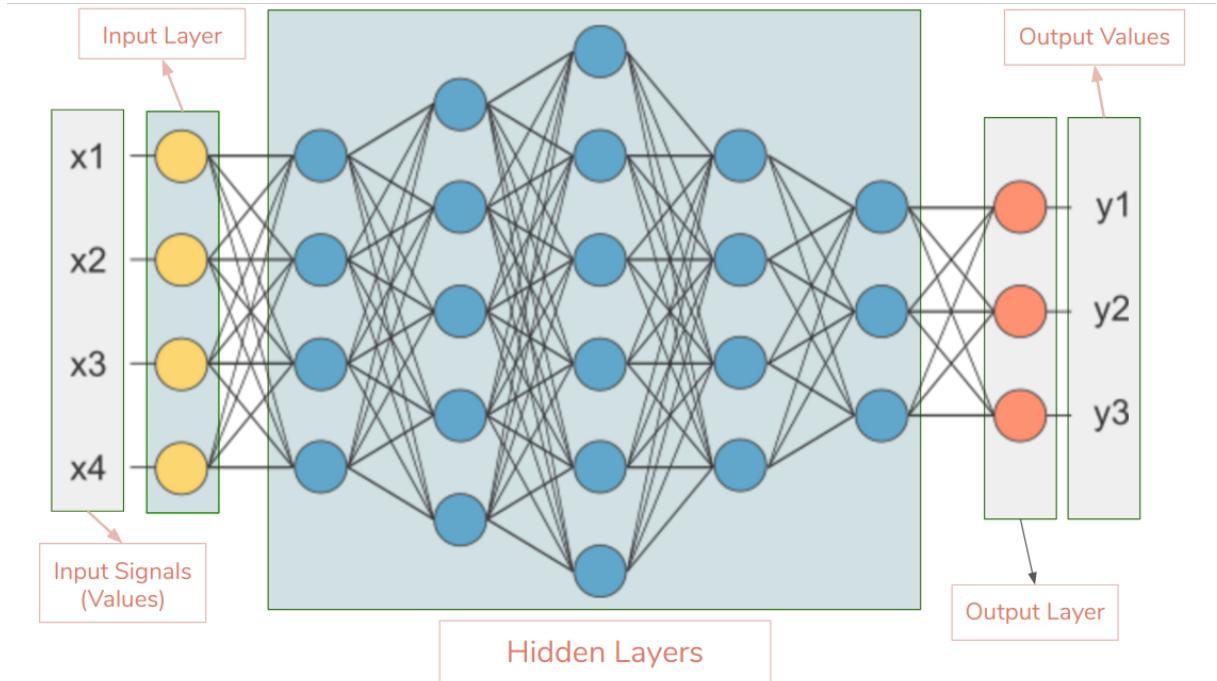


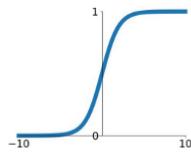
Abbildung 3.3: Layers
[MM:Web07]

- **Activation Function:** Activation Functions führen Nichtlinearität in Neural Networks ein. Dies ist besonders wichtig, weil durch lineare Funktionen, nur lineare Zusammenhänge gelernt und dargestellt werden können. Sie bestimmen die Ausgabe eines Neurons basierend auf der gewichteten Summe seiner Eingaben und eines Thresholds. Gängige Activation Functions umfassen Sigmoid, ReLU (Rectified Linear Unit) und tanh. Der nächste Abschnitt fokussiert sich näher auf Activation Functions.

Activation Functions

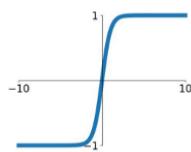
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



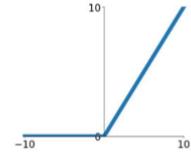
tanh

$$\tanh(x)$$



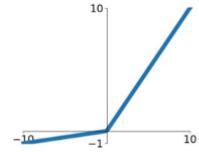
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$



Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

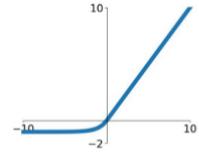


Abbildung 3.4: Activation Functions
[MM:Web09]

Activation Functions sind besonders wichtig um den Neural Networks das Lernen zu ermöglichen und bestimmte Probleme im ML zu lösen.

Diese Liste erklärt die Activation Function welche in 3.4 zu sehen sind.

1. **Sigmoid-Function:** Die Sigmoid-Function wird definiert als $\sigma(x) = \frac{1}{1+e^{-x}}$. Sie ordnet jeden Eingabewert einem Wert zwischen 0 und 1 zu, was sie nützlich für Modelle macht, bei denen wir die Wahrscheinlichkeit als Ausgabe vorhersagen müssen.
2. **Tanh-Function:** Die hyperbolische Tangensfunktion, oder tanh, wird definiert als $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$. Sie gibt Werte im Bereich von -1 bis 1 aus, was sie um Null zentriert und in einigen Fällen der Sigmoid-Funktion vorzuziehen ist.
3. **ReLU-Function:** Die Rectified Linear Unit (ReLU) Funktion wird definiert als $f(x) = \max(0, x)$. Sie gibt den Eingabewert direkt aus, wenn er positiv ist; andernfalls wird Null ausgegeben. ReLU wird aufgrund seiner rechnerischen Effizienz und der Fähigkeit, eine schnellere Konvergenz zu ermöglichen, häufig verwendet.
4. **Leaky ReLU-Function:** Leaky ReLU ist eine Variante von ReLU, definiert als $f(x) = \max(\alpha x, x)$, wobei α eine kleine Konstante ist. Diese Funktion erlaubt einen kleinen, nicht-null Gradient, wenn der Eingabewert negativ ist.
5. **Maxout-Function:** Maxout ist nicht genau eine Activation Function, sondern eine Schicht, die die ReLU und ihre Varianten verallgemeinert. Sie wird

durch die Operation $f(x) = \max(w_1^T x + b_1, w_2^T x + b_2)$ definiert, wobei w_1, w_2 Gewichte und b_1, b_2 Verzerrungen sind. Es kombiniert mehrere lineare Teile, um eine nicht-lineare Funktion zu modellieren.

6. **ELU-Function:** Die Exponential Linear Unit (ELU) Funktion wird definiert als

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{if } x \leq 0 \end{cases}$$
. ELU zielt darauf ab, die Vorteile von ReLU und seinen Varianten zu kombinieren, indem es glattere Ausgaben für negative Werte bietet.

[MM:Web21]

- **Deep Neural Network:** Deep Neural Networks sind neuronale Netzwerke mit mehreren versteckten Schichten. Sie sind in der Lage, komplexe hierarchische Merkmale zu erlernen und bilden die Grundlage des Deep Learning.
- **Deep Learning Frameworks:** Deep Learning Frameworks sind Software-Libraries, die Tools und Schnittstellen zur Erstellung und Schulung von Deep Learning-Modellen bereitstellen. Beispiele sind TensorFlow, PyTorch und Keras. Solche Frameworks werden auch später in dieser Diplomarbeit eingesetzt, um ein eigenes Model zu trainieren.
- **Data Transformation:** Data Transformation bezieht sich auf den Prozess der Umwandlung und Modifikation von Daten, um sie für die Analyse oder das Machine Learning geeignet zu machen. Dies kann das Skalieren, Normalisieren oder Codieren von Daten umfassen. Solche Data Transformations sind wichtig, um dem Model Daten in der richtigen Form zu übergeben, um effizientes Lernen zu ermöglichen.
- **Overfitting:** Overfitting tritt auf, wenn ein Machine Learning Model lernt, auf den Trainingsdaten gut abzuschneiden, aber Schwierigkeiten hat, auf neuen, ungesiehenen Daten zu generalisieren. Dies resultiert in der Regel daraus, dass das Modell Rauschen oder irrelevante Details in den Trainingsdaten lernt.
- **Generalization:** Generalization ist die Fähigkeit eines Machine Learning Models, genaue Vorhersagen auf neuen, ungesiehenen Daten zu treffen, basierend auf dem, was es aus den Trainingsdaten gelernt hat.
- **Validation Data:** Validation Data ist ein separater Datensatz, der während des Trainings zur Bewertung der Leistung des Models verwendet wird. Dies hilft bei der Überwachung von Overfitting und der Anpassung von Hyperparametern.
- **Regularization:** Regularization wird im Machine Learning verwendet, um Overfitting zu verhindern, indem Strafen oder Einschränkungen für die Komplexität des Modells hinzugefügt werden, wie z.B. L1- und L2-Regularisierung.

- **Policies:** Im Reinforcement-Learning ist eine Policy eine Strategie, die das Verhalten eines Agenten definiert. Sie bestimmt die Aktion, die ein Agent in einem gegebenen Zustand ausführen sollte.
- **Markov Decision Process (MDP), (Markov-Entscheidungsprozess):** Ein MDP ist ein mathematischer Prozess, der im Reinforcement Learning zur Modellierung von Entscheidungsproblemen verwendet wird. Es besteht aus Zuständen, Aktionen, Belohnungen und Übergangswahrscheinlichkeiten.

Eine Folge von Zuständen ist "Markovisch", wenn die Wahrscheinlichkeit, in den nächsten Zustand S_{t+1} überzugehen, nur vom aktuellen Zustand S_t abhängt und nicht von den vorherigen Zuständen S_1, S_2, \dots, S_{t-1} . Das heißt für alle t gilt:

$$P[S_{t+1}|S_t] = P[S_{t+1}|S_1, S_2, \dots, S_t].$$

Im RL spricht man oft von einer "time-homogeneous Markov chain", bei dem die Übergangswahrscheinlichkeit unabhängig von t ist:

$$P[S_{t+1} = s_0|S_t = s] = P[S_t = s_0|S_{t-1} = s].$$

Formal betrachtet ist ein Markov Process (oder Markov-Chain) ein Tupel (S, P) , wobei:

- S eine (endliche) Menge von Zuständen ist,
- P eine State Transition Probability Matrix ist, $P_{ss_0} = P[S_{t+1} = s_0|S_t = s]$.

Die Dynamik des Markov Process verläuft wie folgt: Es beginnt in einem Zustand s_0 und bewegt sich auf einen Nachfolgezustand s_1 , der aus $P_{s_0s_1}$ gezogen wird, zu. Dann bewegt es sich zu s_2 , gezogen aus $P_{s_1s_2}$, und so weiter. Dies wird folgendermaßen dargestellt:

$$s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \dots$$

Fügen wir Rewards, Actions, Discounts zu einem Markov Process hinzu, erhalten wir einen Markov Decision Process. Formal betrachtet ist ein Markov Decision Process ein Tupel (S, A, P, γ, R) , wobei:

- S eine endliche Menge von Zuständen ist,
- A eine endliche Menge von Aktionen ist,
- P die State Transition Probability Matrix $P_{ss_0}^a = P[S_{t+1} = s_0|S_t = s, A_t = a]$, ist,
- $\gamma \in [0, 1]$ der Discount Factor ist,
- $R : S \times A \rightarrow \mathbb{R}$ eine Belohnungsfunktion ist.

[MM:Web11]

AI Konzepte:

- **Neural Networks:** Neural Networks können generative oder classifying sein, je nachdem was sie erreichen sollen. Aufgaben in denen zu unterscheiden (Classified) ist, verwenden meist Supervised Learning und Aufgaben in denen generiert werden soll z.B.: Bilder, Videos oder Sprache, wenden meist Unsupervised Learning an.

Neural Networks sind aus mehreren Layers zusammengesetzt. Einer Input Layer, ein oder mehreren hidden Layers und einer Output Layer, welche aus Nodes bestehen. Jede Node, oder auch Artificial Neuron genannt, ist mit einem anderen verbunden.

Jede Node hat sein eigenes Linear Regression Model, welches aus Input Data, Weights, einem Bias und einem Output besteht.

Die mathematische Schreibweise eines Neurons ist folgende:

$$a = \sum_{i=1}^n w_i x_i + bias = wx + bias \quad (3.1)$$

n gibt die Anzahl der Input Nodes an

w_i beschreibt den Wert des Weights

x_i stellt die Input Values dar

$bias$ ist der Bias des Neurons

Der Threshold ist entweder in einem Active oder Inactive state. Das hängt von dem Wert des Thresholds θ (Theta) ab, welcher mit der Funktion $f(x)$ für den Output bestimmt wird:

$$f(x) = \begin{cases} 1 & \text{if } a \geq \theta; \\ 0 & \text{if } a \leq \theta \end{cases}$$

Wenn der Output einer Node den Threshold überschreitet, dann wird dieser an die nächste Node übergeben.

[MM:Web02] [MM:Web10] [MM:Web11] [MM:Web23]

In einem Neural Network wird der Error berechnet um die Weights anzupassen, dies passiert bei der Backpropagation. Die mathematische Formel dieser zur Anpassung der Weights ist:

$$w_{t+1} = w_t - \underbrace{\eta}_{\text{learning rate}} \underbrace{\frac{dL}{dw_t}}_{\text{gradient}}$$

hierbei ist η die Learning Rate, w_{t+1} ist das Value vom nächsten Weight und w_t das value von dem jetzigen Weight, dann wird die Learning Rate mit der Ableitung des Losses und der Weights multipliziert, welcher auch Gradient genannt wird.

Man will den Error minimieren, deshalb verwendet man Gradient Descent und passt damit die weights an.

[MM:Web42]

Gradient Descent kann folgend als Funktion angeschrieben werden:

$$b = a - \gamma \nabla f(a)$$

wobei b die nächste Position ist, a ist die jetzige Position, $f(a)$ ist die Richtung des steilsten Descents und $a - \gamma$ die Minimierung des Gradient Descents bezeichnet. Diese Formel bestimmt also, in welche Richtung der steepest Descent für den Error ist.

Dafür wird die Learning Rate angepasst, um die Größe der Sprünge zu beeinflussen. Für Graphen wie in den Abbildungen 3.5 und 3.6 sind auf der X-Achse die Weights und auf der Y-Achse der Loss abgebildet. Um nun die Learning Rate zu optimieren und den Loss zu minimieren, wird die Ableitung der Kurve genommen. Wenn diese positiv ist, muss weiter nach links gegangen werden, ansonsten weiter nach rechts. Dies wird nun verwendet um bestenfalls ein globales Minimum ($f'(x) = 0$) zu finden wie der grüne Pfeil in Abbildung 3.6 anzeigt, zu finden, oder links davon ein lokales Minimum.

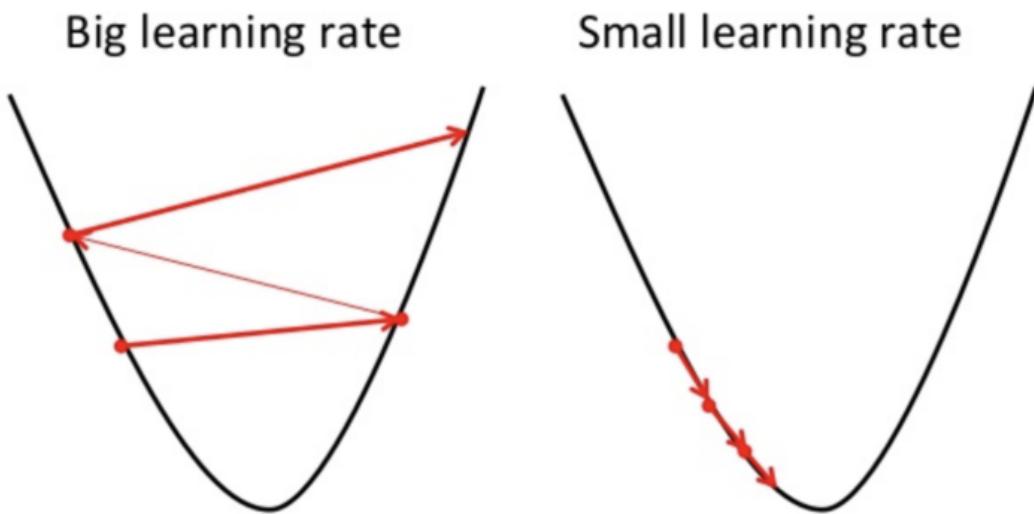


Abbildung 3.5: Gradient Descent
[MM:Web32]

Wie in der Abbildung 3.5 zu sehen ist, muss die learning rate, je nach den Kurven,

wie zum Beispiel in 3.6, dynamisch verändert werden, um das globale Minimum zu finden. Dies wird durch Optimizer erreicht. Ein sehr beliebter und guter Optimizer ist zum Beispiel der 'Adam' (Adaptive Movement Estimation) Optimizer, welcher auch später in einer AI Implementierung verwendet wird.

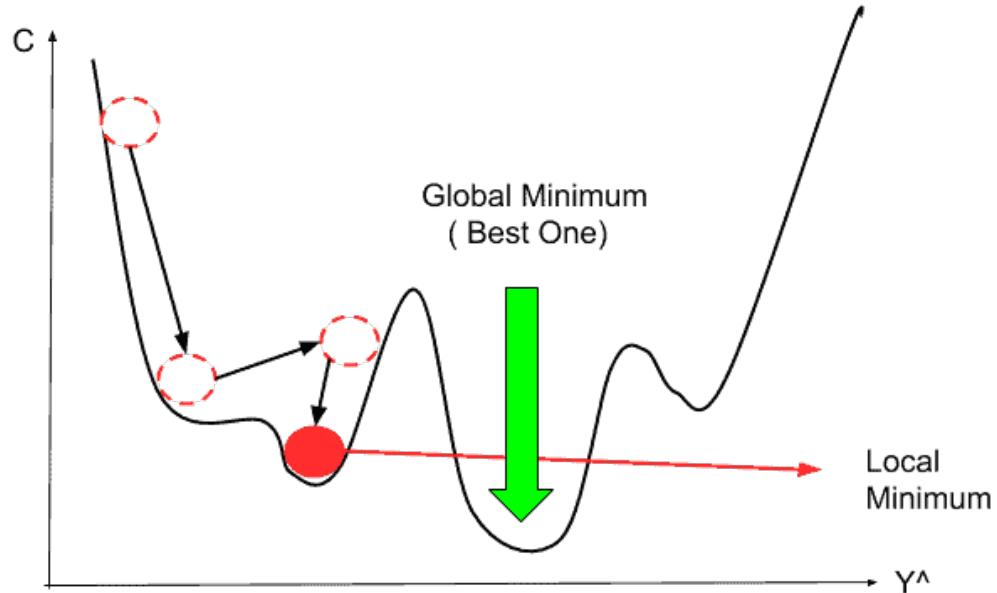


Abbildung 3.6: Gradient Descent Minimum
[MM:Web33]

[MM:Web32] [MM:Web33]

- **Machine Learning:** Machine Learning konzentriert sich darauf, Models zu entwickeln, die aus Erfahrung lernen und sich von selbst verbessern. Es ermöglicht Maschinen, Muster zu erkennen, Entscheidungen zu treffen und die Leistung auf Grundlage von Daten zu optimieren. Es gibt drei Haupttypen: Supervised Learning, Unsupervised Learning und Reinforcement Learning. Das Ziel besteht darin durch Generalization, Maschinen zu ermöglichen von Beispielen zu Lernen und Entscheidungen zu treffen, wenn neue Informationen vorliegen.

[MM:Web04]

- **Supervised Learning:** 3.8 Supervised Learning ist eine Machine Learning Methode, bei der die Eingabedaten so verarbeitet werden, dass neue Daten mit den erwünschten Ausgabedaten verglichen werden, was es dem Netzwerk ermöglicht, durch Generalization neue Daten, die es noch nie zuvor gesehen hat, auch richtig zu verwenden und zu schätzen. [MM:Web03]

Supervised Learning wird in zwei Kategorien aufgeteilt:

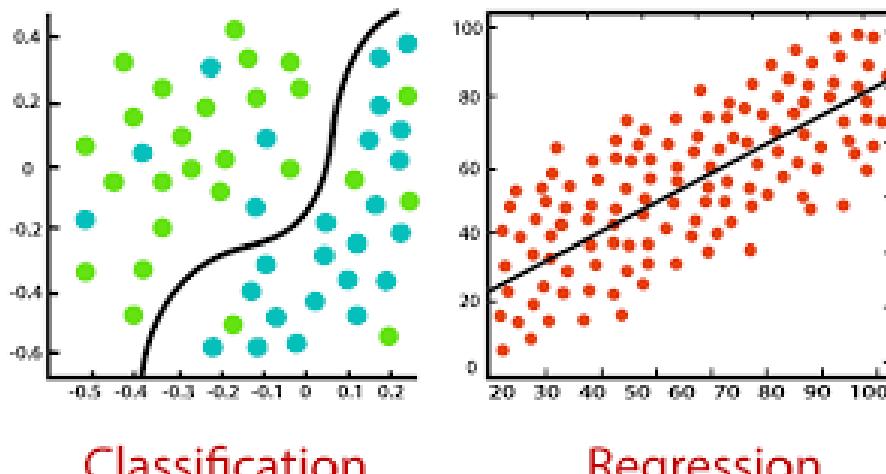


Abbildung 3.7: Classification and Regression
[MM:Web34]

[MM:Web35]

- **Classification:** Bei der Classification werden Eingabedaten vom Modell in Klassen oder Kategorien eingeteilt. Das Ziel besteht darin, eine Funktion zu erlernen, die eine klare Zuordnung zwischen den Eingabedaten und den entsprechenden Klassen ermöglicht.
- **Regression:** Im Gegensatz zur Klassifikation befasst sich die Regression mit der Vorhersage von kontinuierlichen Ausgabewerten. Das bedeutet, dass das Modell versucht, eine Funktion zu erlernen, die die Beziehung zwischen Eingabedaten und beschreibt. Dabei passt sich das Modell an die Daten an und versucht die beste Funktion zu erlernen, um Werte bestimmten Observations zuteilen zu können.

Den Unterschied zwischen Classification und Regression sieht man in der Abbildung 3.7 deutlich.

[MM:Web34]

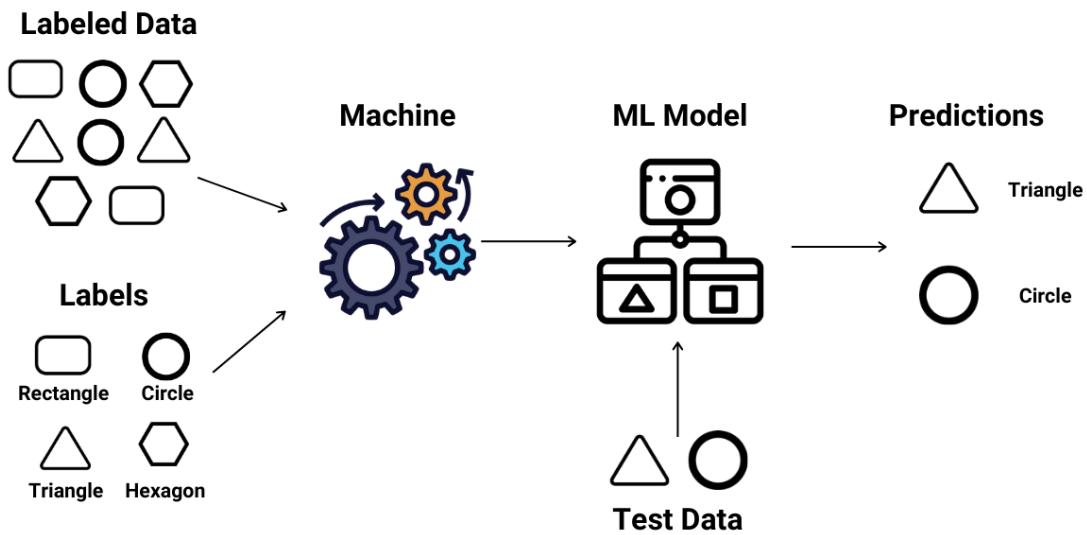


Abbildung 3.8: Supervised Learning
[MM:Web13]

- **Unsupervised Learning:** 3.10 Beim Unsupervised Learning wird versucht, Daten herzustellen und nachzuahmen. Diese Daten werden genutzt, um Fehler zu finden, die mit Gewichtungen im Netzwerk korrigiert werden können. Die Daten beim Unsupervised learning sind nicht gelabelt, das bedeutet, dass sie keine genauen Definitionen haben. Deshalb versucht das Netzwerk bestimmte Merkmale in den Daten zu erkennen, wie z.B.: Zusammenhänge, Folgen und Strukturen. [MM:Web02]

Hierfür kann Supervised Learning in drei Kategorien aufgeteilt werden:

- **Clustering:** beim Clustering werden Daten in Gruppen aufgeteilt, die ähnliche Merkmale besitzen. Dies gibt uns einen Enblick auf Muster, welche in Daten liegen.

Ein Beispiel für Clustering könnte wiefolg aussehen: 3.9

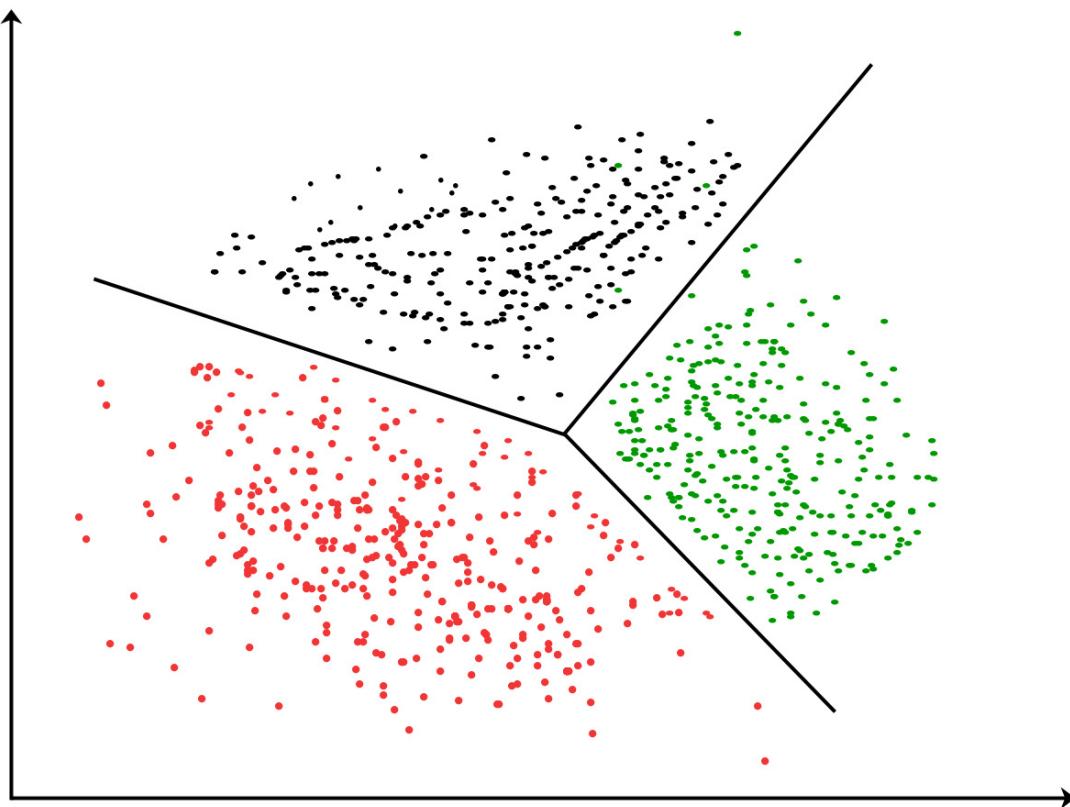


Abbildung 3.9: Clustering
[MM:Web39]

[MM:Web37]

- **Association:** Bei der Assoziation geht es darum, Muster, Zusammenhänge oder Assoziationen zwischen verschiedenen Variablen in den Daten zu identifizieren. Das Ziel ist es, Regeln oder Beziehungen zwischen den Variablen aufzudecken, wie zum Beispiel "Kunden, die Produkt A kaufen, kaufen auch oft Produkt B".

[MM:Web36]

- **Dimensional Reduction:** Die Dimensional Reduction zielt darauf ab, die Anzahl der Merkmale in Daten zu reduzieren und wichtige Informationen beizubehalten. Dies hilft, Redundanzen zu beseitigen, Rauschen zu reduzieren und die Effizienz von Machine-Learning-Algorithmen zu verbessern, indem die Komplexität des Modells verringert wird. Dies ist besonders nützlich wenn die Daten in höheren Dimensionen sind.

[MM:Web38]

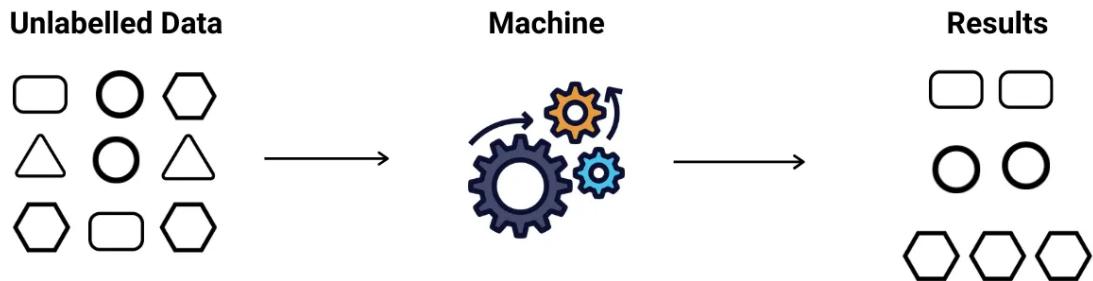


Abbildung 3.10: Unsupervised Learning
[MM:Web13]

[MM:Web35]

- **Deep Learning:** Das Deep Learning verwendet Deep Neural Networks welche es durch mehrere Layers ermöglichen zu lernen, komplexe Muster und Merkmale zu erkennen und erfassen. Deep Learning Algorithmen können auf Aufgaben mit Unsupervised Learning angewendet werden, da sie in der Lage sind, eigenständig Muster und Strukturen in nicht gelabelten Daten zu identifizieren und erlernen.
- **Reinforcement Learning:** Beim Reinforcement Learning handelt der Agent in einer Umgebung, um Belohnungen zu maximieren. Der Agent trifft Entscheidungen, lernt jedoch nicht aus gelabelten Datensätzen, sondern durch Erfahrung und Rückmeldungen in Form von Belohnungen oder Bestrafungen. Das Ziel ist es, eine optimale Strategie zu entwickeln, um langfristig die besten Belohnungen zu erhalten.

[MM:Book01]

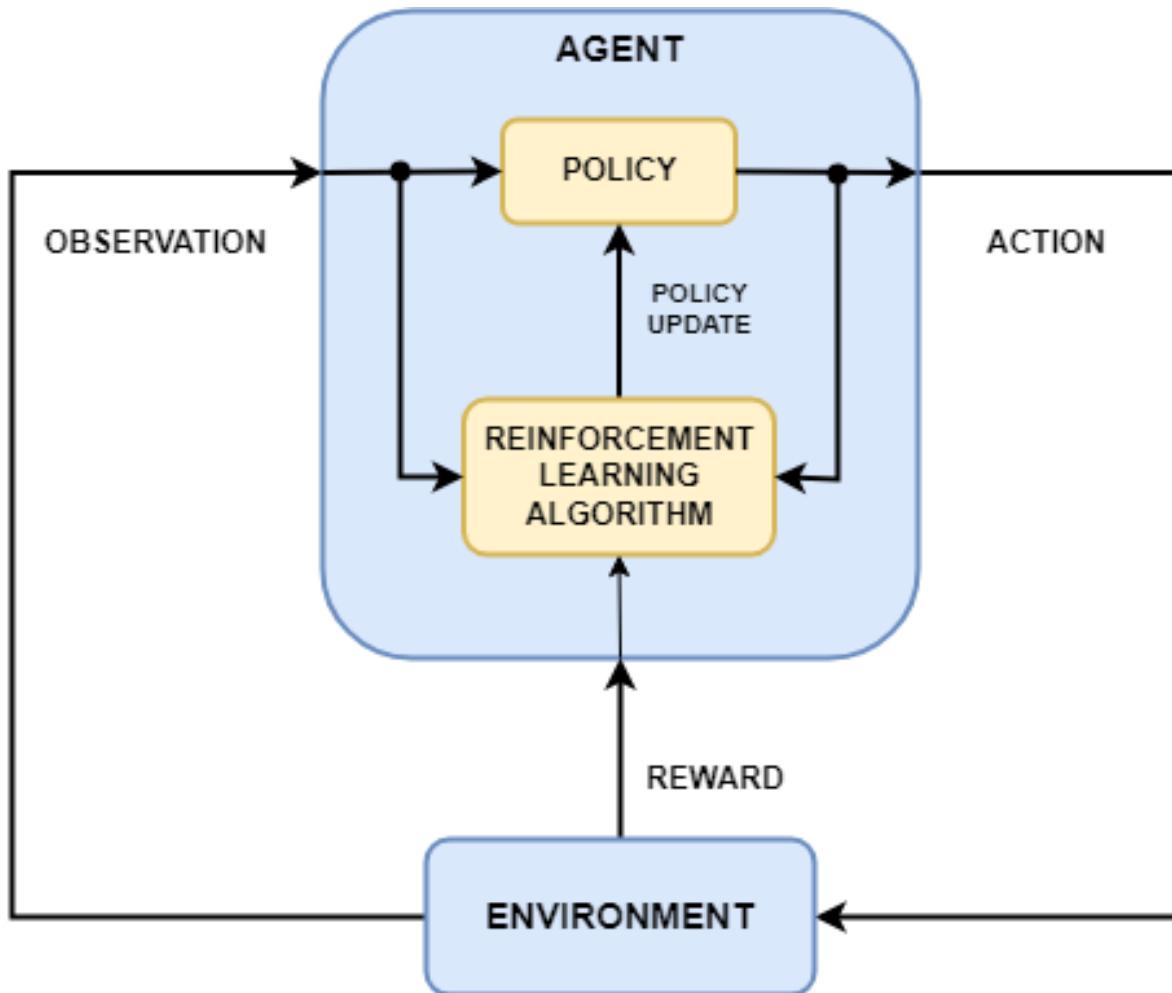


Abbildung 3.11: Reinforcement Learning
[MM:Web14]

Das Ziel des RL (Reinforcement Learning) ist es, im Laufe der Zeit Actions so zu wählen, dass der erwartete Wert des Returns maximiert wird und dabei die optimale Policy ausgewählt wird. Wir definieren Return Werte und Policies wie folgt. Der Return G_t ist die Reward abzüglich des Discounts an einem Time Step t .

$$G_t = R_{t+1} + \gamma R_{t+2} + \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}.$$

Die discounteten zukünftigen Rewards können als der aktuelle Wert zukünftiger Rewards interpretiert werden. Der Discount Factor bewertet die jetzigen Rewards höher als die späteren Rewards: wenn γ nahe an 0 ist, führt es zu "kurzsichtiger" Bewertung; γ nahe an 1 führt zu "weitblickender" Bewertung.
Einige Gründe warum ein Discount Factor in RL benutzt wird sind:

- Der Discount Factor vermeidet unendliche Rückkehr in Markov Processes
- Es besteht Unsicherheit über zukünftige Rewards.

[MM:Web15]

Es ist manchmal möglich, nicht discounted Markov Decision Processes zu verwenden (d.h. $\gamma = 1$).

Eine Policy π ist eine Verteilung über Actions in Abhängigkeit von States:

$$\pi(a|s) = P[A_t = a|S_t = s].$$

Es gibt zwei Value Functions, die State-value Function und die action-value Function. Sie helfen, die optimale Policy zu finden. Die State-value Function v_π eines MDP ist der erwartete Return, die von dem State s ausgeht, und dann der Policy π folgt:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s].$$

Die State-value Function $v_\pi(s)$ gibt den langfristigen Rewards des States s an, wenn die Policy π befolgt wird. Die State-value Function kann in zwei Teile zerlegt werden: in die unmittelbare Belohnung (immediate reward) R_{t+1} und den Discounted Wert des nachfolgenden States $\gamma v_\pi(S_{t+1})$.

$$\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi[G_t | S_t = s] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots | S_t = s] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma R_{t+3} + \dots | S_t = s] \\
&\quad = \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} + | S_t = s] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) + | S_t = s] \\
&\quad = \underbrace{\mathbb{E}_\pi[R_{t+1} | S_t = s]}_{\text{immediate reward}} \\
&\quad + \underbrace{\mathbb{E}_\pi[\gamma v_\pi(S_{t+1}) | S_t = s]}_{\text{discounted value of successor state}}.
\end{aligned}$$

[MM:Web15]

Die Action-value Function $q_\pi(s, a)$ ist der erwartete Return, welcher vom State s ausgeht, die Action a ausführt und dann der Policy π folgt:

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a].$$

Ebenso kann die Action-value Function wie folgt zerlegt werden:

$$q_\pi(s, a) = \mathbb{E}_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a].$$

Indem man $q_\pi(s, a)$ im Bezug auf $v_\pi(s)$ als $v_\pi(s)$ ausdrückt, erhält man die Bellman Gleichung für v_π :

$$v_\pi(s) = \sum_{a \in A} \pi(a|s) \left(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \sum_{a' \in A} \pi(a'|s') q_\pi(s', a') \right).$$

Die Bellman Gleichung verknüpft die State-value Function eines States mit der von anderen States. Die Ballman Gleichung für $q_\pi(s, a)$ lautet wie folgt:

$$q_\pi(s, a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \sum_{a' \in A} \pi(a'|s') q_\pi(s', a').$$

Eine Anwendung der Bellman Gleichung besteht darin, die Value Functions für eine gegebene Policy zu berechnen. Wenn wir alle Bellman Gleichungen in einem MDP mit n States kombinieren, erhalten wir n lineare Gleichungen für die n unbekannten Value Functions. Wir können die Value Functions berechnen, indem wir lineare Gleichungen lösen. Dieser Schritt kann jedoch eine Zeitkomplexität von $O(n^3)$ haben.

[MM:Web15]

Model Based Reinforcement Learning

Model Based Reinforcement Learning (MBRL) ist ein Ansatz im Reinforcement Learning, bei dem ein Agent ein Model der Umgebung lernt und es verwendet, um Entscheidungen zu treffen. S ist der State, A der Action Space und R die Reward Function. Der Agent interagiert mit der Umgebung über diskrete Zeitschritte $t = 0, 1, 2, \dots$

Dynamic Model

Das Dynamic Model, bezeichnet als $f : S \times A \rightarrow S$, sagt den nächsten Zustand voraus, gegeben den aktuellen Zustand und die Aktion. Mathematisch kann das Model wie folgt dargestellt werden:

$$s_{t+1} = f(s_t, a_t) \quad (3.2)$$

Hier repräsentieren s_t und s_{t+1} den aktuellen bzw. nächsten Zustand, und a_t ist die vom Agenten ausgeführte Aktion.

Data Efficiency im MBRL ist die Anzahl von Steps die das Model in einer Umgebung braucht, um ein bestimmtes Ziel zu erreichen. Die Verwendung von einem Dynamics Model führt einen Structural Bias ein, welches den Suchbereich der Verhalten des Agents einschränkt und damit die Sample Efficiency erhöht. Dies ist in Applikationen mit einem hohen Rechenaufwand pro Operation wichtig, was bei Model Based Reinforcement Learning der Fall ist, da MBRL Algorithmen sehr rechenintensiv sind.

Auf Dynamics Models wird später noch genauer eingegangen.
 [MM:Web24] [MM:Web18]

Reward Function

Die Reward Function, bezeichnet als $R : S \times A \rightarrow \mathbb{R}$, definiert die sofortige Belohnung, die der Agent für das Ausführen einer bestimmten Aktion in einem bestimmten Zustand erhält. Mathematisch kann sie ausgedrückt werden als:

$$r_t = R(s_t, a_t) \quad (3.3)$$

Hier ist r_t die Belohnung, die zum Zeitpunkt t erhalten wird.

Beim MBRL wird angenommen, dass eine Struktur gegeben ist, welche angibt, wie die Wahrscheinlichkeit einer Reward in einem bestimmten State ist.

$$R(s', s, a) = \Pr(r_{k+1} | s_{k+1} = s', s_k = s, a_k = a)$$

wobei \Pr die Wahrscheinlichkeit einer Reward bei einem State s und einer Action a ist, mit der Annahme, dass man in den nächsten State s' kommt und dann den erwarteten Reward r_{k+1} hat.

Policy-Optimierung

Das Ziel des Agenten ist es, eine optimale Policy $\pi : S \rightarrow A$ zu finden, die Zustände auf Actions abzubilden und die erwartete Reward zu maximieren. Die erwartete Reward, auch als Return bekannt, wird durch folgenden Ausdruck beschrieben:

$$J(\pi) = \mathbb{E}_{\tau \sim \pi} \left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) \right] \quad (3.4)$$

Hier repräsentiert $\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \dots)$ eine Trajectory, und $\gamma \in (0, 1)$ ist der Discount Factor.

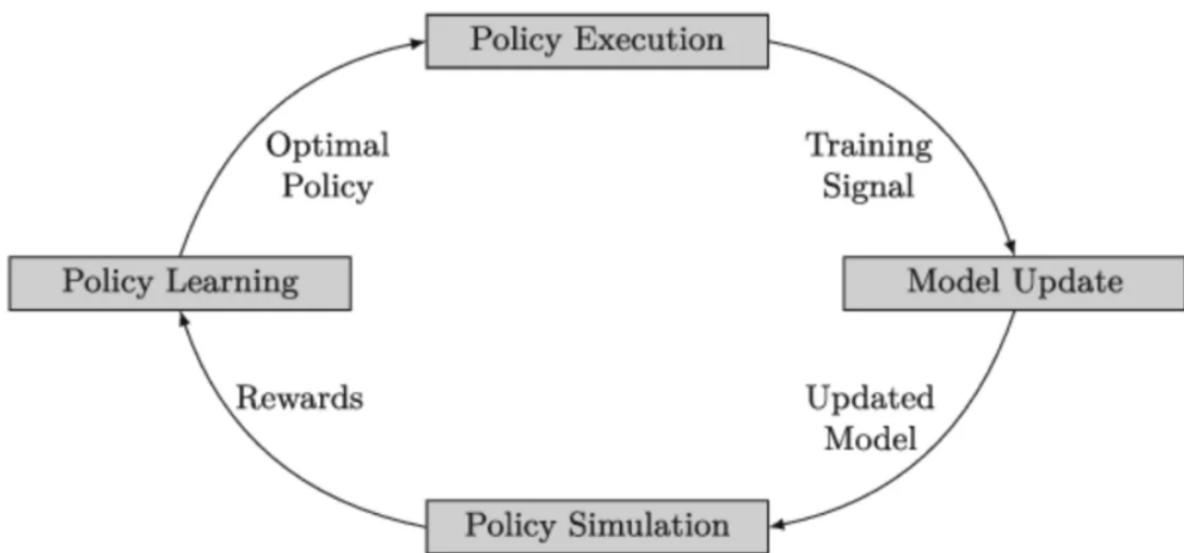


Abbildung 3.12: MBRL Flow Diagram
[MM:Web20]

[MM:Web16] [MM:Web17]

3.1.2 Einführung in Model Based Reinforcement Learning

Planning vs. Learning

Im MBRL sind Planning und Learning wichtige Grundkonzepte. Planning beinhaltet das Treffen von Entscheidungen unter Berücksichtigung eines Dynamics Models, während Learning darauf abzielt, dieses Model aufgrund von Interaktionen mit der Umgebung zu verbessern.

Beim Planning verwendet ein Agent sein Wissen über die Umgebung (Dynamics Model), um mögliche zukünftige States zu simulieren und die besten Actions zu bestimmen. Im Gegensatz dazu beinhaltet Learning das Aktualisieren des Models basierend auf den Ergebnissen von Interaktionen mit dem Environment, um die Genauigkeit der Entscheidungsfindung im Laufe der Zeit zu verbessern.

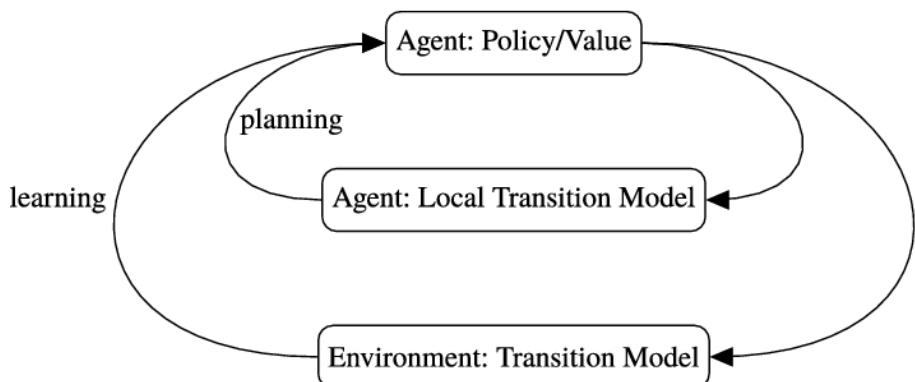


Abbildung 3.13: Planning and Learning
[MM:Web30]

Die Anzahl der Planning Steps kann die Leistung des Models sehr beeinflussen.

Durch Planning wird auch die Efficiency des Models verbessert, da die value function des Agents nicht nur bei der direkten Interaction von jeder Action geupdated werden kann.

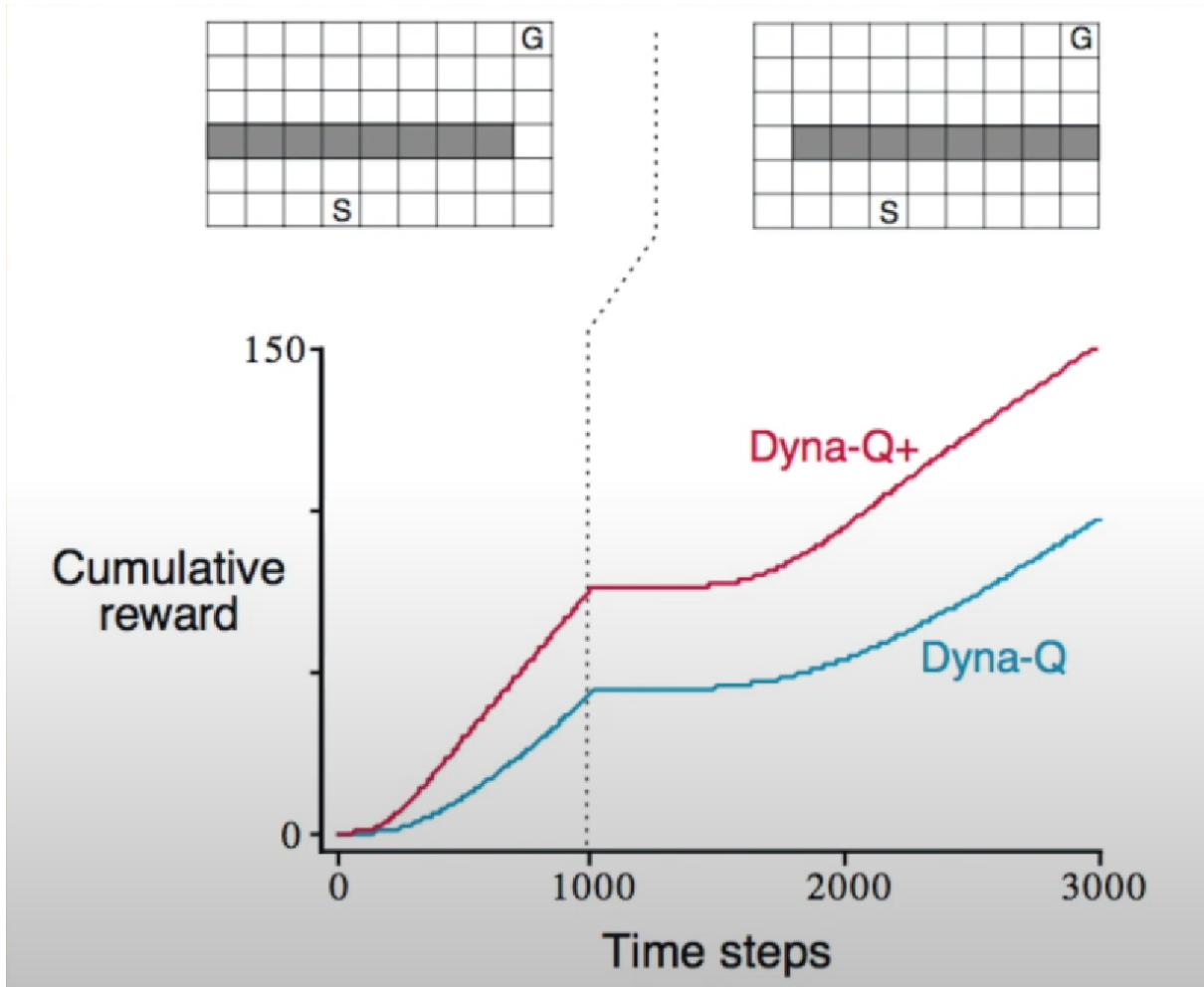


Abbildung 3.14: MBRL Environment Change 1

In der Abbildung 3.14 wird ein Environemnt Model für MBRL dargestellt welches sich ändert. Durch die Änderung des Environments, wäre das bisher gelernte Environment Model falsch. In der Abbildung 3.14 ändert Sich das Environment und wird dadurch Zeitaufwendiger.

Doch in der Abbildung 3.15 ändert sich das Environment so, dass es günstiger für den Agent ist.

Der Unterschied zwischen den 2 Models *Dyna-Q* und *Dyna-Q+* ist, dass *Dyna-Q+* auch Rewards für States gibt, welche noch nicht erreicht wurden, gibt.

In den zwei Abbildungen, kann man gut erkennen, dass wenn sich das Environment ungünstiger für den Agent geändert hat, die Reward für den *Dyna - Q+* Agent viel höher ist, als bei der Abbildung 3.15 wo es sich günstiger geändert hat.

$$r + \kappa\sqrt{\tau}$$

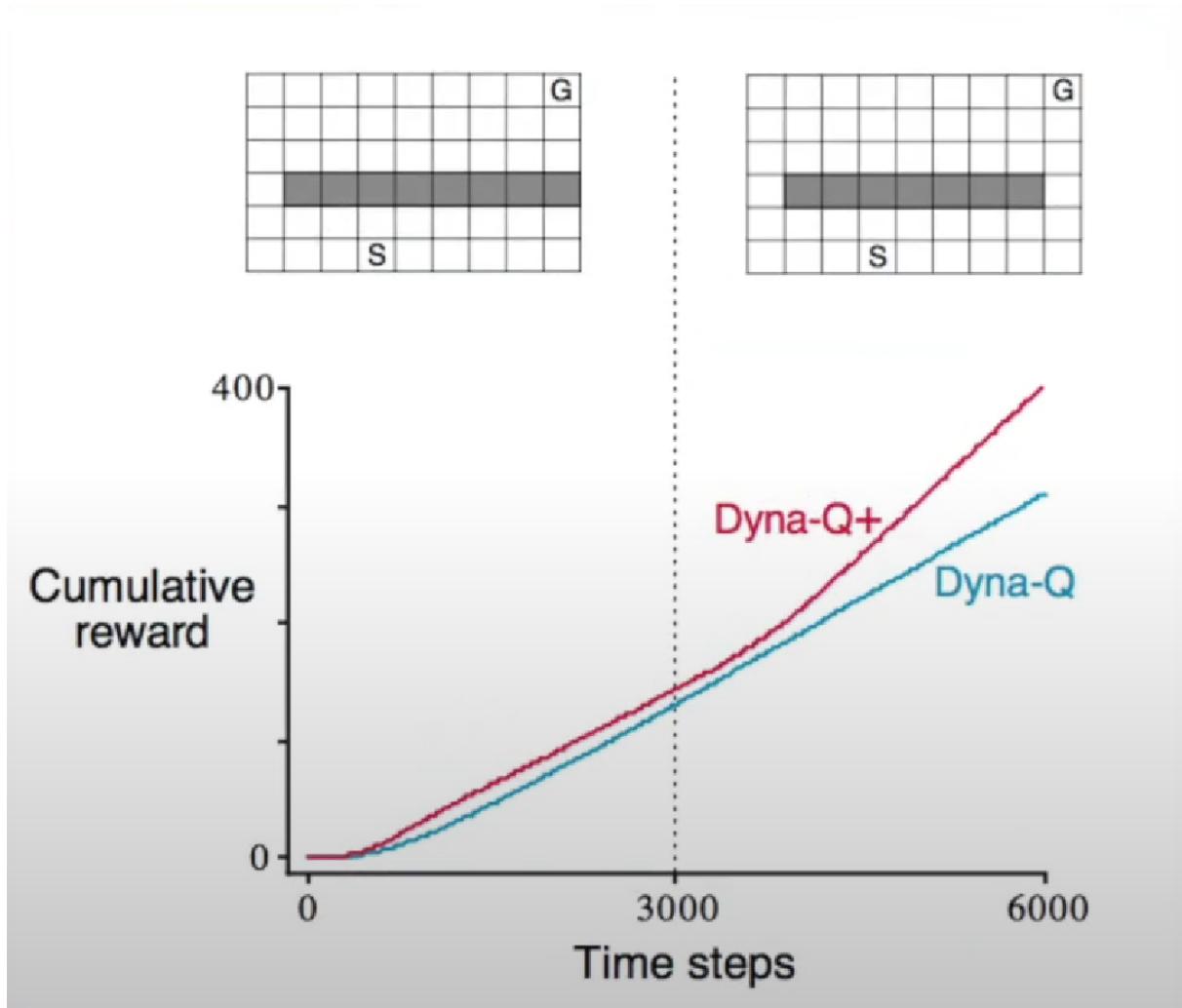


Abbildung 3.15: MBRL Environment Change 2

[MM:Web29]

Monte Carlo Tree Search (MCTS)

Die Monte Carlo Tree Search (MCTS) ist ein fortgeschrittener Algorithmus, der häufig in Entscheidungsprozessen mit großen State- und Action Spaces im Bereich des Reinforcement Learnings eingesetzt wird. Dieser Algorithmus zeichnet sich durch seine

Fähigkeit aus, auch in komplexen Umgebungen gute Entscheidungen zu treffen. Hier werden die grundlegenden Schritte und mathematischen Formulierungen für MCTS detailliert erläutert.

Grundlegende Schritte von MCTS: Die verschiedenen Schritte die im MCTS existieren können in der Abbildung 3.16 gesehen werden.

1. **Selection:** Der Baum wird von der Wurzel beginnend durchsucht, um den vielversprechendsten Knoten zu finden. Dies geschieht durch die Auswahl von Knoten basierend auf einer Trade-off-Funktion, die Exploration und Exploitation berücksichtigt.
2. **Expansion:** Der ausgewählte Knoten wird erweitert, indem neue Kinderknoten hinzugefügt werden. Diese repräsentieren mögliche zukünftige Zustände und Aktionen.
3. **Simulation:** Es werden Simulationen von den neuen Knoten bis zu einem Endzustand durchgeführt. Diese Simulationen können stochastisch sein und helfen dabei, die erwarteten Belohnungen der Aktionen zu schätzen.
4. **Backpropagation:** Die Ergebnisse der Simulation werden zurück entlang des Pfads zum Wurzelknoten propagierte. Dabei werden die statistischen Informationen der besuchten Knoten aktualisiert, wie z.B. die Anzahl der Besuche und die kumulierten Belohnungen.

[MM:Web40]

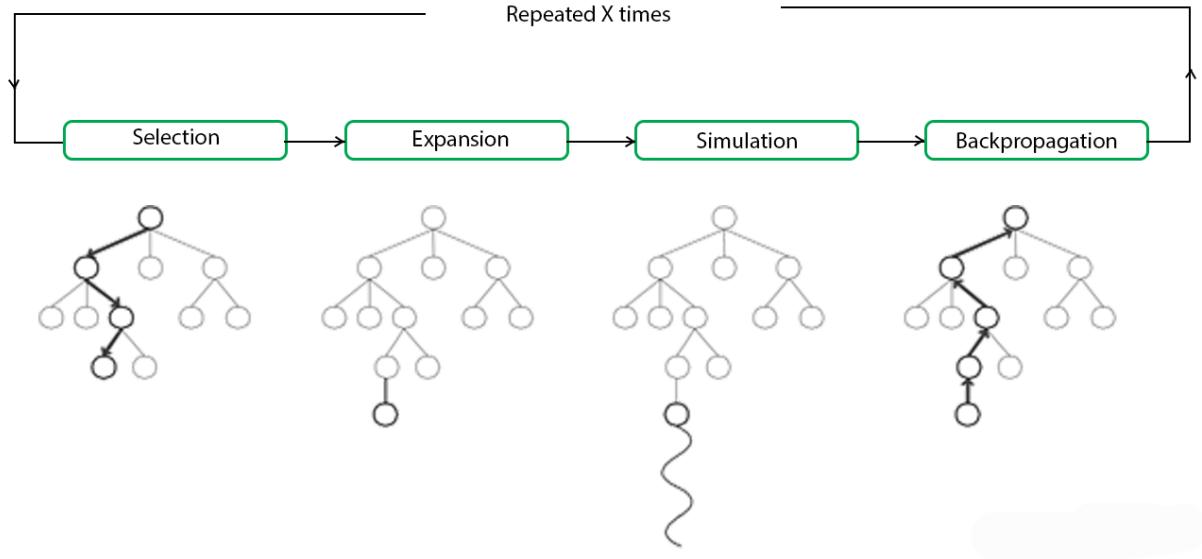


Abbildung 3.16: Monte Carlo Search Tree
[MM:Web19]

Dabei ist $N(s, a)$ die Anzahl der Besuche des Knotens, der dem Zustand s und der Aktion a entspricht. $Q(s, a)$ repräsentiert die kumulierte Belohnung für den Knoten s, a , und $W(s, a)$ ist die Summe der kumulierten Belohnungen der Simulationen von s, a . Die mathematischen Ausdrücke für die Aktualisierung lauten wie folgt:

$$N(s, a) \leftarrow N(s, a) + 1 \quad (3.5)$$

$$W(s, a) \leftarrow W(s, a) + \text{Simulationsbelohnung} \quad (3.6)$$

$$Q(s, a) = \frac{W(s, a)}{N(s, a)} \quad (3.7)$$

Die Auswahl während der Selektion erfolgt basierend auf einer Trade-off-Funktion, die Exploration und Exploitation abwägt. Eine häufig verwendete Funktion ist die UCT (Upper Confidence Bound for Trees) mit der Formel:

$$UCT(s, a) = Q(s, a) + c \cdot \sqrt{\frac{\ln(N(s))}{N(s, a)}} \quad (3.8)$$

Hierbei ist c ein Parameter, der den Trade-off zwischen Exploration und Exploitation steuert. Ein höheres c fördert mehr Exploration.

[MM:Web41]

3.1.3 Model Predictive Control (MPC)

Model Predictive Control (MPC) ist eine Steuerungsstrategie, die den Steuereingang über einen zukünftigen Zeithorizont optimiert, basierend auf einem Model des Systems. Das Ziel ist es, eine Cost Function zu minimieren, unter Berücksichtigung der Systemdynamik und den Einschränkungen. Die allgemeine Form des MPC-Optimierungsproblems ist:

$$\begin{aligned} \text{Die Minimierung von } & J(x_0, u_0, \dots, u_{N-1}) = \sum_{k=0}^{N-1} l(x_k, u_k) + l_f(x_N) \\ \text{unter Berücksichtigung von } & x_{k+1} = f(x_k, u_k), \\ & x_k \in \mathcal{X}, u_k \in \mathcal{U}, \forall k \in \{0, \dots, N-1\} \\ & x_N \in \mathcal{X}_f, \end{aligned} \quad (3.9)$$

wobei x_k und u_k den Zustand und den Steuereingang zum Zeitpunkt k darstellen. J ist die zu minimierende Cost oder Loss Function, bestehend aus einer Summe von Kosten $l(x_k, u_k)$ und den Endkosten $l_f(x_N)$. Die System Dynamics werden durch f gegeben, und \mathcal{X} , \mathcal{U} , und \mathcal{X}_f stellen die State-, Steuer- und Endzustandsbeschränkungen dar.

[MM:Web22]

Control und Optimization

Das Dynamics model wird für Control verwendet, indem es mit Verwendung einer Action, den nächsten State predicted.

Die trade-offs bei MPC im MBRL umfassen die Korrelation zwischen der Maximierung der log-likelihood bzw. wie genau das Model ist und der Maximierung des Episode

Rewards und ob diese Korrelation immer bedeutet, dass ein genaueres Model auch ein nützlicheres ist.

für Näheres siehe: [MM:Web24]

Das iterative Lernen des Dynamics Models in MBRL wird in der folgenden Abbildung 3.17 gezeigt.

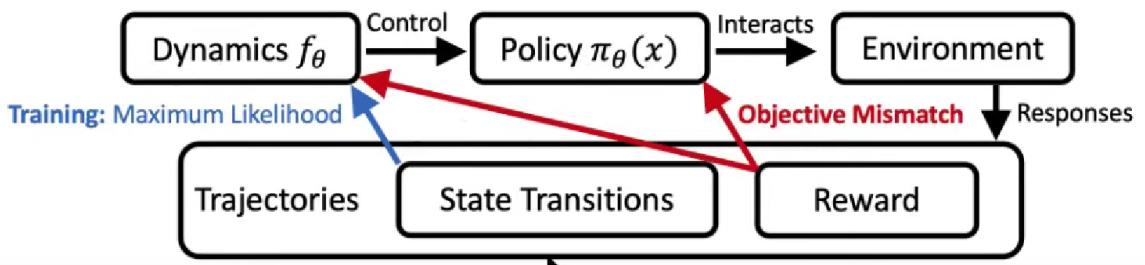


Abbildung 3.17: Dynamics Model

Wie in der Abbildung 3.17 zu sehen ist, wird das Dynamics Model anhand der Maximum Likelihood trainiert, eine Optimierung die hier angewandt werden könnte, auch die Reward für die Optimierung des Dynamics Models, der Policy, oder beides zu verwenden.

[MM:Web24]

Mathematisch kann dieser Objective Mismatch wie folgt angeschrieben werden:

Optimierung im Training:

$$\operatorname{argmax} \sum_{i=1}^N \log p_\theta(s'_i | s_i, a_i)$$

Control:

$$\operatorname{argmax} E_{\pi_\theta}(s_t) \sum_{i=t}^{t+T} r(s_i, a_i)$$

Dieser Objective Mismatch wird verwendet, um zu erforschen ob die Annahme, dass die Maximierung von Maximum Likelihood auch ein besseres Model ergibt.

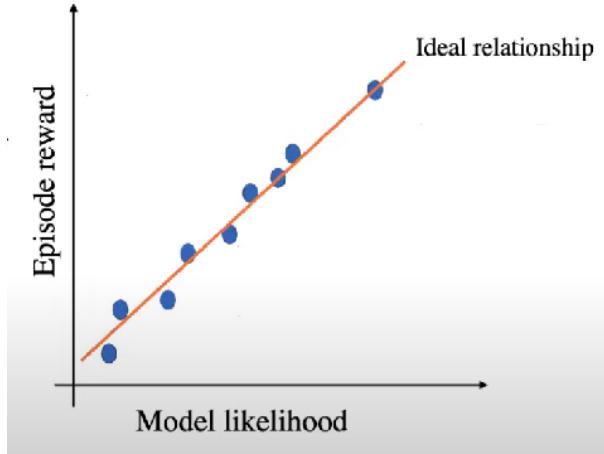


Abbildung 3.18: likelihood - Reward

In der Abbildung 3.18 sieht man, wie eine optimale Korrelation zwischen Likelihood, also Präzision und der Episode Reward.

Des Weiteren werden Tests auf verschiedenen Datasets ausgeführt, welche, wie in der Abbildung 3.19 auf drei verschiedenen Datasets zu sehen ist, hervorheben, dass die Dynamics Models, von welchen die Daten gesammelt wurden, z.B.: auf dem Expert Dataset, fast keine Korrelation zwischen der Likelihood und der Episode Reward besteht. Obwohl das Dynamics Model im MBRL auf einer sehr spezifischen Umgebung trainiert wird, sieht man bei zwei von den drei Datasets wenig Korrelation zwischen den zwei Metriken. [MM:Web25]

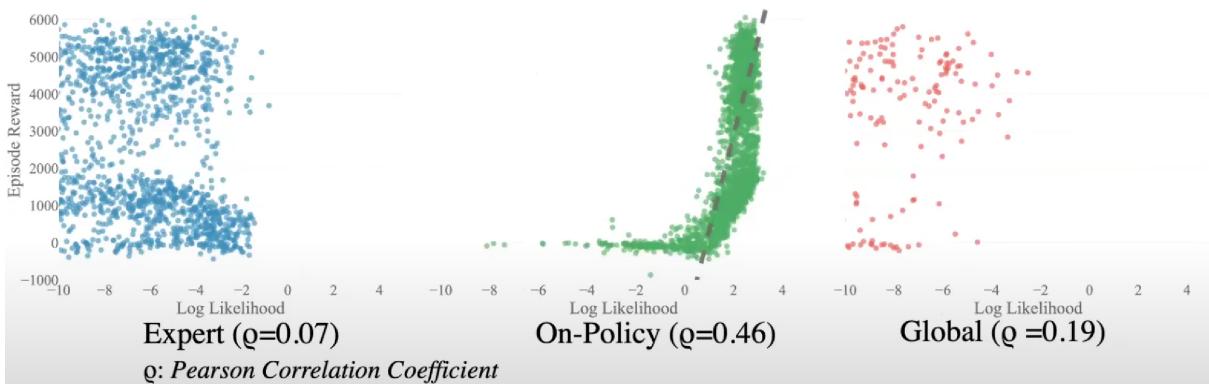


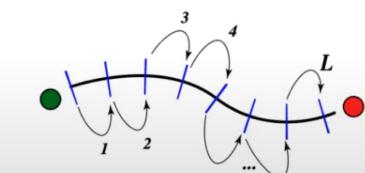
Abbildung 3.19: Correlation

Dies wirft die Frage auf, was man im MPC bei MBRL optimieren sollte. Durch eine Adversarial Attack auf das Dynamics Model, bei dem ein sehr präzises Model verwendet wird, welches wenig Rewards bekommt, dabei wurde die Entdeckung gemacht, dass

Standard one-step lookahead

- Compounding predictions

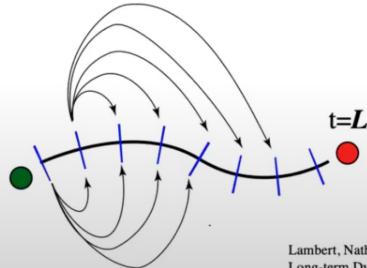
$$s_{t+1} = s_t + f_\theta(s_t, a_t)$$



Trajectory-based models

- Time dependent prediction

$$s_{t+h} = f_\theta(s_t, h, \theta_\pi)$$



Lambert, Nathan O., et al. "Learning Accurate Long-term Dynamics for Model-based Reinforcement Learning." *arXiv preprint arXiv:2012.09156* (2020)

Abbildung 3.20: Trajectory Model

die Model Accuracy gleich blieb, da es Accuracy bei manchen stellen verlor, doch bei anderen dazu gewonnen hatte.

Um diesen Objective Mismatch nun zu beseitigen, wird das Model darauf Trainiert die Trajectories zu predicten, welche in der Abbildung 3.17 als die State Transition und der Rewards zu sehen sind.

In der Abbildung 3.20 wird der Unterschied zwischen dieser Standard one step lookahead dynamics Model Optimierung, welche schon bei State Transition Models besprochen wurde und der Trajectory-based Models dargestellt. Der größte Unterschied besteht darin, dass man das langzeit verhalten der Models bestimmen kann.

Dabei repräsentiert s_t den Starting State, h ist der Prediction Horizon, welcher angibt wie viele Steps man in der Zukunft predicten will. θ_π sind Control Parameter für das Model.

Dieser Ansatz des MPC gibt uns mehr Daten und eine Langzeit Prediction der Action für unser Model. Außerdem führt es eine parallelisierte Prediction mit mehr labeled Daten ein. Dies führt auch zu Recheneffizienteren Models, welche im MBRL wichtig sind.

[MM:Web24] [MM:Web25]

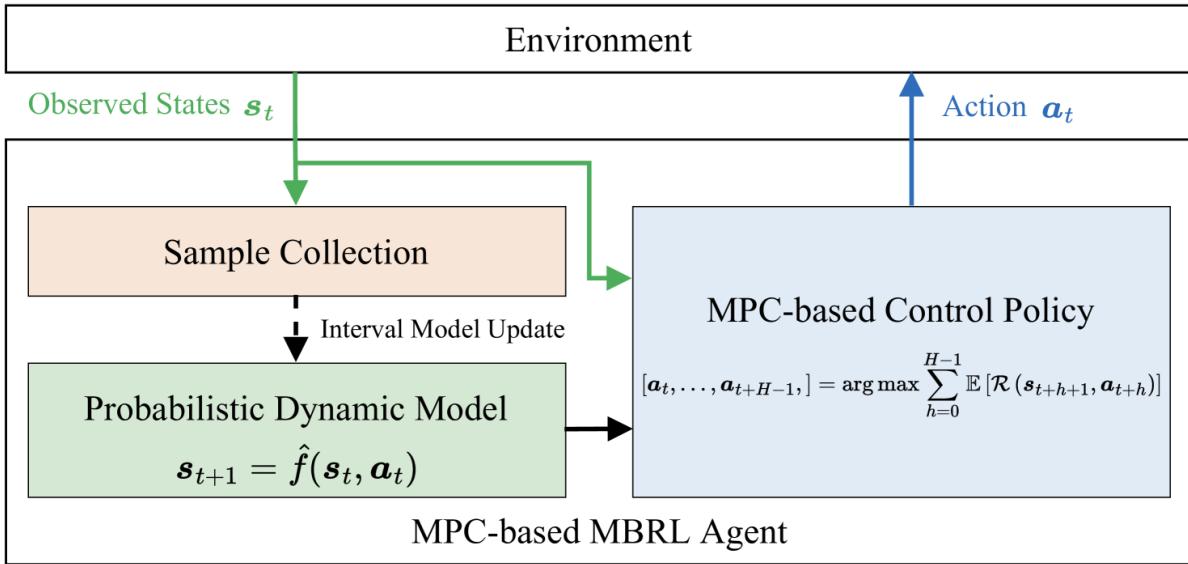


Abbildung 3.21: Probabilistic Model

3.1.4 Probabilistic Models im Model Based RL

Probabilistic Models werden im MBRL verwendet, um die Unsicherheit in der Dynamik der Umgebung darzustellen. Diese Models können die Wahrscheinlichkeitsverteilung des nächsten Zustands, mit dem gegebenen aktuellen Zustand und der Aktion vorhersagen.

In der Abbildung 3.21 ist ein Probabilistic MBRL Agent zu sehen. Der State und Action sind mit s_t und a_t angeschrieben. Das Probabilistic MBRL schätzt die system dynamics mit:

$$[\mu(s_t + 1), \epsilon(s_t + 1)] = \hat{f}(s_t, a_t) + \omega$$

Dabei ist μ der Durchschnitt und ϵ die Varianz. ω repräsentiert die Uncertainty.

[MM:Web26]

Uncertainty Propagation und Risk-Reduction

Umso mehr Daten zur Verfügung stehen, umso weniger Uncertainty besteht. Die Uncertainty des Models beruht auf der Stochastizität, umso stochastischer ein Model ist, umso mehr Uncertainty besteht.

Diese Uncertainty kann durch Diversifikation und Stochastic Policies verbessert werden, in welchen ein Schema konstruiert welches garantiert konvergiert.

Solch ein Algorithmus initialisiert eine Policy mit gleich wahrscheinlich eintretenden Actions. In jeder Iteration wird die Wahrscheinlichkeit der besten Action Q_n^m um $\frac{1}{m}$ erhöht. m ist dabei die jetzige Iteration. Währenddessen werden die Wahrscheinlichkeiten von allen anderen Actions verringert:

$$\forall s, a : \pi^m(a|s) := \begin{cases} \min(\pi^{m-1}(a|s) + \frac{1}{m}, 1), & \text{wenn } a = \arg \max_u Q^{m-1}(s) \\ \max(1 - \pi^{m-1}(s, a_{\arg \max_u Q^{m-1}(s)}) - \frac{1}{m}, 0), & \text{sonst} \end{cases} \quad (3.10)$$

$a_{\arg \max_u Q^{m-1}(s)}$ ist die beste Action. Durch die Verringerung der Veränderungsrate: $\Delta\pi(a|s)$ ist die Erreichung von allen möglichen Policies garantiert.
 [MM:Web27] [MM:Web28]

3.1.5 Implementierung

Das nächste Kapitel veranschaulicht die Implementierung eines MBRL Agents für ein Environment mithilfe eines Flappy Bird Clones und DQN. Flappy Bird wurde als Environment gewählt aufgrund seines Discrete Action Spaces und simplen Observations. Die Ziele und Details der AI werden im Folgenden erläutert.

Flappy Bird Clone

Mit der Pygame Library wurde ein Flappy Bird Clone erstellt, welcher wichtige Komponenten und Funktionalitäten umfasst:

- **Spielinitialisierung:** Richtet das Spiel Fenster, Attribute und Spielmechanikvariablen ein.

```
1 class FlappyBird:  
2     def __init__(self):  
3         pygame.init()  
4         self.screen_width = 600  
5         self.screen_height = 400  
6         self.screen = pygame.display.set_mode((self.screen_width, self.  
7             .screen_height))  
8         pygame.display.set_caption("Flappy Bird Clone")  
9         self.clock = pygame.time.Clock()  
10        self.bird_size = 30  
11        self.bird_color = (255, 255, 0)  
12        self.bird_x = 100  
13        self.bird_y = self.screen_height // 2 - self.bird_size // 2  
14        self.bird_velocity = 0  
15        self.jump_strength = -10  
16        self.pipe_width = 50  
17        self.pipe_color = (0, 255, 0)  
18        self.pipes = []  
19        self.pipe_gap = 150  
20        self.pipe_speed = 5  
21        self.pipe_interval = 65  
22        self.pipe_timer = 0  
23        self.score = 0  
24        self.font = pygame.font.Font(None, 36)  
25        self.distance = 0  
26        self.game_over = None
```

- **Spiel-Schleife:** Verarbeitet Ereignisse, aktualisiert den Spielzustand und zeichnet den Bildschirm bei jedem Tick neu.

```

1 def run(self):
2     while True:
3         for event in pygame.event.get():
4             if event.type == pygame.QUIT:
5                 pygame.quit()
6                 sys.exit()
7             elif event.type == pygame.KEYDOWN and event.key == pygame.K_SPACE:
8                 self.jump()
9             self.distance += 1
10            self.update()
11            self.draw()
12            pygame.display.flip()
13            self.clock.tick()

```

Der Methode `clock.tick()` wurde kein Parameter für die Update-Rate gegeben, da es in diesem Fall die Simulation so schnell wie möglich laufen lässt, welches den Trainingsprozess beschleunigt.

- **Bewegung und Sprungmechanik:** Verwaltet die vertikale Geschwindigkeit und Position des Agents basierend auf Sprüngen.

```

1 def jump(self):
2     self.bird_velocity = self.jump_strength
3
4 def update(self):
5     if not self.game_over:
6         self.bird_velocity += 1
7         self.bird_y += self.bird_velocity
8         self.generate_pipes()
9         self.move_pipes()
10        self.game_over = self.check_collision()

```

- **Erzeugung und Bewegung der Rohre:** Erstellt dynamisch Rohre, die sich über den Bildschirm bewegen.

```

1 def generate_pipes(self):
2     self.pipe_timer += 1
3     if self.pipe_timer == self.pipe_interval:
4         pipe_height = random.randint(100, 300)
5         self.pipes.append({'x': self.screen_width, 'height':
6             pipe_height})
7         self.pipe_timer = 0

```

- **Kollisionserkennung:** Diese Funktion überprüft, ob der Agent mit einem der Rohre oder den Bildschirmgrenzen kollidiert ist.

```

1 def check_collision(self):
2     for pipe in self.pipes:
3         if (
4             self.bird_x < pipe['x'] + self.pipe_width
5             and self.bird_x + self.bird_size > pipe['x']
6             and (self.bird_y < pipe['height'] or self.bird_y +
7                   self.bird_size > pipe['height'] + self.pipe_gap)
8         ):
9             return True
10
11         if self.bird_y + self.bird_size > self.screen_height or self.
12             bird_y < 0:
13                 return True
14
15     return False

```

Die Funktion durchläuft alle Rohre und überprüft, ob der Agent mit einem der Rohre kollidiert, indem sie die Position des Agents mit den Positionen und Größen der Rohre vergleicht. Wenn eine Kollision festgestellt wird, gibt die Funktion True zurück, was bedeutet, dass das Spiel beendet ist. Andernfalls fährt das Spiel fort, bis eine Kollision erkannt wird oder der Spieler das Spiel beendet.

Das Script hat auch getter-Methoden für wichtige Daten, welche für das Training des AI Agents gebraucht werden.

```

1 def render(self):
2     self.distance += 1
3     self.update()
4     self.draw()
5
6     pygame.display.flip()
7     self.clock.tick() #default: 30

```

Die render-Methode wird im Environment in jedem step aufgerufen, um die distance zu erhöhen und die Simulation zu aktualisieren.

Environment

Anschließend wird der Code, der eine Umgebung für einen AI Agent schafft, um mit dem Spiel Flappy Bird zu interagieren, indem er die `gym` Bibliothek verwendet, erklärt. Diese Bibliothek ist in der Forschung für Machine Learning und insbesondere für Reinforcement Learning weit verbreitet.

Initialisierung des Environments

```
1 class FlappyBirdEnvironment(gym.Env):
2     def __init__(self):
3         super(FlappyBirdEnvironment, self).__init__()
4         self._flappy_bird_game = FlappyBird()
5         self._action_space = spaces.Discrete(2)
6         self._observation_space = spaces.Box(low=-np.inf, high=np.inf,
7                                             shape=(5,), dtype=np.float32)
8         self._state = None
9         self._reward = 0.0
10        self._episode_ended = False
```

Die Klasse `FlappyBirdEnvironment` erbt von `gym.Env`, um eine benutzerdefinierte Umgebung zu schaffen. Der Konstruktor initialisiert das Spiel, definiert einen Action Space von zwei möglichen Aktionen (nicht springen und springen) und legt den Observation Space fest, der aus einem 5D-Vektor besteht, welcher die Zustände des Spiels repräsentiert. Diese Zustände sind wie auch in 3.22 zu sehen:

1. Die vertikale Distanz zum nächsten oberen Rohr
2. Die vertikale Distanz zum nächsten unteren Rohr.
3. Die horizontale Distanz zu den nächsten Rohren
4. Die Y-Position des Agents
5. Die Fallgeschwindigkeit (Velocity) des Agents

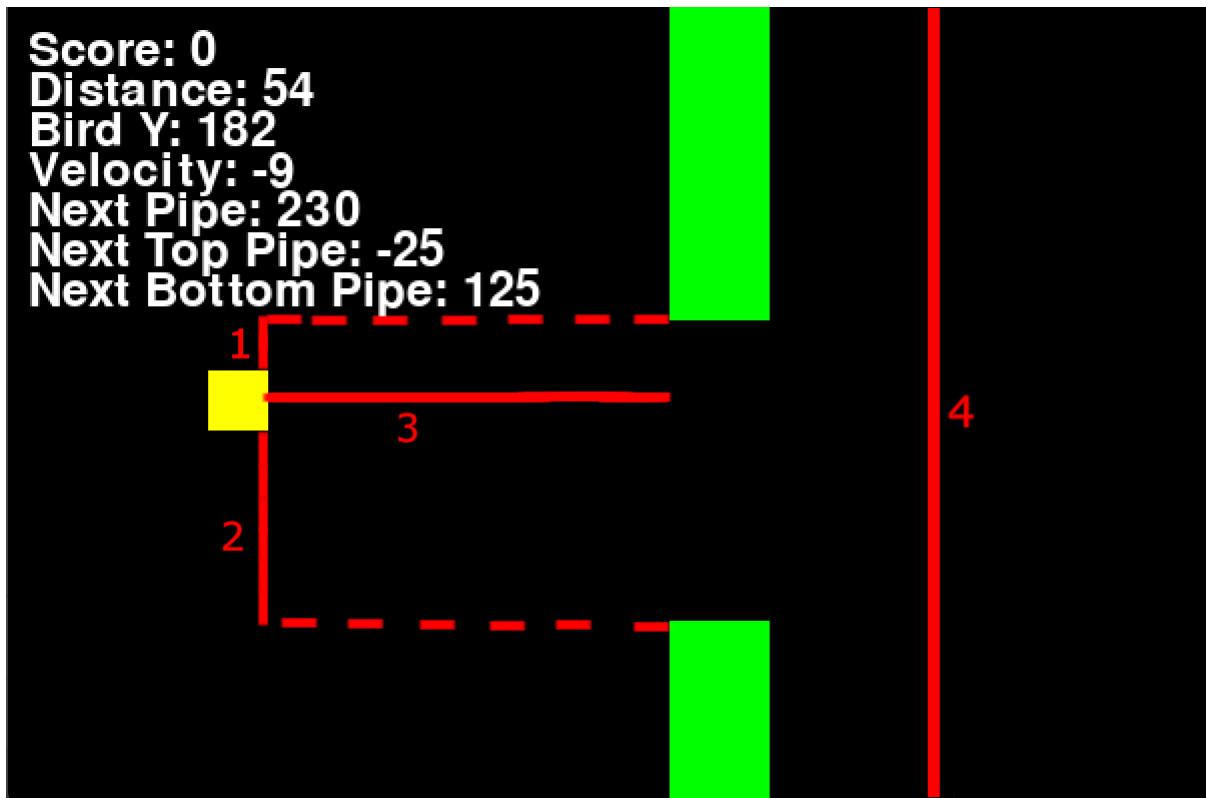


Abbildung 3.22: Flappy Bird Observations

Zurücksetzen des Environments

```

1 def reset(self):
2     self._flappy_bird_game.reset()
3     self._state = np.full((5,), 0.0, dtype=np.float32)
4     self._reward = 0.0
5     self._episode_ended = False
6     self._state = np.array(self._state, dtype=np.float32)
7     return self._state

```

Die `reset`-Methode wird aufgerufen, um die Umgebung auf ihren Anfangszustand zurückzusetzen. Dabei wird das Spiel neu gestartet und der Zustandsvektor neu initialisiert. Die Reward, welcher dem Agenten übergeben wird, wird wieder auf 0 gesetzt und die Variable `_episode_ended` welche angibt, ob das Environment zurückgesetzt werden soll, wird auf `False` gesetzt. Die `reset`-Methode returned den State des Environments beim Zurücksetzen.

Ausführen einer Action

Diese Methode wird bei jedem Schritt, den der Agent macht, aufgerufen.

```

1 def step(self, action):
2     if self._episode_ended:
3         return self.reset()
4
5     self._flappy_bird_game.render()
6
7     observation = self._get_observation()
8
9     if action == 1:
10        self._flappy_bird_game.jump()
11
12    self._reward = self._flappy_bird_game.score + (self._flappy_bird_game.
13        distance / 200)
14
15    if self._flappy_bird_game.game_over:
16        self._episode_ended = True
17        print(f"Episode ended with score {self._flappy_bird_game.score}"
18              f" and distance {self._flappy_bird_game.distance}")
19        return observation, -20 + self._reward, self._episode_ended, {}
20
21    return observation, self._reward, self._episode_ended, {}

```

Die `step`-Methode führt eine Action aus (springen, wenn `action==1`) und aktualisiert die Umgebung basierend auf dieser Aktion. Sie berechnet auch Belohnung und Beobachtung und prüft, ob das Spiel vorbei ist. Die Belohnung setzt sich aus der Distanz, welche der Agent zurückgelegt hat und dem `score`, welcher angibt, durch wie viele Rohre ohne Kollision durchgeflogen wurde, zusammen.

Wenn die Variable `game_over` im Flappy Bird Script `True` ist, dann wird eine Penalty von $-20 + \text{reward}$ gegeben. Damit wird die Penalty je kleiner, umso weiter der Agent es schafft. Nach jedem Reset gibt das Environment Daten über die Episode zurück, diese wären, der Score und die Distance.

Falls keine Kollision auftritt wird die `reward` für diesen Timestep returned und die Simulation läuft weiter.

Observations erhalten

```

1 def _get_observation(self):
2     bird_y = self._flappy_bird_game.get_bird_y()
3     bird_velocity = self._flappy_bird_game.get_bird_velocity()
4     distance_to_next_pipe = self._flappy_bird_game.
5         get_distance_to_next_pipe()
6     vertical_distance_to_next_bottom_pipe = self._flappy_bird_game.
7         get_vertical_distance_to_next_bottom_pipe()
8     vertical_distance_to_next_top_pipe = self._flappy_bird_game.
9         get_vertical_distance_to_next_top_pipe()
10
11     observation = np.array([
12         bird_y,
13         bird_velocity,
14         distance_to_next_pipe,
15         vertical_distance_to_next_top_pipe,
16         vertical_distance_to_next_bottom_pipe,
17     ], dtype=np.float32)
18
19     return observation

```

Diese Methode extrahiert den aktuellen Zustand der Umgebung als einen Vektor, der die Y-Position des Agents, seine Geschwindigkeit, die Distanz zum nächsten Rohr und die vertikalen Distanzen zu den nächsten oberen und unteren Rohren umfasst.

Zugriffsmethoden

```

1 @property
2 def action_space(self):
3     return self._action_space
4
5 @property
6 def observation_space(self):
7     return self._observation_space

```

Diese Eigenschaftsmethoden geben den Action- und Observation Space zurück, die für die Interaktion mit dem Environment essentiell sind.

Der Code schafft eine interaktive Lernumgebung für den AI Agent, indem er die Komplexität des Flappy Bird Spiels in ein Format bringt, das mit Reinforcement Learning Methoden kompatibel ist. Dies ermöglicht es, Algorithmen zu entwickeln und zu testen, die lernen, das Spiel durch Trial-and-Error zu meistern.

Agent

Folgend wird eine MBRL Implementierung eines DQN Agenten erklärt welcher dann mit diesem Flappy Bird Environment interagiert, um es zu lernen.

Zuerst müssen wir alle nötigen Komponenten für den MBRL Agenten importieren:

```
1 import numpy as np
2 from matplotlib import pyplot as plt
3 from DQN import DQN
4 from EnvModel import EnvironmentModel
5 from Env import env
6
7 agent = DQN()
8
9 state_size = env.observation_space.shape[0]
10 action_size = env.action_space.n
11
12 env_model = EnvironmentModel(state_size, action_size)
```

Hier werden das `DQN` das `EnvModel` (Environment Model) und das `Env` (Environment) importiert.

Das Environment wird durch die `gym` Library registriert und dann für die Verwendung als `env` initialisiert.

```
1 import gym
2
3 gym.register(id='FlappyBird-m', entry_point='flappy_bird_gym.envs.
4     flappy_bird_env:FlappyBirdEnvironment')
5 env = gym.make('FlappyBird-m')
```

In diesem Fall heißt das Environment "FlappyBird-m" und wird aus dem Package `flappy_bird_gym.envs.flappy_bird_env` importiert.

Das DQN ist die Klasse, welche der agent zum Lernen verwendet.

```

1 class DQN():
2     def __init__(self):
3         self.discount_factor = 0.99
4         self.learning_rate = 0.001
5         self.eps_min = 0.1
6         self.eps_max = 1.0
7         self.eps_decay_steps = 2000000
8         self.eps_decay_rate = (self.eps_max - self.eps_min) / self.
9             eps_decay_steps
10        self.replay_memory_size = 5000
11        self.replay_memory = deque([], maxlen=self.replay_memory_size)
12        self.n_steps = 40000 # total number of training steps
13        self.training_start = 5 # start training after 10,000 game
14            iterations
15        self.training_interval = 4 # run a training step every 4 game
16            iterations
17        self.save_steps = 50 # save the model every 1,000 training steps
18        self.copy_steps = 100 # copy online DQN to target DQN every 10,000
19            training steps
20        self.discount_rate = 0.99
21        self.batch_size = 64
22        self.iteration = 0 # game iterations
23        self.done = True # env needs to be reset
24        self.rewards_history = []

25        self.model = self.DQNmodel()
26        self.target_model = self.DQNmodel()
27        self.update_target_model()

```

Discount Factor

Wert: 0,99

Beschreibung: Der Discount Factor, auch als γ bezeichnet, bestimmt die Wichtigkeit zukünftiger Belohnungen. Ein Wert von 0,99 bedeutet, dass zukünftige Belohnungen fast so wichtig sind wie sofortige Belohnungen, was den Agenten ermutigt, langfristige Vorteile zu berücksichtigen.

Learning Rate

Wert: 0,001

Beschreibung: Dies ist die Schrittgröße, mit der die Gewichte des neuronalen Netzwerks während des Trainings aktualisiert werden. Ein niedriger Wert wie 0,001 hilft bei kleinen Anpassungen, um eine sanfte Konvergenz zu gewährleisten, kann jedoch den Lernprozess verlangsamen.

eps_min, eps_max und eps_decay_steps

Werte: 0,1, 1,0, 2000000

Beschreibung: Diese Parameter steuern den Trade-off zwischen Exploration und Exploitation durch eine epsilon-greedy-Strategie. `eps_max` ist die anfängliche Erkundungsrate, `x_min` ist die minimale Erkundungsrate und `eps_decay_steps` ist die Anzahl der Schritte, über die die Erkundungsrate abnimmt. Dies ermöglicht es dem Agenten, anfangs weitgehend zu erkunden und dann mehr auszunutzen, während er lernt.

`eps_decay_rate`

Berechnung: $\frac{\text{eps_max} - \text{eps_min}}{\text{eps_decay_steps}}$

Beschreibung: Bestimmt die Rate, mit der die Wahrscheinlichkeit der Erkundung im Laufe der Zeit abnimmt und fördert einen allmählichen Übergang von Exploration zu Exploitation.

Epsilon Greedy:

```

1 def epsilon_greedy(self, q_values, step):
2     self.epsilon = max(self.eps_min, self.eps_max - (self.eps_max - self.
3         eps_min) * step / self.eps_decay_steps)
4     if np.random.rand() < self.epsilon:
5         return np.random.randint(2) # random action
6     else:
7         return np.argmax(q_values) # optimal action

```

- **Epsilon-Berechnung:** Die Funktion beginnt mit der Berechnung des Werts von Epsilon (ϵ) für den aktuellen Schritt. Der Wert von ϵ wird so eingestellt, dass er über `self.eps_decay_steps` Schritte von `self.eps_max` auf `self.eps_min` abnimmt. Diese allmähliche Abnahme ermöglicht es dem Agenten, zu Beginn weitreichend zu exploren und mehr zu exploiten, während er über die Umgebung lernt.
- **Zufällige Aktionsauswahl:** Wenn eine zufällig generierte Zahl kleiner als der aktuelle Epsilon-Wert ist, wählt der Agent eine zufällige Action. Dies ist der Exploration Teil, in dem der Agent verschiedene Actions ausprobiert, um deren Konsequenzen zu lernen.
- **Optimale Aktionsauswahl:** Wenn die zufällig generierte Zahl größer oder gleich dem aktuellen Epsilon-Wert ist, wählt der Agent die Aktion mit dem höchsten Q-Wert. Dies ist der Exploitation Teil, in dem der Agent sein aktuelles Wissen nutzt, um die beste Entscheidung zu treffen.
- **Rückgabe der Aktion:** Die Funktion gibt die ausgewählte Action zurück, die entweder eine zufällige Action (für die Exploration) oder die Action mit dem höchsten Q-Wert (für die Exploitation) sein kann.

replay_memory_size

Wert: 5000

Beschreibung: Größe der Replay Memory. Dies ist die Anzahl der gespeicherten früheren Erfahrungen für das Training des Networks. Eine größere Memory Size ermöglicht eine vielfältigere Sammlung von Erfahrungen während des Trainings, was das Lernen verbessern kann.

replay_memory

Typ: deque mit einer maxlen

Beschreibung: Eine Double-Ended-Queue, die die letzten 'replay_memory_size' Erfahrungen speichert. Ältere Erfahrungen werden automatisch entfernt, während neue hinzugefügt werden, wodurch eine feste Größe beibehalten wird.

save_steps und copy_steps

Werte: 50, 100

Beschreibung: Diese Parameter verwalten das Training. `save_steps` ist, wie oft das Model gespeichert wird, und `copy_steps` ist die Häufigkeit, mit der die Gewichte des Online-DQN auf das Ziel-DQN für stabiles Lernen kopiert werden.

batch_size

Wert: 64

Beschreibung: Die Anzahl der Erfahrungen, die aus der Replay Memory entnommen werden, um das Netzwerk bei jedem Trainingsschritt zu aktualisieren..

model und target_model

Beschreibung: `model` repräsentiert das Online-DQN, das für die Auswahl von Aktionen verwendet wird, während `target_model` für die Berechnung der Ziel-Q-Werte während des Trainings verwendet wird. Diese Trennung hilft, den Lernprozess zu stabilisieren.

```
1 def DQNmodel(self):
2     model = Sequential()
3     model.add(Dense(32, input_shape=(5,), activation='relu'))
4     # Adjust input shape
5     model.add(Dense(32, activation='relu'))
6     model.add(Dense(2, activation='softmax'))
7     model.compile(loss='categorical_crossentropy', optimizer=
        Adam(lr=self.learning_rate))
8     return model
```

Das DQN (Deep Q-Network) Model, definiert im Code-Snippet, ist eine einfache neuronale Netzwerkarchitektur, die für Aufgaben des Reinforcement-Learnings konzipiert wurde. Es verwendet die Sequential API von Keras, was bedeutet, dass es sich um einen linearen Stapel von Netzwerkschichten handelt.

1. **Initialisierung des Sequential-Models:** Das Model wird als `Sequential` Objekt initialisiert, das ein linearer Stapel von Neural Network Layers ist.
2. **Eingabeschicht:** Die erste `Dense`-Schicht dient als Eingabeschicht mit 32 Neuronen und verwendet die ReLU Activation Function. `input_shape=(5,)` gibt an, dass die Eingabe des Models Arrays der Form `(5,)` sein werden.
3. **Versteckte Schicht:** Die zweite `Dense`-Layer ist eine Hidden Layer, ebenfalls mit 32 Neuronen und der ReLU-Aktivierungsfunktion.
4. **Ausgabeschicht:** Die dritte `Dense`-Layer ist die Output Layer des Models mit 2 Neuronen, die den möglichen Aktionen des Agenten entsprechen. Sie verwendet die `softmax` Aktivierungsfunktion.
5. **Kompilierung:** Das Model wird mit der `categorical_crossentropy` Loss-Function und dem Adam-Optimizer mit der oben erklärten `self.learning_rate` spezifizierten Lernrate kompiliert. Es verwendet `accuracy` als Leistungsmetrik während des Trainings.

```

1 def sample_memories(self, batch_size):
2     # Ensure there are enough samples
3     batch_size = min(batch_size, len(self.replay_memory))
4
5     indices = np.random.permutation(len(self.replay_memory))[:batch_size]
6     cols = [[], [], [], [], []] # state, action, reward, next_state,
7         continue
8     for idx in indices:
9         memory = self.replay_memory[idx]
10        for col, value in zip(cols, memory):
11            col.append(value)
12    cols = [np.array(col) for col in cols]
13    return cols[0], cols[1], cols[2], cols[3], cols[4]

```

Die Methode `sample_memories` dient dazu, eine Stichprobe von Erfahrungen aus der Replay Memory zu entnehmen.

1. **Zufällige Stichprobenauswahl:** Sie erstellt ein Array von Indizes durch zufällige Permutation der im Wiederholungsspeicher verfügbaren Indizes und wählt dann die ersten `batch_size` Indizes aus.
2. **Organisation der Daten:** Die Methode initialisiert eine Liste von Listen zum Speichern von States, Actions, Rewards, nächsten States und der done-Flag
3. **Umwandlung in Arrays:** Jede Liste wird in ein NumPy-Array umgewandelt, was eine effiziente Verarbeitung durch das Neural Network während des Trainings erleichtert.
4. **Rückgabe der stichprobenartigen Daten:** Schließlich gibt die Methode die Arrays zurück, die die Observations, Actions, Rewards und nächsten Observations enthalten.

Alle Werte werden in die gebrauchte Form umgeformt.

```

1     def update_target_model(self):
2         """Copy weights from the online model to the target model."""
3         self.target_model.set_weights(self.model.get_weights())

```

Diese Metode aktualisiert die Weights vom Target Model und setzt sie auf die Weights des zu trainierenden Models.

```

1 def train_model(self):
2     if len(self.replay_memory) < self.batch_size:
3         return
4     mini_batch = self.sample_memories(self.batch_size)
5
6     states = np.array(mini_batch[0])
7     actions = np.array(mini_batch[1])
8     rewards = np.array(mini_batch[2])
9     next_states = np.array(mini_batch[3])
10    dones = np.array(mini_batch[4])
11
12    next_states = next_states.reshape(next_states.shape[0], -1) # Reshape
13      next_states to 2D
14
15    model_params = self.model.trainable_variables
16    with tf.GradientTape() as tape:
17        predicts = self.model(states)
18        one_hot_action = tf.one_hot(actions, self.get_action_space())
19        predicts = tf.reduce_sum(one_hot_action * predicts, axis=1)
20
21        target_predicts = self.model(next_states)
22        target_value = rewards + (1 - dones) * self.discount_factor * np.
23          amax(target_predicts)
24
25        loss = tf.reduce_mean(tf.square(target_value - predicts))
26
27        grads = tape.gradient(loss, model_params)
28        self.optimizer = tf.keras.optimizers.Adam(learning_rate=self.
29          learning_rate)
30        self.optimizer.apply_gradients(zip(grads, model_params))

```

1. **Überprüfung der Größe der Replay Memory:** Zunächst überprüft die Funktion, ob genügend Proben in der Replay Memory vorhanden sind.
2. **Stichprobenentnahme aus der Replay Memory:** Die Funktion entnimmt einen Mini-Batch von Erfahrungen aus der Replay Memory, um diese zu verwenden und aus dem Environment zu lernen.
3. **Umformung der nächsten States:** Die nächsten States werden umgeformt, um sicherzustellen, dass sie im korrekten Format für die Eingabe in das Model sind.
4. **Gradient Tape für automatische Differentiation:** Die Funktion verwendet GradientTape von TensorFlow, um Operationen für die automatische Differenzierung aufzuzeichnen.

5. Modelvorhersagen und Loss Berechnung:

- Es werden die vorhergesagten Q-Werte für die aktuellen Zustände berechnet.
- Aktionen werden in One-Hot-Vektoren umgewandelt, und die entsprechenden vorhergesagten Q-Werte für die ausgeführten Actions werden extrahiert.
- Anschließend werden die Ziel-Q-Werte für die nächsten Zustände berechnet und der Zielwert für jede Erfahrung in der Mini-Batch berechnet.
- Der Loss wird als Mean Squared Error zwischen den vorhergesagten Q-Werten und den Zielwerten berechnet.

6. Gradient und Optimierung:

Loss Gradients in Bezug auf die trainierbaren Parameter des Models werden berechnet. Ein Adam-Optimizer wird verwendet, um diese Gradients auf die Parameter anzuwenden und das Model zu aktualisieren, um den Loss zu minimieren.

Diese Methode wird in der `update`-Methode aufgerufen

```
1 def update(self, state, action, predicted_reward, predicted_next_state):  
2     self.replay_memory.append([state, action, predicted_reward,  
3                                predicted_next_state, False])  
4     self.train_model()  
5     self.update_target_model()
```

Außerdem werden hier die `update_target_model`-Methode aufgerufen und die Observations, die Action, der Vorhergesagte Reward und Observations in die Replay Memory gegeben.

```

1 def save_model(self, directory="saved"):
2     if not os.path.exists(directory):
3         os.makedirs(directory)
4     model_path = os.path.join(directory, 'my_model.keras')
5     target_model_path = os.path.join(directory, 'my_target_model.keras')
6     self.model.save(model_path)
7     self.target_model.save(target_model_path)
8     print("Model saved.")
9
10 def load_model(self, directory="saved"):
11     model_path = os.path.join(directory, 'my_model.keras')
12     target_model_path = os.path.join(directory, 'my_target_model.keras')
13     if os.path.exists(model_path) and os.path.exists(target_model_path):
14         self.model = tf.keras.models.load_model(model_path)
15         self.target_model = tf.keras.models.load_model(target_model_path)
16         print("Models loaded.")
17     else:
18         print("Saved models not found. Starting from scratch.")

```

Letztens hat die DQN Klasse auch eine `save_model` und `load_model` Methode, welche das Target Model und das Trainings Model in das Directory `/saved` speichern. Diese werden im `.keras` File Format gespeichert. In einem `.keras` File werden die Model-Architektur, Weights und Training Configuration gespeichert.

Die `EnvironmentModel` Klasse repräsentiert ein Model der Umgebung im MBRL, das verwendet wird, um den nächsten Zustand und die Belohnung für den gegebenen Zustand und die ausgeführte Action vorherzusagen.

```

1 class EnvironmentModel:
2     def __init__(self, state_size):
3         self.model = Sequential([
4             Input(shape=(6,)),
5             Dense(128, activation='relu'),
6             Dense(128, activation='relu'),
7             Dense(state_size + 1) # Predicting next state and reward
8         ])
9         self.model.compile(optimizer='adam', loss='mse')

```

- Initialisierung:** Der Konstruktor (`__init__`) der `EnvironmentModel` Klasse nimmt die Größe des States und die Action als Eingabeparameter. Diese Größe wird verwendet, um die Output Layer des Models zu definieren.

2. Modelarchitektur:

- Das Model ist ein sequentielles Model (Sequential)
- Es beginnt mit einer Input Layer, die die Form der Eingabedaten angibt. In diesem Fall ist die Input Shape festgelegt auf (6,) (State size + Action).
- Nach der Input layer folgen zwei Dense Layers mit jeweils 128 Neuronen und ReLU-Activation Functions.
- Die Output Layer ist eine Dense Layer mit state_size + 1 Einheiten. Die state_size Einheiten dienen zur Vorhersage des nächsten States und die zusätzliche Einheit zur Vorhersage des Rewards. Diese Einrichtung ermöglicht es dem Model, sowohl den vorhergesagten nächsten States als auch die zugehörige Belohnung aus einem gegebenen Zustand und Action auszugeben.

3. Kompilierung: Das Model wird mit dem Adam-Optimierer und der Mean Squared Error (MSE) Loss Function kompiliert. Der Adam-Optimierer wird aufgrund seiner Effizienz beim Trainieren von Deep-Learning-Modellen gewählt, und die MSE-Loss Function eignet sich für Regression, wie die Vorhersage des nächsten Zustands und der Belohnung.

Diese Funktionen sind Teil des MBRL Models und ermöglichen es dem Model, die Auswirkungen von in einem gegebenen Zustand unternommenen Actions vorherzusagen und aus Erfahrungen zu lernen.

`predict` Funktion

```

1 def predict(self, state, action):
2     action = action.reshape(-1, 1) # Reshape action to 2D
3     input_data = np.concatenate([state, action], axis=-1)
4     prediction = self.model.predict(input_data, verbose=0)
5     predicted_next_state = prediction[:, :-1] # All but the last element
6     predicted_reward = prediction[:, -1:] # Only the last element
7     return predicted_next_state, predicted_reward

```

1. **Umformung der Aktion:** Die Aktion wird in 2D umgeformt, um die Kompatibilität mit den Inputs des Models zu gewährleisten.
2. **Vorbereitung der Eingabedaten:** Observations und Action werden zusammengeführt, um die Eingabedaten für das Model zu bilden.
3. **Vorhersagen machen:** Das Model macht Vorhersagen über die Eingabedaten, einschließlich des nächsten State und der Reward.

4. **Extraktion der Vorhersagen:** Der vorhergesagte nächste State und die Rewards werden von der Vorhersage des Models getrennt.
5. **Rückgabe der Vorhersagen:** Die Funktion gibt den vorhergesagten nächsten State und die vorhergesagte Reward zurück.

train Funktion

```
1 def train(self, states, actions, next_states, rewards):  
2     actions = actions.reshape(-1, 1) # Reshape to 2D if necessary  
3     rewards = rewards.reshape(-1, 1) # Reshape to 2D if necessary  
4  
5     inputs = np.concatenate([states, actions], axis=-1)  
6     outputs = np.concatenate([next_states, rewards], axis=-1)  
7  
8     self.model.fit(inputs, outputs, epochs=10, verbose=0)
```

1. **Umformung von Aktionen und Belohnungen:** Actions und Belohnungen werden auf 2D umgeformt, um den erwarteten Input- und Output des Models zu entsprechen.
2. **Vorbereitung von Eingaben und Ausgaben:** Eingaben werden durch Zusammenführen von Observations und Actions vorbereitet. Ausgaben werden durch Zusammenführen der nächsten Observations und Rewards vorbereitet.
3. **Model training:** Das Model wird anhand der vorbereiteten Inputs und Outputs für eine festgelegte Anzahl von Epochen trainiert. Beim Training werden keine Daten über das Training von Keras ausgegeben (`verbose=0`).

Das EnvironmentModel hat auch eine `save_model` und `load_model` Methode um das Environment Model in / saved zu speichern und zu laden.

```

1 def save_model(self, directory="saved"):
2     if not os.path.exists(directory):
3         os.makedirs(directory)
4     model_path = os.path.join(directory, 'env_model.keras')
5     self.model.save(model_path)
6     print("Environment model saved.")
7
8 def load_model(self, directory="saved"):
9     model_path = os.path.join(directory, 'env_model.keras')
10    if os.path.exists(model_path):
11        self.model = tf.keras.models.load_model(model_path)
12        print("Environment model loaded.")
13    else:
14        print("Saved environment model not found. Starting from scratch.")
```

Der File in dem das EnvironmentModel gespeichert wird ist `env_model.keras`

Model laden

Zu Beginn versucht der Code, vortrainierte Models sowohl für den Agent (`agent.load_model()`) als auch für das Environment Model (`env_model.load_model()`) zu laden. Wenn die Models nicht gefunden werden, wird eine Nachricht ausgegeben, die darauf hinweist, dass das Training von Grund auf neu beginnt.

```

1 try:
2     agent.load_model()
3     env_model.load_model()
4 except:
5     print("Models not found. Starting from scratch.")
```

Trainingszyklus

Der Trainingszyklus wird für eine vordefinierte Anzahl von Episoden (`total_episodes`) ausgeführt. Jede Episode repräsentiert eine Sequenz von Interaktionen zwischen dem Agenten und der Umgebung, beginnend bei einem Anfangszustand bis zu einem Endzustand.

```
1 total_episodes = 10000
```

In diesem Fall trainiert der Agent für 10,000 Episoden.

Interaktion zwischen Agent und Umgebung

Innerhalb jeder Episode läuft eine Schleife, bis das Flag `done` auf `True` gesetzt wird, was das Ende der Episode signalisiert. Der Agent prognostiziert Q-Values für den aktuellen Zustand und wählt die Action mit dem höchsten Q-Value. Die ausgewählte Action wird dann in der Umgebung ausgeführt, was zu einem neuen Zustand, einer Reward und einer Aktualisierung des `done`-Flags führt.

```

1 for episode in range(total_episodes):
2     state = env.reset()
3     done = False
4     while not done:
5         q_values = agent.model.predict(state.reshape(1, -1), verbose=0)
6         action = np.argmax(q_values[0])
7         next_state, reward, done, _ = env.step(action)

```

Training des Environment Models

Das `EnvironmentModel` wird mit der Observation Transition (`state, action, next_state, reward`) trainiert, um den nächsten State und den Reward angesichts eines States und einer Action vorherzusagen.

```

1 # Train environment model
2 env_model.train(np.array([state]), np.array([action]), np.array([next_state]),
3                  np.array([reward]))

```

Planung mit dem Environment Model

Der Agent verwendet das `EnvironmentModel`, um das Ergebnis seiner Actions vorherzusagen, bevor er sein DQN mit den vorhergesagten Ergebnissen aktualisiert. Dieser Ansatz führt zu einem effizienteren Lernen, indem zukünftige Schritte simuliert werden, ohne tatsächliche Interaktion auszuführen.

```

1 # Use environment model for planning
2 predicted_next_state, predicted_reward = env_model.predict(np.array([state]),
3                                                               np.array([action]))

```

Aktualisierung des DQN Agents

Das DQN des Agenten wird mit den vorhergesagten Ergebnissen aktualisiert, um das Verständnis des Agents für die Dynamik der Umgebung und seinen Entscheidungsprozess zu verbessern.

```

1 # Update DQN with the predicted outcomes
2 agent.update(state, action, predicted_reward, predicted_next_state)
3 state = next_state

```

Abschluss der Episode und Model-speicherung

Nach Abschluss einer Episode wird die Belohnung in `agent.rewards_history` protokolliert und Informationen über die Episode und die Reward werden ausgegeben. Die Models werden regelmäßig pro `save_interval` (50) gespeichert, um den Fortschritt zu sichern.

```

1 if done:
2     agent.rewards_history.append(reward)
3     print(
4         "Episode: {} / {}, reward: {}".format(episode, total_episodes, reward
5             ))
6     break
7
8 if episode % save_interval == 0:
9     agent.save_model()
    env_model.save_model()

```

Unterbrechungsbehandlung

Der Code ist in einen try-except-Block eingewickelt, um Unterbrechungen zu behandeln. Wenn das Training vorzeitig gestoppt wird, werden die Models gespeichert und eine Zusammenfassung der Belohnungen über die Episoden hinweg angezeigt. Dies garantiert, dass man zu jeder Zeit das Training stoppen kann, ohne Fortschritt zu verlieren.

```

1 except KeyboardInterrupt:
2     agent.save_model()
3     env_model.save_model()
4     print("Training stopped. Models saved.")

```

Visualisierung

Nach dem Training werden zwei Diagramme generiert, um die Reward History und den Loss über die Zeit des Trainings zu visualisieren, was hilft, die Leistung und den Lernfortschritt des Agenten zu bewerten. Hierfür wird beim ersten Graphen die Rewards über die Episodes geplottet und beim Zweiten, der Loss des Agents über die Steps.

```
1 rewards = agent.rewards_history
2 losses = agent.loss_history
3 episodes = range(len(rewards))

4
5 plt.plot(range(len(agent.rewards_history)), agent.rewards_history)
6 plt.xlabel('Episode')
7 plt.ylabel('Reward')
8 plt.title('Reward per Episode')
9 plt.show()

10
11 plt.plot(range(len(agent.loss_history)), agent.loss_history)
12 plt.xlabel('Step')
13 plt.ylabel('Loss')
14 plt.title('Loss per Episode')
15 plt.show()
```

Solche Graphen sind in der Abbildung 3.24 und 3.23 zu sehen.

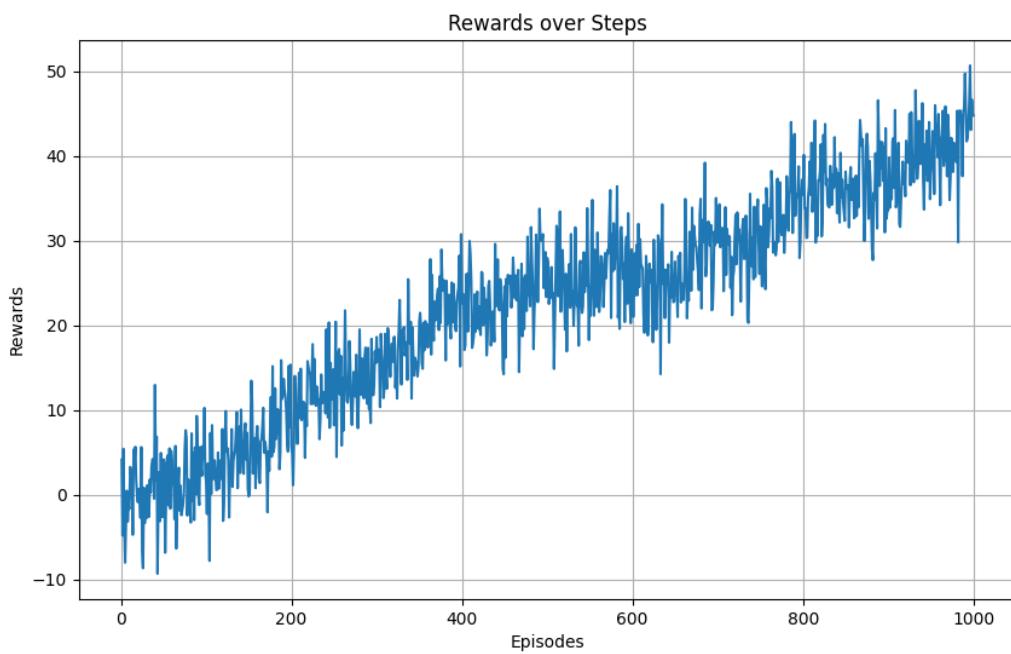


Abbildung 3.23: Rewards over Steps

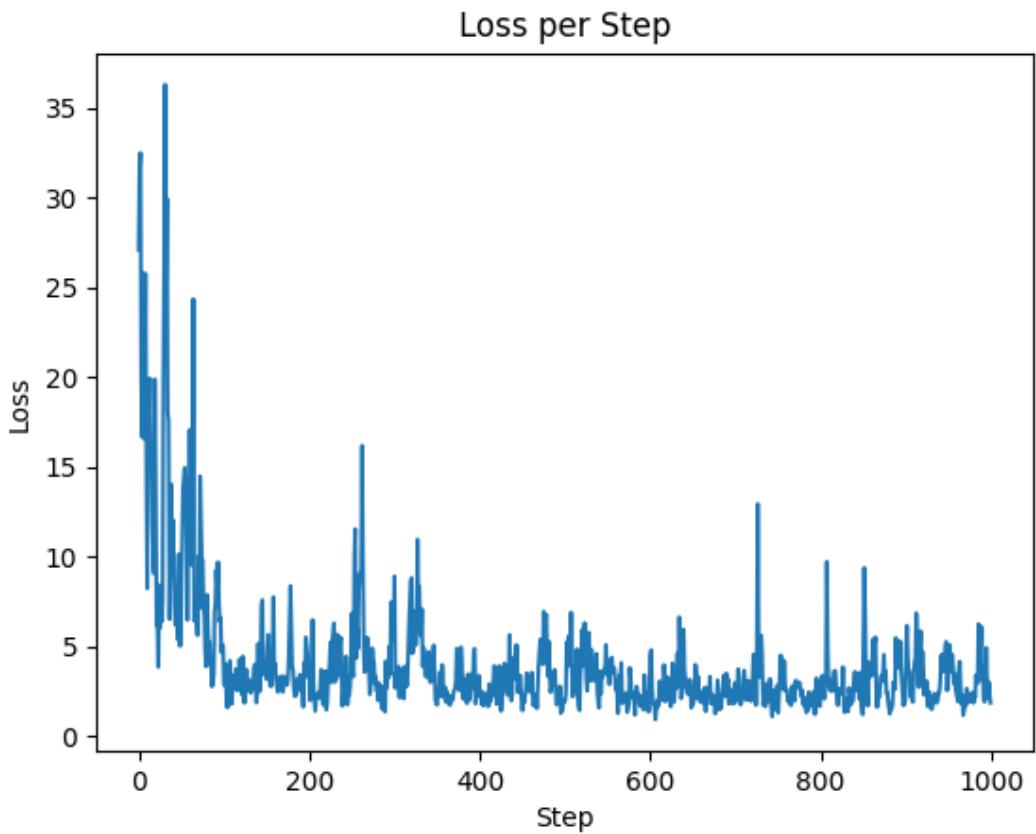


Abbildung 3.24: Loss over Steps

Wie man auf den Graphen sehen kann hat der Agent nach ungefähr 1000 Episodes das Environment gelernt, da sich der Loss nach 800 Episodes nicht mehr viel verändert und der Episode Reward immer ansteigt. Auf dem Graphen in Abbildung 3.24 kann man den Lernprozess gut erkennen, da der Loss über die Zeit immer weniger geworden ist und sich dann nach einer 900 Steps im Bereich < 5 bleibt.

Durch diese Implementierung des Agents wurde der DQN Agent mit einem MBRL Algorithmus verwendet. Es wurde ein Dynamics Model verwendet, welches den nächsten State und die Reward voraussagt, um das Planning des Agents durchzuführen, dies ermöglicht eine viel bessere Sample Efficiency für die sehr rechenintensiven MBRL Algorithmen. Der Agent hat sehr schnell das Environment gelernt und schneidet auch gut in der simplen Umgebung ab. Die Verwendung des DQN Agents mit Model Based Planning verbessert den Lernprozess und kann auch mit wenigen Rewards effizient das Environment lernen und trainieren.

Ein Vorteil der Kombination von DQN und dem Environment-Modell liegt in der verbes-

sernen Robustheit des Agents gegenüber Umgebungen mit geringen Rewards. Denn auch wenn das Environment geringe oder keine Rewards gibt, können durch das Dynamics Model auch wichtige Informationen gelernt und herausgenommen werden, um das Training und Lernen voranzutreiben. Außerdem kann der Agent auch effektiver auf unvorhergesehene oder komplexe Situationen reagieren und sein Verhalten entsprechend anpassen.

Immer öfter werden Model Based Algorithmen mit Model Free Algorithmen kombiniert, um das Beste von beiden zu bekommen und Agents zu optimieren. Die Sample Efficiency und Dynamics Models von Model Based Agents und die on-policy Trainings in Environments der Model Free Agents werden Kombiniert um das Environment, Sample Efficient, zu lernen und eine gute Policy zu erreichen.

für mehr über die Kombination von Model Based und Model Free Algorithmen , siehe: [MM:Web43], [MM:Web44]

3.2 Projektbezug

Das Projekt LiCAR zielt darauf ab, autonomes Fahren mithilfe eines AI Agents, welcher die Daten eines LiDAR Sensors, und Pathfinding einsetzt, zu erreichen. Die Implementierung des AI Agents ist der SAC Agent, welcher Observations über das Environment, welches der Carla Simulator ist, bekommt. Neben der im Projekt angewandten Methode, des SAC Agents, kann das autonome Fahren auch mit einem dem in dieser Diplomarbeit Implementierten Ansatz des DQN Agents mit Model Based Planning erreicht werden. Dem Agent, welcher das Fahrzeug kontrolliert, würde ein Dynamic Model des Environments übergeben werden, wodurch es das Environment schneller lernen kann, da Autofahren ein sehr komplexer Vorgang ist. Durch die vielen Observations die bei dem Autofahren und für das autonome Fahren aufkommen:

- Lidar Daten (Polar)
- Throttle und Steering des Autos
- Position des Autos
- Position der nächsten 2 Nodes, welche im Simulator am kürzesten Pfad liegen

Deshalb wäre es sinnvoll durch MPC und Dynamic Models die Sample Efficiency zu erhöhen und auch das Environment zu lernen.

Kapitel 4

Model Free Reinforcement Learning

4.1 Einleitung

Model Free Reinforcement Learning ist ein Ansatz beim Machine Learning, um einen Agenten direkt mithilfe dessen Erfahrung zu trainieren, ohne dass dieser jegliches Wissen über seine Umwelt hat. Dieser Agent soll mithilfe von Belohnungen und Bestrafungen lernen.

4.1.1 Hintergrund und Kontext

Der Ursprung des Model Free Reinforcement Learning kann bis ins späte 19. Jahrhundert zurückverfolgt werden, als Edward Thorndike das „Law of Effect“ vorschlug. Das Gesetz besagt, dass „Aktionen mit positiven Auswirkungen in einer bestimmten Situa-

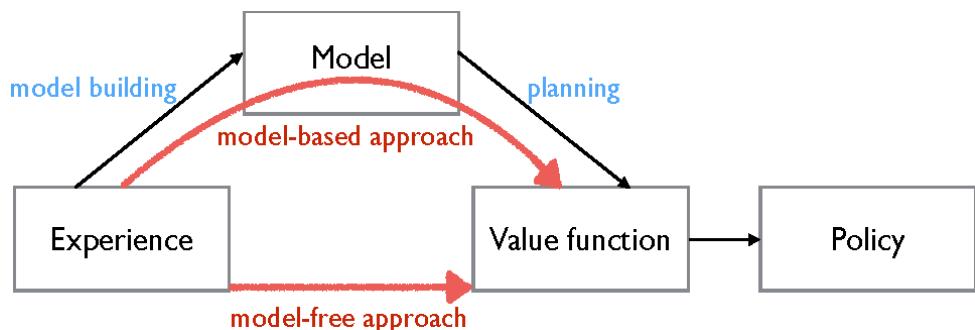


Abbildung 4.1: Vergleich: Model Free vs. Model Based
[AK:Web03]

tion in dieser Situation wieder auftreten, und Reaktionen, die negative Auswirkungen haben, werden in Zukunft weniger wahrscheinlich.“

Im nachfolgenden Experiment bewies Thorndike, dass eine Katze ein Puzzlebox entkommen konnte. Dies war möglich, da die Katze für jeder dessen Aktionen in irgend-einer Weise belohnt wurde und davon die Konsequenzen erlitt. Und im Laufe der Zeit hat die Katze diese Aufgabe immer geschickter geschafft. Im Englischen wird dieses Verfahren auch oft als "positive reinforcement" genannt.

[AK:Web03]

4.1.2 Bedeutung des Model Free Reinforcement Learning

Model Free Reinforcement Learning wird häufig in Bereichen eingesetzt, in denen die Modellierung der Umgebung schwierig ist oder die Umgebung dynamisch ist und sich mit der Zeit ändern kann. Beispiele dafür sind Spielstrategien, Robotik, autonomes Fahren und viele Ähnliche.

4.2 Konzepte der Model Free Reinforcement Learning

4.2.1 Einleitung

Im Gegensatz zur Model Based, werden bei Model Free Algorithmen keine Transition-Probabilities sowie keine Reward-Functions geschätzt. Die beiden werden oft gemeinsam als Modell der Umwelt genannt und werden bei Model Free vom Agenten gelernt indem es Actions durchführt.

Die Vorgehensweise von Model Free Reinforcement Learning kann auch als Trial and Error betrachtet werden. Häufig verwendete Funktionen für MFRL sind Q-Learning und die Monte Carlo Methoden.

[AK:Web04]

4.2.2 Erklärung

Im Model Free Reinforcement Learning ist die Approximation von Monte Carlo (MC) ein zentraler Bestandteil einer großen Klasse modellloser Algorithmen. Ein Agent, der mit dieser Methode trainiert wird, muss durch den eigenen Handlungen lernen, welche Actions richtig und welche falsch sind. Die Richtigkeit der Actions wird dem Agenten mithilfe von sogenannten Rewards kommuniziert.

[AK:Web04]

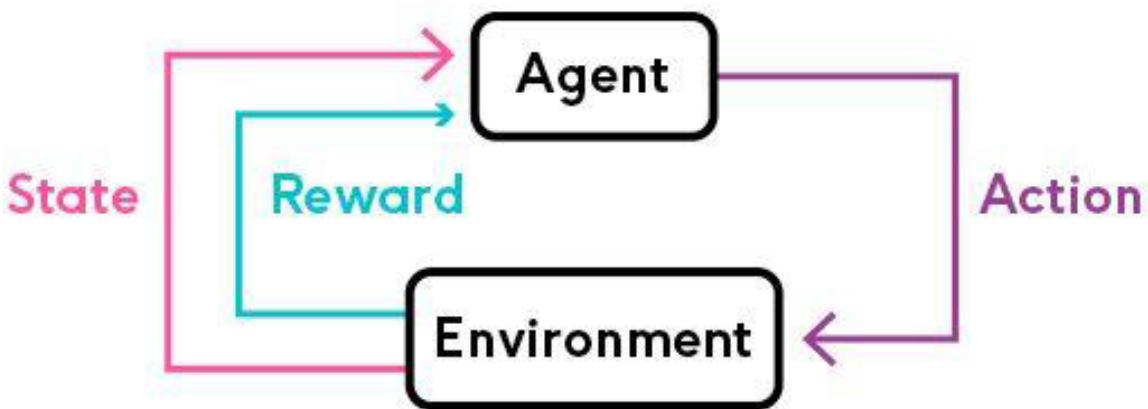


Abbildung 4.2: Reinforcement Learning
[AK:Web02]

4.2.3 Konzepte

Policy

Model Free Algorithmen können Policy-based oder Value-based sein. Jede Policy hat 2 entsprechenden Value-Functions, nämlich die Q-Value und die V-Value. Eine Policy zählt als besser, wenn deren V-Value höher ist. Zum Beispiel, wenn die Value-Function von Policy Y höher ist, als die von Policy X, dann zählt Policy Y als der bessere Policy. Nach entsprechend genügend Zeit erreicht der Agent hoffentlich den Optimalen Policy.

[AK:Web01]

Value-Based vs Policy-Based Algorithmen

Value-based Algorithmen finden die optimalen Zustandswerte. Die optimale Policy kann dann daraus abgeleitet werden. Policy-based Algorithmen brauchen nicht den optimalen Wert und finden die Optimal Policy direkt.

[AK:Web01]

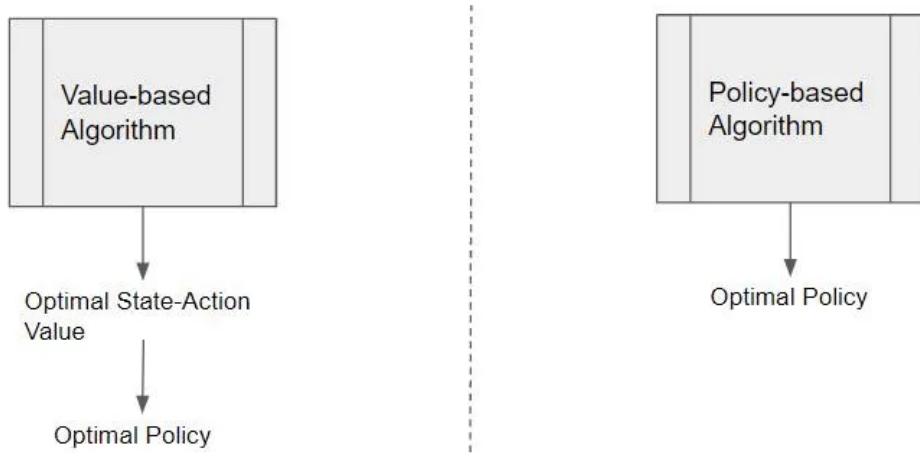


Abbildung 4.3: Value vs Policy
[AK:Web01]

Optimal Policy

Wenn man dieses Verfahren ständig nachverfolgt, dann wird man in der Lage sein, die sogenannte beste Policy zu finden.

Dieser soll besser als alle anderen sein und daher auch optimal für den Nutzungsfall. Die Optimal Policy hat zwei Value-Functions, welche besser als alle anderen sind. Im folgenden Bild kann betrachtet werden, dass die S1 bis S5 der Policy Y höher sind als die von Policy X. Es zählt daher als der bessere Policy und wird bevorzugt gegenüber Policy X. Wenn man die Optimal Policy findet, dann hat man auch indirekt die Optimal State Value gefunden.

[AK:Web01]

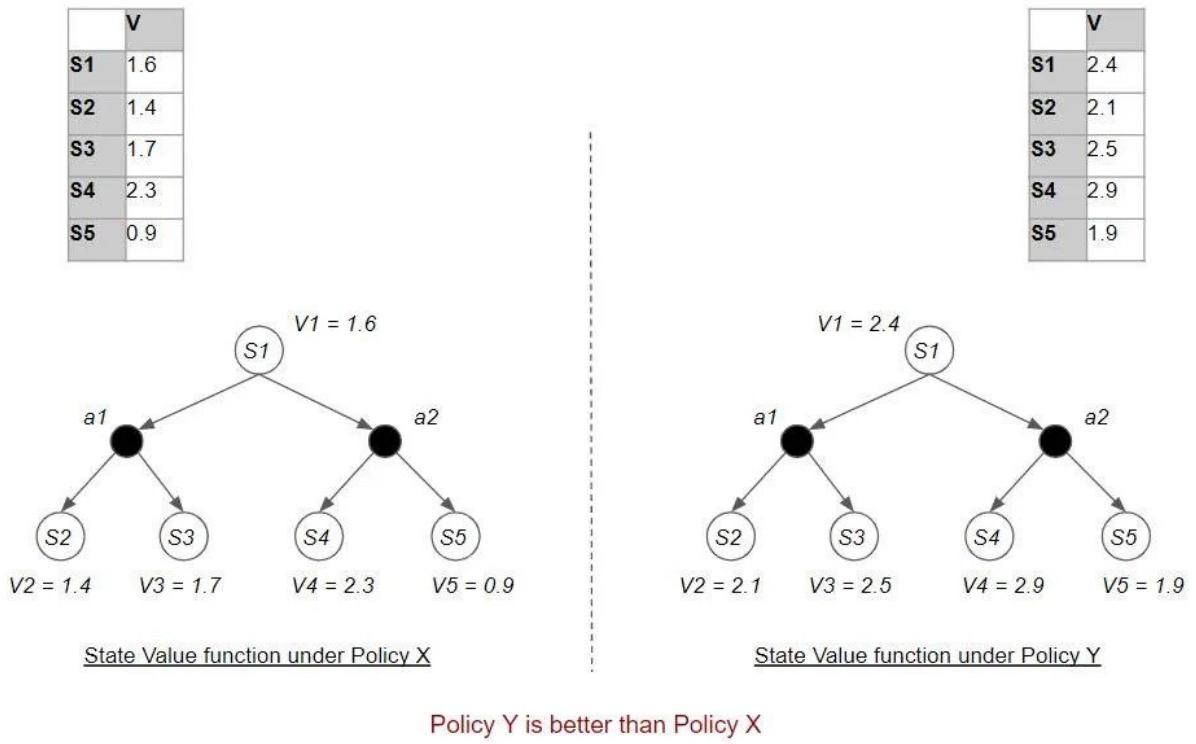


Abbildung 4.4: State Value Vergleich
[AK:Web01]

Angewandter Algorithmus

Der Model Free Ansatz verwendet einen iterativen Algorithmus basierend auf Actions und eine dazupassende Rückmeldung von der Umgebung (Environment). Auf einer höheren Stufe haben Value-Based und Policy-Based Algorithmen insgesamt vier Basisoperationen, die sie ausführen. Sie verwenden am Anfang arbiträre Approximationen und diese werden dann im Laufe des Trainings verbessert.

In 4.5 wird ein vereinfachter Algorithmus gezeigt, wie so ein Agent daher handeln kann. Dieser ist ein Value-Based Algorithmus und daher versucht es, mithilfe des optimalen Value, die optimale Policy zu finden.

[AK:Web01]

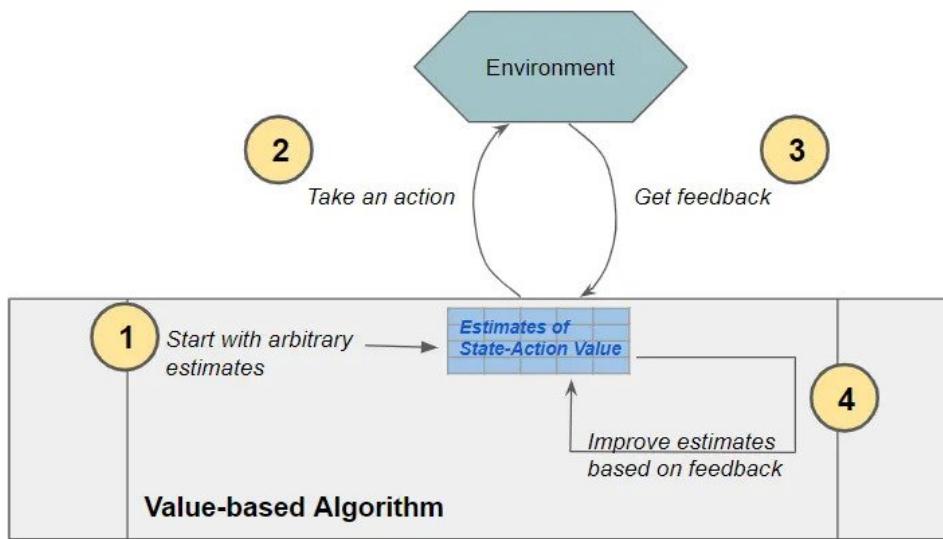


Abbildung 4.5: Algorithmus illustriert
[AK:Web01]

- **Beim ersten Schritt** werden alle Werte auf 0 gesetzt, da der Agent nicht weiß, welche Werte richtig sind.
- **Beim zweiten Schritt** wird eine Aktion ausgeführt. Da der Agent keine Ahnung über die Richtigkeit seiner Aktionen zur Zeit hat, wird er alle Optionen ausführen um die beste Option zu finden.
- **Beim Dritten Schritt** bekommt der Agent eine Reward abhängig von seinen Aktionen. Dieser Reward wird dann im nächsten Schritt gebraucht.
- **Beim Letzten Schritt** werden die Approximationen auf Hinsicht der Rewards verändert, um die zukünftigen Aktionen zu verbessern

Die verwendeten Approximationen können mithilfe von drei Faktoren verbessert werden. Die Gewichtungen der drei Faktoren: Frequency, Depth und der Formula unterscheiden sich zwischen den verschiedenen Algorithmen.

[AK:Web01]

Frequency: Anzahl der Vorwärtsschritte vor dem Update

- Episode: Bis zum Ende der Episode werden Aktionen ausgeführt und die Beobachtungen gespeichert, um die Approximationen zu verbessern
- One: Die Approximationen werden nach einem Schritt schon verbessert

- N: Update nach n-Schritten

Depth: Anzahl der zurückverfolgten Rückschritte, bevor ein Update eingesetzt wird

- Episode: Alle Schritte, die bisher in der Episode gemacht wurden
- One: Nur den jetzigen State-Action Pair
- N: Ein Update nach jedem N-ten Schritt

Formula: wird verwendet, um die aktuellen Approximationen zu berechnen und die Policy zu evaluieren (bewerten)

- TD Error: Der Q-Value wird mit einem Error erweitert, der Error hängt von dem jetzigen und dem nächsten Q-Value ab.

Vergleich der Model Free Algorithmen

Obwohl sie am Anfang sehr verschieden vorkommen, hat jedes Algorithmus eine bestimmte Vorgehensweise. Der größte Unterschied zwischen ihnen ist nur der Weg, wie sie die einzelnen Faktoren einsetzen und betonen. Deren Output unterscheidet sich aber auch.

[AK:Web01]

Noch ein weiterer wichtiger Faktor beim Wahl der Algorithmus ist der Nutzungsfall. Manche Algorithmen funktionieren besser in limitierten (discrete) und manche in komplexeren (continuous) Action Spaces. Der Hauptunterschied zwischen den beiden ist, dass bei continuous Action Spaces Dezimalwerte erlaubt werden.

| | Output | Frequency | Depth | Formula |
|-------------------|------------------------------------|--------------------|---------|--------------------------------|
| Prediction | Monte Carlo Prediction | State Value | Episode | Return Error |
| | TD(0) | State Value | One | TD Error |
| Control | TD(λ) | State Value | One | TD Error with weighted Returns |
| | Monte Carlo Control | State Action Value | Episode | Return Error |
| | Sarsa ie. TD Control | State Action Value | One | TD Error |
| | Sarsa(λ) | State Action Value | One | TD Error with weighted Returns |
| | Q Learning | State Action Value | One | TD-max Error |
| | Deep Q Networks | State Action Value | One | TD-max Error |
| | Policy Gradient | Policy | Episode | Return-based Loss |
| | Actor Critic | Policy | N | Advantage-based Loss |

Abbildung 4.6: Vergleich MFRL Algorithmen
[AK:Web01]

4.3 Grundlagen des Reinforcement Learnings

4.3.1 Markov-Decision-Process

Der MDP ist ein mathematischer Prozess, welcher im Reinforcement Learning zur Modellierung von Entscheidungsproblemen verwendet wird. Es besteht aus Zuständen, Actions, Rewards und Übergangswahrscheinlichkeiten. Die Lösung von einer MDP ist eine Funktion, die zu jedem State die Action ausgibt, die den Gewinn über die Zeit maximiert. Bekannte Lösungsverfahren sind unter anderem das Value-Iteration-Verfahren und Reinforcement Learning.

Ziel von MDP

Der Ziel von MDP ist es, eine gute Policy für den Entscheidungstreffer zu finden. Eine Funktion $\pi(s)$, der Abhängig vom State s eine passende Wahl trifft. Das heißt, einen Policy zu finden, der den kumulierten Wert der Rewards maximiert. In Reinforcement Learning werden MDPs mittels Simulation angewendet.

Formel für Optimal Policy

Folgende Formel wird verwendet, um die Optimale Policy zu finden:

$$E \left[\sum_{t=0}^{\infty} \gamma^t R_{a_t}(s_t, s_{t+1}) \right]$$

- γ befindet sich im Intervall [0;1] und bezeichnet den Discount Faktor
- Als $a(t)$ wird der Action von unserer Policy eingefügt; $\pi(s)$

Als Ergebnis dieser Formel wird der Optimal Poliy π^* erhalten. Ein MDP kann aber auch mehrere Optimalen Policies besitzen.

[AK:Web06]

4.3.2 Reward-Functions

Wie schon vorher erwähnt, geht es beim Reinforcement Learning darum, eine Optimaile Policy zu finden, der den Reward maximiert und auch in den meisten Fällen auch recht schnell ist. Ein einfacher Agent handelt mit der eigenen Umgebung in bestimmten diskreten Zeitintervallen.

Bei jedem Zeitschritt bekommt der Agent den derzeitigen State der Umwelt und eine Reward. Danach wird eine Action aus dem Set der möglichen Actions gewählt und in der Umwelt ausgeführt. Beim Machine Learning ist es gut, wenn es zumindest zwei Reward Functions gibt.

Einen, der bei jedem Zeitschritt dem Agenten gegeben wird und einen anderen Reward Function, der für den Agenten als finalen Reward fungiert (zum Beispiel wenn der Agent vom Reset betroffen wird). Mit dem anderen Function sollte der Agent daher dann einen finalen Bewertung deren Handlung erhalten und damit weiter lernen.

4.3.3 Value-Functions

Ein Value Function versucht es, eine Policy zu finden, der den Reward mit den wenigen Discounts $E(G)$ ermittelt. Dies geschieht mithilfe eines Sets, der eine Menge

von erwarteten Rewards mit Discounts beinhaltet für entweder den jetzigen (on-policy Algorithmen) oder den optimalen (off-policy Algorithmen).

Der State Value einer Policy kann mit der folgenden Formel definiert werden. Unter G wird der Reward mit Discounts gemeint, welcher vom State s und Policy π abhängt:

$$V^\pi(s) = E[G \mid s, \pi]$$

Der höchstmögliche State Value V^* kann dann so berechnet werden. Bei dieser Formel ist es aber auch möglich, dass π sich verändert:

$$V^*(s) = \max_{\pi} V^\pi(s)$$

Daher versteht man unter Optimal Policy eine Policy, die in jener State die Optimale State Value Function erreicht. Damit weiß man auch, dass eine zufällig ausgewählte State mit einer optimalen Action gefolgt wird.

[AK:Web08]

4.4 Exploration-Exploitation dilemma

Grundsätzlich geht es bei diesem Dilemma darum, ob der Agent schon bereits bekannte gute Optionen ausnutzt (Exploitation) oder ob es unbekannte Optionen ausprobiert, um vielleicht bessere Optionen zu finden (Exploration).

4.4.1 Balancieren von Exploitation und Exploration

Die beiden Prinzipien basieren sich auf eines der simpelsten Austauschprinzipien in der Natur. Ein Musiker hat die Option, sich entweder auf die Gitarre zu fokussieren wo er dies schon seit mehreren Jahrzehnten praktiziert oder einen neuen Instrument zu erlernen.

Zum einen bekommt der Musiker die Fähigkeit, einen neuen Instrument spielen zu können.

nen, gibt aber dafür Zeit ab. Zeit, womit man sich eher an seinen Stärken fokussieren könnte, um sich in dem jeweiligen Feld besser zu verbessern.

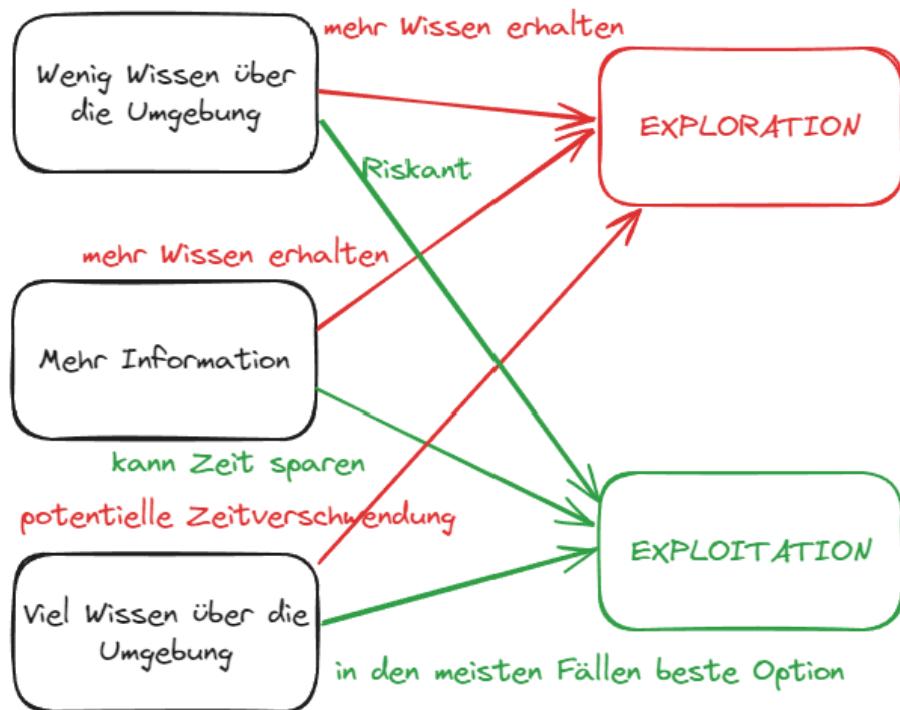


Abbildung 4.7: Tendenz zur Exploration bzw. Exploitation abhängig vom jetzigen Wissensstand

Ähnlich funktioniert es in Reinforcement Learning. Grundsätzlich ist es wichtig, die beiden Faktoren ausgewogen zu berücksichtigen. Bei Exploitation entsteht das Risiko, dass der Agent nicht alles über seine Umwelt weiß und dadurch "falsch" lernt.

Bei Exploration gibt es folgende Faktoren zu betrachten: Der Agent könnte zwar bessere Optionen erlernen, gibt aber dafür eine Chance der Exploitation ab. Dadurch ist die Ausgewogenheit dieser Faktoren ein wichtiger Aspekt beim erschaffen eines Agenten, der für längere Zeit handeln soll.

Dieser Modell versucht, sich zwischen **Energiegewinnung (Exploitation)** oder **Wissensaneignung (Exploration)** zu entscheiden. Die angewandte Strategie zeigt den Prozentsatz der Zeit, den das Subjekt während seiner Lebensspanne T max für die Wissensaneignung aufwendet, abhängig von der Zeit. Somit schaut die Modell, wenn man die Energie mit E , den sogenannten Wissen mit L und die Zeitabhängige Strategie mit $u(t)$ bezeichnet, so aus:

$$\frac{dE}{dt} = \frac{f_{max}L}{K_L + L} - m - u(t), \quad \frac{dE}{dt} = \frac{f_{max}L}{K_L + L} - m - u(t)$$

Diesem Modell nach, wird die Energie E als eine Art Sättigungsfunktion des vorhandenen Wissens L betrachtet. Dabei spielt die halbsättigende Konstante kL eine entscheidende Rolle. Diese Konstante bewirkt, dass die Aufnahme von Wissen bei bereits höherem Wissensstand mit geringerem Energiegewinn verbunden ist. Es entsteht also ein Gegensatz, wobei die Energieaufwendung für Wissensaneignung bei zunehmendem vorhandenem Wissen abnimmt.

[AK:Web09]

4.4.2 Exploration Strategies

Bei Exploration Strategies wird eine Methodik gewählt, wo eine Balancierung der Exploration und Exploitation stattfindet und damit die Vorgehensweise dem Agenten erleichtert wird. Damit wird die Entscheidung zwischen den zwei Optionen nur mehr vom jetzigen Basis des Wissens abhängig.

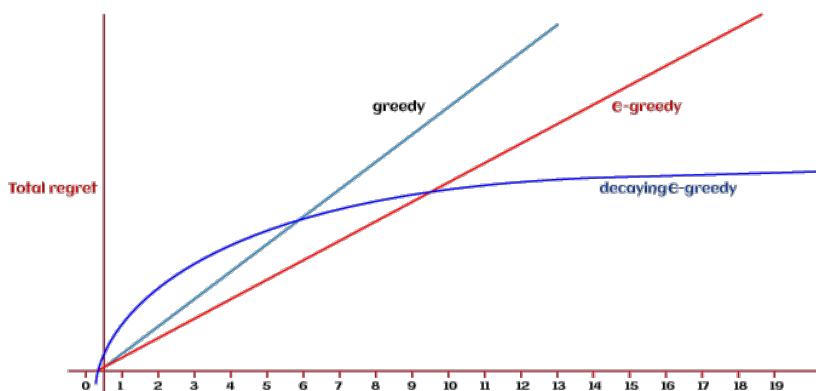


Abbildung 4.8: Vergleich Greedy, E-Greedy und Decaying-E

Greedy Strategy

Bei Greedy Strategies wird immer die für den besten gehaltene Action gewählt, aufgrund von der jetzigen Wissensbasis. Diese Strategie ist hauptsächlich 0% Exploration und 100% Exploitation. Ohne Exploration kommt man in eine State, wo nicht mehr

weitertrainieren kann, wenn nur suboptimale Actions gewählt werden. Dabei wird die Action gewählt, wo der höchste Reward erreichbar ist.

[AK:Web10]

Epsilon-Greedy

"Pick the action assumed to be the best in some cases, explore randomly otherwise"
 Bei e-Greedy wird zwischen zwei Optionen gewählt, die Action mit der höchsten Reward oder eine komplett zufällige neue Action, welcher potenziell besser sein kann. Der Wert von ϵ deutet dann darauf, welche gewählt wird. Ein ϵ Wert von 0 steht für volle Exploitation und ein Wert von 1 steht dann für volle Exploration. Epsilon-Greedy hat daher das Problem, dass in manchen Fällen der Agent nur unendlich exploren kann. Bei Epsilon Greedy ist der Epsilon Wert konstant.

[AK:Web09]

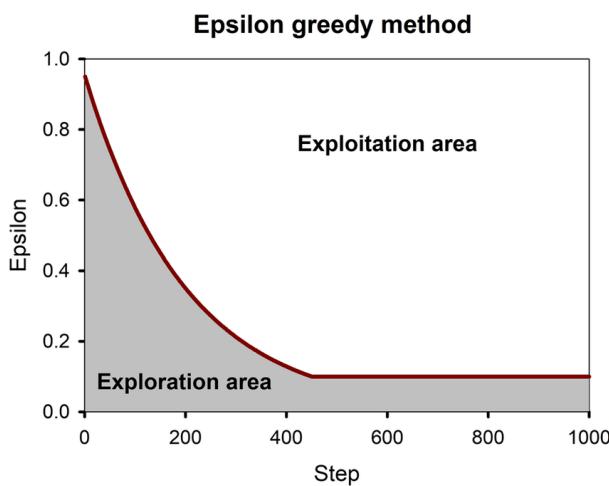


Abbildung 4.9: Epsilon Greedy
 [AK:Img01]

Decaying Epsilon-Greedy

Um Epsilon-Greedy zu optimieren, kann ein Decaying Epsilon-Greedy im Training eingeführt werden. Dabei wird mit der Exploration aufgehört, sobald die beste Option identifiziert wurde. Diese Strategie ist etwas schwieriger einzuführen und wird in der Praxis öfter als ein Epsilon-Greedy mit einer decaying (mit der Zeit sinkender) ϵ anstatt einer vorbestimmten Wert realisiert.

[AK:Web10]

Im 4.9 ist es sichtbar, wie ein Decaying Wert von Epsilon (y-Achse) sich mit der Zeit (x-Achse) ändert. Dabei wird im Training mehr Fokus auf die frühe Exploration und die spätere Explotation gelegt.



Abbildung 4.10: Decaying Epsilon Beispiel

Upper Confidence Bound

"Pick the action with the highest upper bound estimate of the reward"

Mit dieser Methode wird eine Unsicherheit des Wertes modelliert mit entweder Confidence Intervals oder Probability Distributions. Dabei wird unsere jetzige Wissensstand mit der relativen Unsicherheit modelliert und angewendet, um die Exploration zu verbessern. Bei UCB wird die Zeit, der mit der Exploration verbraucht wird, reduziert. Damit senkt die Unsicherheit. Im Gegensatz zur Decaying Epsilon, ist diese Strategie etwas flexibler, da es weniger Fine-Tuning braucht, unabhängig vom angewandtem Modell.

[AK:Web10]

4.5 Methoden des Machine Learnings

4.5.1 Monte-Carlo-Methoden

Die Monte Carlo Methode bezieht sich auf eine Methode, welche mithilfe von Random Sampling numerische Ergebnisse liefert. Der Grundkonzept von diesem ist es, einen Problem mit Zufall zu lösen, wobei das Problem im Grunde deterministisch sein könnte. Sie wird in der Physik und in der Mathematik angewendet, vor allem in Fällen, wo andere Möglichkeiten sich als schwer oder unmöglich erweisen. Angewendet findet man diesen in der Optimierung, Numerische Integration und Probability Distribution.

[AK:Web11]

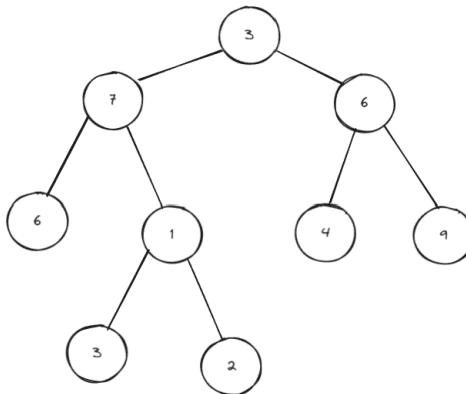


Abbildung 4.11: Selbsterstellter Suchbaum

Monte Carlo Tree Search

Einen weiteren Einsatz von Monte Carlo Methoden findet man in Spielen, und zwar unter dem Namen Monte Carlo Tree Search. Dabei wird ein Suchbaum mit den möglichen Actions erstellt. Es wird ein Action gewählt und man folgt den Baum weiter und weiter, damit ist es möglich, den relative Wert eines jeden Actions zu ermitteln.

Wie im Baum ersichtlich, werden solche Suchbäume mit dem relativen Wert der Action beschriftet. Beim Anwendung dieser Methode kommt auch ein Einsatz von Exploration vs Exploitation. Dabei muss eine Action mit einem guten Wert gewählt werden, aber potenziell gibt es noch bessere Actions im Baum. Diese Methode wurde schon angewendet, um mehrere Brettspiele(Hex, Go) und nicht-deterministische Spiele (wie Poker, Catan, Magic: The Gathering) zu lösen.

Ein bekannter Beispiel für MCTS ist der von Google Deepmind entwickelte KI AlphaGo, welcher einen Meistertitel in Go bekam, nachdem er einen Meister 4 zu 1 geschlagen hat. Dieser KI verwendet Monte Carlo Tree Search mithilfe von Artificial Neural Networks um den besten Zug zu wählen, und teilweise in die Zukunft vorauszuschauen. Für eine grafische Darstellung, siehe 4.12

[AK:Web12]

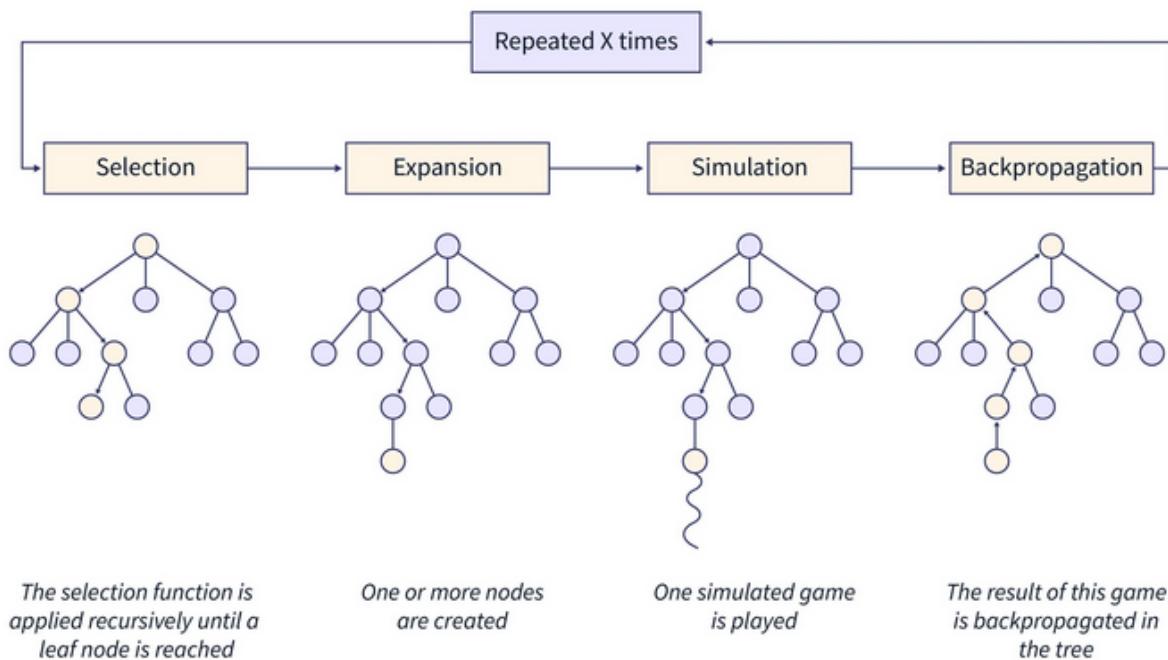


Abbildung 4.12: MCTS Methoden
[AK:Web13]

Beim Nutzen dieser Struktur werden mehrere Strategien parallel getestet, nicht nur der beste, eine Ableitung vom Exploration und Exploitation. Beim Exploration werden unbekannte Teile des Baumes entdeckt, welcher potenziell zu einer besseren Strategie verleitet. Das heißt, je mehr entdeckt wird, desto breiter wird der Baum. Exploitation wird daher dann nur an dem Knoten ausgeführt, wo der höchste Action-Value möglich ist. Die Upper Confidence Bound (UCB) wird auch angewendet, um die Selektion zu vereinfachen.

Folgender Formel gilt dabei:

$$S_i = X_i + C \sqrt{\frac{\ln(t)}{n_i}}$$

$S_i \rightarrow$ Wert von Node i

$X_i \rightarrow$ empirischer Mittelwert von Node i

$C \rightarrow$ Konstanter Wert
 $t \rightarrow$ Anzahl der Simulationen

4.5.2 Episodenbasiertes Lernen

Bei dieser Methode lernt ein Agent mithilfe von mehreren Episodes. Eine Episode beschreibt eine Sequenz des Agenten in ihrer Umwelt. Jede Episode hat einen Startzustand und einen Endzustand und besteht aus Steps. Ein Step beschreibt dabei eine Action, den ein Agent vorgenommen hat und ein Step endet nachdem eine bestimmte Bedingung erfüllt wird wie zum Beispiel, wenn ein Auto so lange fährt bis er eine Mauer anfährt. Episodes können dabei auch auf mehreren Wegen den Endzustand erreichen. Ein möglicher Ansatz wäre, eine positive und eine negative Möglichkeit zu haben.

4.5.3 Bewertung von Policies

Der Ziel von diesem Vorgang ist es, den Wert einer Policy zu ermitteln, das heißt, wie viel Reward (Wert des State-Value Function) von einer Policy zu erwarten ist.

Dabei können sich Agents folgendermaßen unterscheiden:

- **Sequential:** Der Agent bekommt verzögerte Information. Als Problem erweist sich die Tatsache, dass es schwierig zu ermitteln ist, woher der Reward eigentlich stammt.
- **Evaluative:** Der Feedback ist nur noch mehr relativ, da die Umgebung nicht ganz bekannt ist. Dabei erhebt sich Exploration und Exploitation, da der Agent nichts über die Transition in die nächste State weiß.
- **Sampled:** Der Agent generalisiert den gesammelten Feedback und trifft eine intelligente Entscheidung basierend auf diese Generalisierung,

Um die Bewertung zu ermöglichen werden spezielle Algorithmen angewandt, einige davon sind:

- **Monte Carlo**
- **Temporal Difference (TD):** Mit TD ist es möglich, Schätzungen nach sehr wenigen Steps zu machen. Wenn der Agent schon einen Step gemacht hat, dann

kann man auf den nächsten State Beobachtungen machen, um die Werte der State-Value Function zu approximieren. Es nutzt den vorigen State-Value, um die nächste nur mithilfe der Observations zu approximieren.

- **N-step TD:** Ähnlich wie bei TD, bei N-Step passieren die Approximationen nach einer bestimmten Anzahl an vergangenen Steps.

[AK:Web18]

Vorteile und Einschränkungen

Natürlich gibt es bei dieser Methode viele Vorteile aber auch Nachteile. Einige der nennenswerten Vorteile:

- **Effizienz:** Durch die Bewertung von Policies kann der Agent herausfinden, welches seiner Actions in verschiedenen Situationen am sinnvollsten sind, ohne dass er alle ausprobiert. Das macht den Trainingsprozess schneller.
- **Generalisierung:** Die Bewertung der Policies ermöglicht es dem Agenten, Muster zu erlernen, den auf mehrere States anwenden kann. Damit wird das adaptieren des Agenten gestärkt.

Grundsätzlich wird der Trainingsverlauf verbessert oder schneller gemacht. Aber es ergeben sich einige Nachteile bzw. Einschränkungen bei dieser Methode.

- **Exploration vs Exploitation:** Die Bewertung von Policies kann dazu führen, dass der Agent nur mehr Exploitation betreibt, ohne neue Actions auszuprobieren, die vielleicht besser sein könnten.
- **Komplexe Umgebungen:** Die Bewertung von Policies erweist sich in komplexeren Umgebungen als recht schwer. Dies ist der Fall, wenn traditionelle Evaluierungsmethoden nicht sehr effektiv einsetzbar sind.
- **Ressourcenintensivität:** Die Bewertung der Policies kann viel Zeit und Rechenleistung aufbrauchen, vor allem bei Umgebungen mit vielen Actions und States.

[AK:Web18]

4.6 Methoden des Policygradients

4.6.1 Parametrisierung von Policy

Policy Gradient Methods gehören zu einer der wichtigsten Reinforcement Learning Methoden, die mithilfe von Parametrisierung Policies optimieren können. Dabei werden die Returns in einer sogenannten Fixed Policy Klasse gesucht. Im Gegensatz dazu suchen traditionelle Value Function Approximations bei einer Value Function. Mit ihnen ist es möglich, die optimale Policy leichter darzustellen, und zwar indem sie weniger Parameter benötigen. Ein Nachteil ist jedoch, dass sie in Off-Policy Umgebungen schwer zu implementieren sind.

[AK:Lit01]

4.6.2 REINFORCE

Eine beliebte Methode des Policy Gradient heißt REINFORCE. Dies ist ein Model-Free Approach, der in Umgebungen mit sowohl diskreten als auch kontinuierlichen Action Spaces angewandt werden kann. Da es Model-Free ist, benötigt es kein Wissen über seine Umgebung, um angewandt zu werden. In Policy-Based Methods wird die Policy ohne eine Value Function optimiert. REINFORCE optimiert die Policy, indem es die Gewichtungen des optimalen Policies mit Gradient Ascent approximiert.

[AK:Web16]

4.6.3 Advantage Actor-Critic (A2C)

Ein Nutzungsfall für Policy Gradients findet man in Actor Critic (Actor bezieht sich auf den Agenten und die Critic vergleicht die guten Werte mit den schlechteren Werten). Das Ziel von Actor Critic ist es, die Varianz zu reduzieren.

Actor Critic funktioniert, indem der Actor eine Action durchführt und der Critic diese Action dann bewertet und dem Actor einen Feedback ab liefert. Am Anfang macht der Actor nur zufällige Actions und erneuert die Policy mithilfe des Feedbacks vom Critic. Der Critic verbessert auch seinen Feedback, damit es besser beim nächsten Mal ist. Es werden 2 Neural Networks verwendet, einen für den Actor und einer für den Critic. Der Actor nutzt einen Policy Function und der Critic nutzt eine Value Function um die Actions zu bewerten.

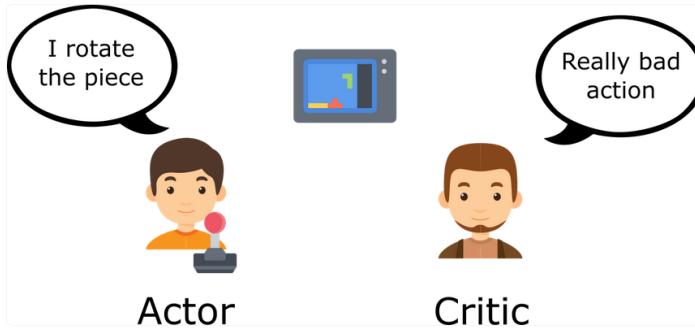


Abbildung 4.13: Actor Critic Beispiel
[AK:Web16]

Man kann den Lernprozess des Actors verbessern, indem eine Advantage Function angewandt wird, anstatt einer State Value Function. Advantage betrachtet, wie viel besser eine Action im gegebenen State $Q(s,a)$ ist, im Vergleich zum durchschnittlichen Wert $V(s)$ des States.

$$A(s, a) = Q(s, a) - V(s)$$

Diese Funktion berechnet daher den Reward, den der Actor bekommen würde, wenn er die bestimmte Action nehmen würde, verglichen mit dem durchschnittlichen Reward im State. Wenn $A(s, a) > 0$, dann wird unsere Policy in die Richtung verändert.

Das größte Problem von Advantage ist es, dass es zwei Value Functions benötigt, $Q(s, a)$ und $V(s)$. Um damit umzugehen, kann man einen TD Error als Schätzungs-funktion für Advantage nutzen.

[AK:Web16]

4.7 Deep Q-Netzwerke (DQNs)

Beim Q-Learning wird der Value einer Action in einer bestimmten State gelernt. Dieser braucht keinen Modell (daher Model-Free) und kann Probleme mit stochastischen Übergängen und Rewards ohne große Adaption behandeln. Für jede endliche Markov Decision Process (MDP) findet Q-Learning dafür die optimale Policy.

Dies kann unter den folgenden Bedingungen passieren, solange es eine unendliche Zeitangabe für die Exploration und eine teilweise zufällige Policy gibt. Der Q-Wert bezieht sich auf den erwarteten Wert des Action a im State s , welcher dann vom Algo-

rithmus berechnet wird.

[AK:Web14]

4.7.1 Approximation von Q-Functions

Ein wichtiger Nutzen der MDP findet es in Reinforcement Learning, zum Beispiel beim berechnen einer Q-Value. Das kann mithilfe dieser Funktion berechnet werden.

$$Q(s, a) = \sum_{s'} P_a(s, s')(R_a(s, s') + \gamma V(s'))$$

Die Q-Value wird mithilfe vom State s und Action a ermittelt.
 $s' \rightarrow$ was passiert ist, nachdem der Action a im State s ausgeführt wurde

4.7.2 Experience Replay

Bei Experience Replay werden die Experiences des Agenten in einer Tabelle (Replay Buffer) gespeichert. Ein Experience-Eintrag schaut daher dann so aus. Diese Daten können dann für State-Value Berechnungen angewandt:

$$e_t = (s_t, a_t, r_{t+1}, s_{t+1})$$

$s_t \rightarrow$ State zur Zeitpunkt t

$a_t \rightarrow$ Action zur Zeitpunkt t

$r_{t+1} \rightarrow$ Reward

$s_{t+1} \rightarrow$ Nächster Step

[AK:Web15]

4.7.3 Target-Networks

Deep Q Networks verwenden 2 Netzwerke, eine Q-Network und eine Target-Network. Der Target Network wird weniger häufig updated als das Q-Network. Der Sinn dahinter

ist es, die Divergenz während des Trainings zu reduzieren. Der Q-Network wird bei jeder Iteration updated und der Target-Network nur nach einem bestimmten Intervall.

[AK:Web15]

4.7.4 Umsetzung einer DQN

Um einen Beispiel für DQN herzuzeigen, wird ein einfacher Nachbau des Spiels Asteroids verwendet. Asteroids ist ein Spiel, welcher im Weltall spielt und zum ersten Mal im Jahr 1979 erschien ist. Im Spiel geht es darum, dass der Spieler eine Raumschiff steuert und versucht, die Asteroiden auszuweichen oder sie anzuschließen.

Der Nachbau wurde mit der Python Library Pygame erstellt. Der Spieler ist ein Dreieck und die Asteroiden werden als Kreise gezeigt. Jedes Asteroid bewegt sich in eine Richtung mit konstanter Geschwindigkeit. Der Spieler hat die Wahl, Asteroiden anzuschließen oder sie auszuweichen. Der Spieler kann sich drehen und sich auch bewegen. Der Grund für den Nachbau ist es, bestimmte Parameter zugänglich für den Agenten zu machen, ohne Einsatz von Bilderkennung oder ähnlichen.

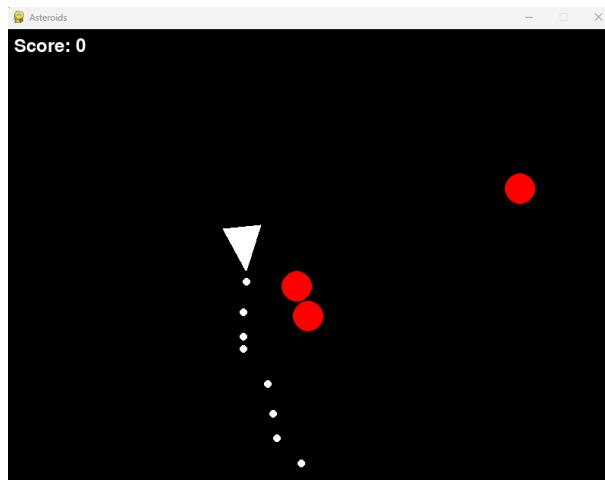


Abbildung 4.14: Screenshot vom Asteroids Spiel

Aufbau des Agenten

Der Agent verwendet einige Hyperparameter, um wichtige Einstellungen zu verändern, diese werden dann dem Agenten übergeben. Als Inputs bekommt der Agent dann die X und Y Koordinaten des Spielers, den momentanen Winkel des Spielers und auch die X und Y Koordinaten der nähersten Asteroiden und des nähersten Projektils:

```

1 # Anzahl der zu machenden Steps
2 num_iterations = 10000
3
4 # Einstellung der Collection Steps
5 initial_collect_steps = 15
6 collect_steps_per_iteration = 50
7 replay_buffer_capacity = 100000
8
9 # Layer des Neuralen Netzes
10 fc_layer_params = (100,)
11 batch_size = 64
12 learning_rate = 1e-3
13 gamma = 0.99
14 log_interval = 200
15 num_atoms = 51
16 min_q_value = -20
17 max_q_value = 20
18 n_step_update = 2
19
20 num_eval_episodes = 10
21 eval_interval = 1000
22 # Erstellung des Environments, der vom Agenten genutzt wird
23 env = AsteroidsEnvironment(AsteroidsGame())
24 env.reset()

```

Listing 4.1: Hyperparameter DQN

Der Categorical DQN Agent wird dabei erstellt und einige der Hyperparameter werden auch übergeben. Der Grund, einen Categorical DQN zu nutzen, ist es, die Verstreuung der Q-Values zu lernen, anstatt die Erwartungen zu approximieren.

```

1 agent = categorical_dqn_agent.CategoricalDqnAgent(
2     train_env.time_step_spec(),
3     train_env.action_spec(),
4     categorical_q_network=categorical_q_net,
5     optimizer=optimizer,
6     min_q_value=min_q_value,
7     max_q_value=max_q_value,
8     n_step_update=n_step_update,
9     td_errors_loss_fn=common.element_wise_squared_loss,
10    gamma=gamma,
11    train_step_counter=global_step)
12
13 agent.initialize()

```

Listing 4.2: DQNAgent Konstruktor

Der Average Return bezieht sich auf den durchschnittlichen Wert der Rewards, den der Agent bekommt. Diese Werte werden dann in eine Liste eingegeben, die dann zur

Zwerk eines Plots angewendet werden. Der Reward, den der Agent bekommt, hängt vom Punktestand und auch die Anzahl der Zeiteinheiten, wo der Agent nicht von einem Asteroiden getroffen wurde.

```

1 def compute_avg_return(environment, policy, num_episodes=10):
2     total_return = 0.0
3     for _ in range(num_episodes):
4         time_step = environment.reset()
5         episode_return = 0.0
6
7         while not time_step.is_last():
8             action_step = policy.action(time_step)
9             time_step = environment.step(action_step.action)
10            episode_return += time_step.reward
11            total_return += episode_return
12
13    avg_return = total_return / num_episodes
14    return avg_return.numpy()[0]

```

Listing 4.3: Average Return von DQN

So funktioniert der Training Loop, dabei wählt der Agent Actions, die dann in der Environment auch ausgeführt werden. Da werden mittels zwei If-Statements die betreffenden Checkpoints erstellt, um den Agenten zu persistieren. Die letzten 4 Checkpoints werden behalten. Das heißt, wenn der Agent während des Trainings gestoppt wird, wird trotzdem ein Checkpoint erstellt. Dieser wird dann am Anfang des nächsten Trainings gefunden und weiterverwendet. In jedem Checkpoint werden wichtige Variablen für den Agenten gespeichert, d.h. die derzeit beste Policy, bei welchem Step der Agent sich befand usw.

```

1  for _ in range(num_iterations):
2      for _ in range(collect_steps_per_iteration):
3          collect_step(train_env, agent.collect_policy)
4
5      # Daten vom Buffer nehmen und den Netzwerk des Agenten damit updaten
6      experience, unused_info = next(iterator)
7      train_loss = agent.train(experience).loss
8      step = agent.train_step_counter.numpy()
9
10     if step % log_interval == 0:
11         print('step = {0}: loss = {1}'.format(step, train_loss))
12         train_checkpointer.save(global_step)
13         print(f"Checkpoint saved to {checkpoint_dir}")
14
15     if step % eval_interval == 0:
16         avg_return = compute_avg_return(eval_env, agent.policy,
17                                         num_eval_episodes)
18         print('step = {0}: Average Return = {1:.2f}'.format(step,
19                                               avg_return))
20         returns.append(avg_return)

```

Listing 4.4: Training Loop DQN

Am Ende des Trainings wird ein Plot erstellt und angezeigt (Siehe 4.15). Auf der X-Achse findet man den Wert des Average Return (durchschnittlicher Wert aller Rewards) und auf der Y-Achse den jeweiligen Step.

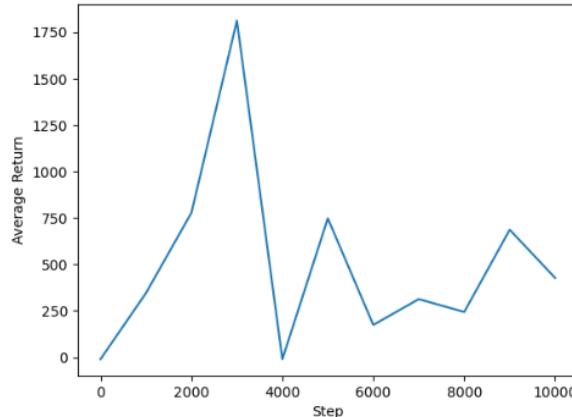


Abbildung 4.15: Training Ergebnis Plot

Aufbau der Umgebung

In der Umgebung werden die Steps, den der Agent bei jeder Iteration nimmt, definiert. Die Rewards und die Inputs, den der Agent bekommt, werden auch hier so definiert. Die Umgebungen werden so definiert, dass viele Arten von Agenten sie benutzen können. Im folgenden Codefragment sieht man wie die `_step` Methode definiert wurde. In ihr werden die benötigten Actions gemappt und die Inputs, den der Agent für den nächsten Step braucht, angegeben.

```

1 def _step(self, action):
2     # Actions gemappt auf die jeweiligen Aktionen, den ein Spieler machen
3     # kann
4     if action == 0:
5         self._asteroids_game.move_player_left()
6     elif action == 1:
7         self._asteroids_game.move_player_right()
8     elif action == 2:
9         self._asteroids_game.move_player_up()
10    elif action == 3:
11        self._asteroids_game.move_player_down()
12    elif action == 4:
13        if self._asteroids_game.bullet_timer >= 5:
14            self._asteroids_game.shoot_bullet()
15
16    # Ruft die Update Methode vom Spiel auf
17    self._asteroids_game.render()
18
19    # Inputs
20    observation = self.get_observation()
21    self.set_observation(observation)
22
23    # Derzeitigen Punktestand holen
24    self._score = self._asteroids_game.get_score()
25
26    # Belohnung aus Verhaeltnis zwischen der vergangenen Zeit und dem Score
27    # ab, sowie die vergangene Zeit
28    time_reward = self._asteroids_game.game_timer / 1000.0
29    score_time_ratio = self._score / (time_reward * 100)
30    self._reward = time_reward + score_time_ratio
31
32    # Belohnung wenn ein Asteroid getroffen wurde
33    if self._score > self._last_score:
34        self._reward = self._score + time_reward + score_time_ratio
35        self._last_score = self._score
36
37    if self._asteroids_game.get_collided():
38        self._episode_ended = True
39        print(f"Episode ended, last_reward: {self._reward}, score: {self._score}")
40        # Eine Bestrafung, wenn der Agent mit einem Asteroiden kollidiert
41        return ts.termination(observation=np.squeeze(observation), reward=-10)
42
43    return ts.transition(observation=np.squeeze(observation), reward=np.squeeze(self._reward), discount=1.0)

```

Listing 4.5: Action Space DQN

Der Agent bekommt folgende Inputs: derzeitige Position des Players, der Winkel des

Players (wohin die Spitze der Rakete zeigt = woraus dann geschossen wird). Dazu bekommt der Agent noch die Position des nähersten Asteroiden und auch den des nächsten Kugels, der geschossen wurde.

```

1 def get_observation(self):
2     player_x = self._asteroids_game.get_player_x()
3     player_y = self._asteroids_game.get_player_y()
4     player_angle = self._asteroids_game.get_player_angle()
5
6     closest_asteroid_dx, closest_asteroid_dy = self.
7         calculate_closest_asteroid()
8     closest_bullet_dx, closest_bullet_dy = self.calculate_closest_bullet()
9
10    observation = np.array([
11        player_x,
12        player_y,
13        player_angle,
14        closest_asteroid_dx,
15        closest_asteroid_dy,
16        closest_bullet_dx,
17        closest_bullet_dy
18    ], dtype=np.float32)
19
20    return observation

```

Listing 4.6: Observations DQN

4.8 Proximal Policy Optimization (PPO)

PPO wurde von John Schulman in 2017 entwickelt, mit dem Ziel, eine schwere Aufgabe zu lösen in dem es die Entscheidungsfunktion eines Agenten trainiert. Bekannt für seine einfache Implementierung und Tuning noch dazu mit der großen Performance, die er bietet, PPO wird auch seitdem als einer der Standard RL Algorithmus bei OpenAI verwendet.

[AK:Web19]

4.8.1 Policyoptimierung

PPO zählt als eines der Policy Gradient Methods, das bedeutet, dass der Policy eines Agenten trainiert wird. Der Policy wird mithilfe von Steps (Policy Updates) aktualisiert. Diese Steps dürfen weder zu klein (niedrigere Effizienz) noch zu groß (lenkt die Policy in eine falsche Richtung) sein.

[AK:Web19]

Clipping Mechanism

Um diese Updates zu verbessern, nutzt PPO eine Clip Function, welcher den Policy Update davon hindert, weder zu klein noch zu groß zu sein. Clipping reguliert dabei die Policy Updates und behaltet wiederverwendet Trainingsdaten, um die Effizienz zu behalten.

[AK:Web19]

$$L^{CLIP}(\theta) = \mathbb{E}_t[\min(r_t(\theta)A_t), \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon)A_t]$$

$$\text{Ratio Function } r_t(\theta) = \frac{\pi_\theta(a_t|b_t)}{\pi_{\theta(\text{old})}(a_t|b_t)}$$

Wenn $r_t(\theta) > 1$, dann ist die Action a_t wahrscheinlicher in der jetzigen Policy als im Alten. Ein $r_t(\theta)$ zwischen 0 und 1 deutet darauf, dass der Action a_t weniger wahrscheinlich in der jetzigen Policy ist.

Der Clip Function $\text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon)$ zeigt, dass der Ratio Function zwischen $1 - \epsilon$ und $1 + \epsilon$ geclipped ist.

Nach diesen Operationen erhält man die Surrogate Objective.

[AK:Web20]

4.8.2 Surrogate Objectives

Die Sample Efficiency (Probeneffizienz) von PPO ist eines seiner Stärken, sie gibt nämlich an, ob ein Algorithmus mehr oder weniger Daten braucht, um eine gute Policy zu trainieren. On-Policy Algorithmen wie PPO und TRPO (Trust Region Policy Optimization), weisen allgemein eine niedrige Efficiency auf. Trotzdem erreicht PPO eine hohe Efficiency aufgrund seiner Verwendung von Surrogate Objectives. Diese Objectives ermöglichen es dem Algorithmus, dass der neue Policy sich nicht zu weit von der alten abweicht. Die Sample Efficiency ist besonders nützlich für komplizierte Aufgaben, bei denen die Datensammlung und -berechnung kostspielig sein können.

Nachdem es berechnet wurde, hat der Agent zwei Objectives, eine clipped und eine non-clipped Objectives, von den beiden wird der minimum genommen, damit der Update eine möglichst sichere Update ist.

[AK:Web19]

4.8.3 Vergleich DQN Agent vs PPO Agent

Der PPO Agent ist ähnlich konstruiert, mithilfe der tf_agents Python Library. Der einzige großer Unterschied im Code der beiden liegt an den Hyperparametern den beide bekommen. Trotz dessen hat der PPO Agent eine maßgebend bessere Performance als die des DQN Agenten. Es lernt die Ziele des Spiels in weniger Zeit. Es lernt auch einige Schwachstellen im Codebasis und lernt, sie auch auszunutzen. Um sie zu vergleichen, hat der DQN Agent einen durchschnittlichen High-Score von 17, während der PPO Agent einen durchschnittlichen High-Score von 52 erreicht, auch nach sehr wenig Training.

Ein weiterer wesentlicher Unterschied zwischen den Agenten ist der Trainingszeit. Der DQN Agent schafft 10.000 Iterationen in einer Zeitspanne von einer Stunde, während der PPO Agent für 500 Iterationen über 4 Stunden braucht. Doch trotz der wesentlichen Unterschiede in Trainingszeit, zeigt der PPO Agent viel mehr Fortschritte und Kompetenz.

```

1 actor_net = actor_distribution_network.ActorDistributionNetwork(
2     train_env.observation_spec(),
3     train_env.action_spec(),
4     fc_layer_params=fc_layer_params)
5
6 value_net = value_network.ValueNetwork(
7     train_env.observation_spec(),
8     fc_layer_params=fc_layer_params)
9
10 optimizer = tf.compat.v1.train.AdamOptimizer(learning_rate=learning_rate)
11 train_step_counter = tf.Variable(0)
12 agent = ppo_agent.PPOAgent(
13     train_env.time_step_spec(),
14     train_env.action_spec(),
15     optimizer,
16     actor_net,
17     value_net,
18     num_epochs=10,
19     train_step_counter=train_step_counter,
20     discount_factor=gamma,
21     entropy_regularization=0.2,
22     importance_ratio_clipping=0.2,
23     use_gae=True,
24     use_td_lambda_return=True)

```

Listing 4.7: PPOAgent Konstruktor

4.9 Soft Actor Critic (SAC)

4.9.1 SAC-Einführung

Soft Actor-Critic wurde als eine Ergänzung zur Actor Critic eingeführt, um ein Paar der Schwächen der Algorithmen zu beseitigen. Ein Haupteinsatzgebiet für diesen Algorithm ist ein kontinuierlicher Action Space.

Das größte Merkmal von SAC ist, dass es eine modifizierte RL Objective-Function verwendet. Anstatt nur danach zu streben, die Lifetime-Rewards zu maximieren, versucht SAC auch, die Entropie der Policy zu maximieren.

[AK:Web17]

4.9.2 Implementierung der Netzwerke

Code Quelle: <https://github.com/vaishak2future/sac/blob/master/sac.ipynb>

Es werden 3 Netzwerke definiert, ein Value Network (V-Network), der Q-Network und der Policy-Network. Diese werden dann verwendet, um die Policy zu aktualisieren.

```

1  class ValueNetwork(nn.Module):
2      def __init__(self, state_dim, hidden_dim, init_w=3e-3):
3          super(ValueNetwork, self).__init__()
4
5          self.linear1 = nn.Linear(state_dim, hidden_dim)
6          self.linear2 = nn.Linear(hidden_dim, hidden_dim)
7          self.linear3 = nn.Linear(hidden_dim, 1)
8
9          self.linear3.weight.data.uniform_(-init_w, init_w)
10         self.linear3.bias.data.uniform_(-init_w, init_w)
11
12     def forward(self, state):
13
14
15 class SoftQNetwork(nn.Module):
16     def __init__(self, num_inputs, num_actions, hidden_size, init_w=3e-3):
17         super(SoftQNetwork, self).__init__()
18
19         self.linear1 = nn.Linear(num_inputs + num_actions, hidden_size)
20         self.linear2 = nn.Linear(hidden_size, hidden_size)
21         self.linear3 = nn.Linear(hidden_size, 1)
22
23         self.linear3.weight.data.uniform_(-init_w, init_w)
24         self.linear3.bias.data.uniform_(-init_w, init_w)
25
26     def forward(self, state, action):
27
28
29 class PolicyNetwork(nn.Module):
30     def __init__(self, num_inputs, num_actions, hidden_size, init_w=3e-3,
31                  log_std_min=-20, log_std_max=2):
32         super(PolicyNetwork, self).__init__()
33
34         self.log_std_min = log_std_min
35         self.log_std_max = log_std_max
36
37         self.linear1 = nn.Linear(num_inputs, hidden_size)
38         self.linear2 = nn.Linear(hidden_size, hidden_size)
39
40         self.mean_linear = nn.Linear(hidden_size, num_actions)
41         self.mean_linear.weight.data.uniform_(-init_w, init_w)
42         self.mean_linear.bias.data.uniform_(-init_w, init_w)

```

```
43     self.log_std_linear = nn.Linear(hidden_size, num_actions)
44     self.log_std_linear.weight.data.uniform_(-init_w, init_w)
45     self.log_std_linear.bias.data.uniform_(-init_w, init_w)
46
47     def forward(self, state):
48
49     def evaluate(self, state, epsilon=1e-6):
50
51
52     def get_action(self, state):
```

Listing 4.8: Network Aufbau SAC

4.9.3 Entropy-Regulation in SAC

Der Begriff "Entropie" bedeutet in diesem Fall die Unvorhersehbarkeit einer Zufallsvariable. Wenn eine Zufallsvariable immer einen einzelnen Wert annimmt, hat sie eine Entropie von Null, weil sie überhaupt nicht unvorhersehbar ist. Wenn eine Zufallsvariable jeden realen Wert mit gleicher Wahrscheinlichkeit annehmen kann, hat sie eine sehr hohe Entropie, da sie unvorhersehbar ist.

Warum sollte dann eine Policy eine hohe Entropie haben? Eine hohe Entropie (in unserer Policy) trägt dazu bei, explizit Exploration zu fördern, die Policy dazu zu ermutigen, gleiche Wahrscheinlichkeiten auf Actions zuzuweisen, die dieselben oder fast gleichen Q-Values haben. Dabei wird auch sichergestellt, dass sie nicht in die wiederholte Auswahl einer bestimmten Action zusammenbricht, die möglicherweise eine Inkonsistenz in der Q-Function ausnutzt. Daher wird überwunden, indem es das Policy Network dazu ermutigt, Exploration zu betreiben und keine hohe Wahrscheinlichkeit der bestimmten Teile des Action Spaces zuzuweisen.

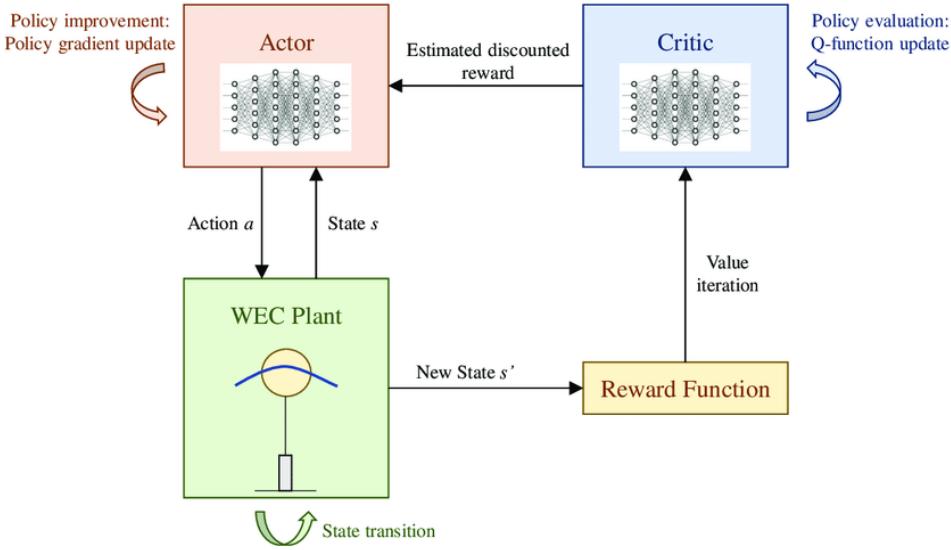


Abbildung 4.16: SAC Algorithmus
[AK:Img02]

[AK:Web17]

4.9.4 Q-Value-Optimierung in SAC

Bei SAC werden 2 Q-Networks angewandt und Ziel ist es, bei jedem State-Action pair die Q-Values zu approximieren. Während des Trainings werden die Netzwerke mit dem TD Error Function aktualisiert. Um die Policy zu aktualisieren, wird der Q-Network mit dem minimum gewählt. Die Target Q-Values, die in der Bellman Berechnung (Optimalitätsprinzip von Bellman) erscheinen, werden mithilfe der Target Policy und Target Value-Function berechnet. Diese Q-Values werden langsamer aktualisiert, um den Training stabiler zu halten. SAC aktualisiert neben den Q-Networks auch das Policy-Network, um die erwartete, kumulierte Reward zu maximieren.

[AK:Web17]

4.9.5 Zielfunktion von SAC

Bei SAC wird ein Value-Network verwendet, um die V-Functions zu aktualisieren. Unser V-Network ist von der Q-Network abhängig, welcher im Gegenzug von unserer V-Network abhängig ist, da eine Bellman Consistency erreicht werden soll. Das heißt,

unser V-Network hängt von den selben Parametern ab, den wir auch trainieren möchten. Dieser Vorgang macht den Training instabiler.

Die Lösung für dieses Dilemma ist es, Parameter zu verwenden, die den unserer Haupt V-Network ähneln, nur mit einer Zeitverzögerung.

[AK:Web17]

4.10 Anwendung des Model Free Reinforcement Learnings

4.10.1 Spielstrategie und Spielen

Beim Anwendung in Spielen brauchen MFRL Algorithmen relativ viele Inputs und Trainingsdaten, um einen Spiel zu erlernen und auch viel Trainingszeit und Optimierung, um sie auch noch zu meistern. Wie schon erwähnt, ist es möglich, einen Agenten zu trainieren, welcher Brettspiele (wie Schach und Go) und auch Videospiele (Atari, Asteroids...) spielen kann, in den meisten Fällen mit Unterstützung von Neural Networks. Mit genug Zeit und Mühe können diese Agente sogar besser werden, als die besten menschlichen Spieler. Bekannte Beispiele dafür sind AlphaGo, welcher von Google Deepmind entwickelt wurde.

4.10.2 Natürliche Sprachverarbeitung

Model Free Reinforcement Learning findet auch einen Nutzen beim Entwickeln von LLMs (Large Language Models). Bei Language Processing geht es darum, Texte zu generieren oder sie zu interpretieren. In Bezug auf Interpretation kann ein Model entwickelt werden, welcher sagen kann, ob ein Wort oder Phrase negativ, positiv oder neutral ist.

Beispiel:

- "Dieses Auto ist schwarz" → neutral
- "Die Sonne scheint" → positiv

PPO hilft in diesem Fall, den Model in kleineren, konsistenteren Schritten zu trainieren, damit es schrittweise besser wird. Als Beispiel kann man den LLM als Schüler betrachten,

der einen Essay schreibt. PPO hilft dabei, dass der Schüler besser wird, ohne seinen Schreibstil zu ändern.

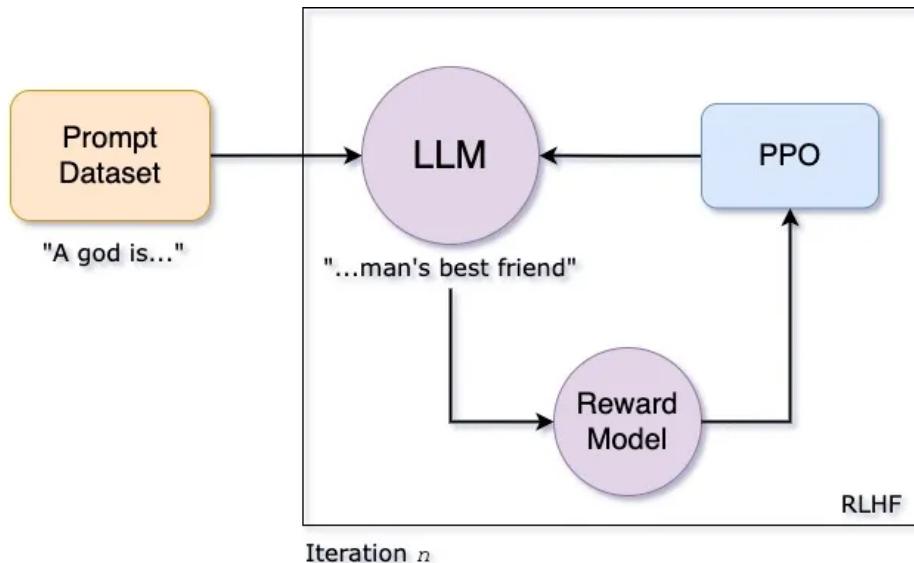


Abbildung 4.17: Einsatz von PPO in einer LLM
[AK:Web21]

[AK:Web21]

4.11 Projektbezug

Beim Projekt "Einsatz von LiDAR im autonomen Fahren" verwendet unser Team einen Model-Free Agenten, den Soft Actor Critic (SAC) Agent, welcher die Optimale Policy für den selbstfahrenden Auto findet. In der CARLA Simulation nutzen wir die Map "Town 5", da es eine relativ lange geradlinige Strecke anbietet, mit gut erkennbaren Hindernissen. Bei jedem Trainingstart laden wir die gewählte Map im Simulator mithilfe eines Python Programms, da CARLA über einen Python API verfügt.



Abbildung 4.18: CARLA Simulation

Als Input bekommt der Agent die LiDAR Daten (polar, d.h. Distanz und Winkel zum Punkt), die Geschwindigkeit- und Lenkungswerte, die derzeitige Location und die nächsten zwei Checkpoints (Nodes). Eine Kollision ist, wenn der Fahrzeug in der Simulation in einen anderen Objekt einfährt. Der Agent hat dann folgende Actions, die er wählen kann: Gas geben (0,1) und lenken (-1,1). Wie stark gelenkt wird, hängt vom Auto ab, welcher in der Simulation angewandt wird.

Am Anfang des Trainings macht der Agent einige Collection Steps, nach diesen Steps fängt der Agent mit seinen Versuchungen an. Während des eigentlichen Trainings werden Checkpoints vom aktuellen Policy gemacht, welcher beim Neustarten des Trainings wieder geladen wird. Zusätzlich werden der Step und der jeweilige Reward in einer CSV Datei geschrieben, welcher auch später geplottet werden kann.

Bei jedem Step macht der Agent seine Observations auf die LiDAR Daten und entscheidet, wie viel Gas er gibt und wie viel er lenkt. Er bekommt dann einen Reward bei jedem Step, dieser Reward wird erhöht, je weiter der Agent gefahren ist und je mehr Nodes erreicht wurde.

Im Falle einer Kollision oder beim Überschreiten eines arbiträren Step Limits bekommt der Agent einen Final Reward. Dieser hängt davon ab, wie weit der Agent gefahren ist. Den Step Limit nutzen wir, damit der Fahrzeug nicht unendlich fahren kann und auch den Trainingszeit wir damit verkürzen.

Am Ende des Trainings, werden alle Rewards und die jeweiligen Steps gesammelt und geplottet, dieser Graph soll zeigen, wie gut der Agent gelernt hat. Ein Erkenntnis, welcher bei einem konstanten Step Limit gefunden wurde, der Plot bei 2000 Iterationen 4.20, nach sehr wenigen Iterationen relativ flach geworden ist. Das deutet darauf, dass der Limiter nicht konstant sein sollte, oder dass er höher sein sollte.

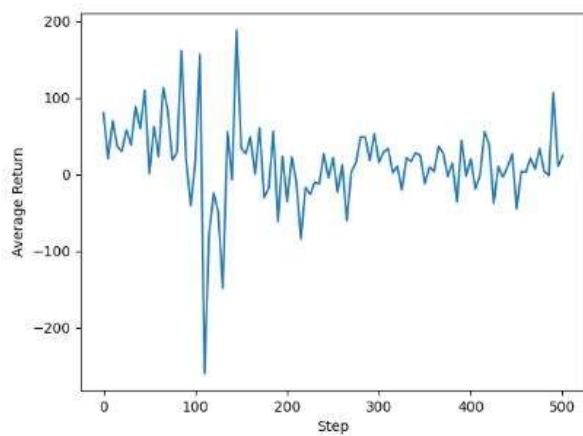


Abbildung 4.19: Plot nach 500 Iterationen

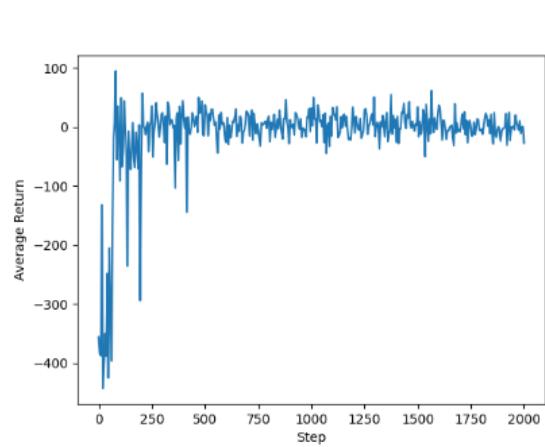


Abbildung 4.20: Plot nach 2000 Iterationen

Abbildungsverzeichnis

| | | |
|------|---|----|
| 1.1 | Funktionsweise von Sonar [PF:Web13] | 2 |
| 1.2 | Aufbau eines nodding mirror type LiDAR-Sensors [PF:Web23] . . . | 5 |
| 1.3 | Aufbau eines polygonal type LiDAR-Sensors [PF:Web34] | 6 |
| 1.4 | Funktionsweise Time-of-Flight [PF:Web16] | 7 |
| 1.5 | Erkennung der Phasenverschiebung [PF:Web32] | 8 |
| 1.6 | Aufbau LAS-Datei [PF:Web20] | 10 |
| 1.7 | Klassifizierung der erkannten Punkte [PF:Web12] | 12 |
| 1.8 | Beispiel für eine Punktfolge [PF:Web38] | 13 |
| 1.9 | Architektur von PointNet [PF:Web44] | 14 |
| 1.10 | Architektur von PointNet++ [PF:Web45] | 17 |
| 1.11 | Darstellung der Grouping-Methoden [PF:Web45] | 20 |
| 1.12 | Topografisches LiDAR [PF:Web08] | 21 |
| 1.13 | Bathymetrisches LiDAR [PF:Web08] | 21 |
| 1.14 | LiDAR-Sensoren unterteilt in Dimensionalität der Datenerfassung [PF:Web09] | 22 |
| 1.15 | Funktionsweise eines 1D-LiDAR-Sensors [PF:Web10] | 23 |
| 1.16 | Funktionsweise eines 2D-LiDAR-Sensors [PF:Web10] | 23 |
| 1.17 | Funktionsweise eines 3D-LiDAR-Sensors [PF:Web10] | 24 |

| | | |
|------|--|----|
| 1.18 | Unterteilung von Sensoren [PF:Web23] | 26 |
| 1.19 | Funktionsweise Radar [PF:Web24] | 31 |
| 1.20 | Unterschied der Wellenarten von LiDAR und Radar [PF:Web24] . . | 32 |
| 1.21 | Funktionsweise Ultraschall [PF:Web28] | 34 |
| 1.22 | Beispielhafte Sensorfusion eines Autos [PF:Web50] | 39 |
| 1.23 | WayPonDEV DTOF D300 [PF:Web41] | 42 |
| 1.24 | Montierter Sensor am Modellauto | 43 |
| 1.25 | Anzeige der erkannten Punkten | 46 |
| 2.1 | Beispiel für Pathfinding auf einem Graphen [EZ:Web05] | 50 |
| 2.2 | Ein gewichteter Graph [EZ:Web04] | 52 |
| 2.3 | Ein Digraph [EZ:Web20] | 52 |
| 2.4 | Ein Multigraph mit einer Schlinge [EZ:Web27, EZ:Web28] | 53 |
| 2.5 | Ein gerichteter Graph mit Multikanten [EZ:Web29] | 54 |
| 2.6 | Ein ungerichteter Graph mit Multikanten [EZ:Web29] | 54 |
| 2.7 | Ein einfacher Graph mit sechs Knoten und sieben Kanten [EZ:Web25] | 55 |
| 2.8 | Ein Netzwerk [EZ:Web10] | 55 |
| 2.9 | Ein zyklischer Graph mit einem Kreis [EZ:Web12] | 57 |
| 2.10 | Die Kreisgraphen C_1, C_2, C_3, C_4 und C_5 [EZ:Web14] | 57 |
| 2.11 | Die vollständigen Graphen K_1, K_2, K_3, K_4 und K_5 [EZ:Web33] . . | 58 |
| 2.12 | Ein einfacher, nicht vollständig bipartiter Graph mit Partitionsklassen U und V [EZ:Web19] | 59 |
| 2.13 | Ein einfacher, vollständig bipartiter Graph [EZ:Web19] | 59 |
| 2.14 | Beispiel für Pathfinding auf einem Grid [EZ:Web02] | 60 |
| 2.15 | C_3 bzw. K_3 , der kleinste Dreiecksgraph [EZ:Web31] | 61 |

| | | |
|------|---|-----|
| 2.16 | Beispiele für Min- und Max-Heap [EZ:Web60] | 96 |
| 2.17 | Struktur eines Fibonacci-Heaps [EZ:Web54] | 97 |
| 2.18 | Die Idee hinter der bidirektionalen Suche [EZ:Web62] | 103 |
| 2.19 | Graph mit zwei unterschiedlich langen Pfaden von A zu D | 108 |
| 2.20 | Die Korrelationen zwischen den Spalten | 110 |
| 2.21 | Vergleich der iterativen Algorithmen exklusive DFS | 112 |
| 2.22 | Vergleich der iterativen Algorithmen inklusive DFS | 114 |
| 2.23 | Der gelöste Zustand des 15-Puzzles | 115 |
| 2.24 | Ein ungelöster Zustand des 15-Puzzles | 116 |
| 2.25 | Ergebnisse von Bidi A* am 15-Puzzle | 120 |
| 2.26 | Die 18 möglichen Züge beim Rubik's Cube [EZ:Web61] | 121 |
| 2.27 | Das von LiCAR verwendete Modellauto | 125 |
| 3.1 | Agent interagiert mit Umgebung [MM:Web01] | 131 |
| 3.2 | Artificial Neuron [MM:Web08] | 132 |
| 3.3 | Layers [MM:Web07] | 133 |
| 3.4 | Activation Functions [MM:Web09] | 134 |
| 3.5 | Gradient Descent [MM:Web32] | 138 |
| 3.6 | Gradient Descent Minimum [MM:Web33] | 139 |
| 3.7 | Classification and Regression [MM:Web34] | 140 |
| 3.8 | Supervised Learning [MM:Web13] | 141 |
| 3.9 | Clustering [MM:Web39] | 142 |
| 3.10 | Unsupervised Learning [MM:Web13] | 143 |
| 3.11 | Reinforcement Learning [MM:Web14] | 144 |

| | | |
|------|---|-----|
| 3.12 | MBRL Flow Diagram [MM:Web20] | 148 |
| 3.13 | Planning and Learning [MM:Web30] | 149 |
| 3.14 | MBRL Environment Change 1 | 150 |
| 3.15 | MBRL Environment Change 2 | 151 |
| 3.16 | Monte Carlo Search Tree [MM:Web19] | 153 |
| 3.17 | Dynamics Model | 155 |
| 3.18 | likelihood - Reward | 156 |
| 3.19 | Correlation | 156 |
| 3.20 | Trajectory Model | 157 |
| 3.21 | Probabilistic Model | 158 |
| 3.22 | Flappy Bird Observations | 164 |
| 3.23 | Rewards over Steps | 182 |
| 3.24 | Loss over Steps | 183 |
| 4.1 | Vergleich: Model Free vs. Model Based [AK:Web03] | 185 |
| 4.2 | Reinforcement Learning [AK:Web02] | 187 |
| 4.3 | Value vs Policy [AK:Web01] | 188 |
| 4.4 | State Value Vergleich [AK:Web01] | 189 |
| 4.5 | Algorithmus illustriert [AK:Web01] | 190 |
| 4.6 | Vergleich MFRL Algorithmen [AK:Web01] | 192 |
| 4.7 | Tendenz zur Exploration bzw. Exploitation abhangig vom jetzigen Wissensstand | 195 |
| 4.8 | Vergleich Greedy, E-Greedy und Decaying-E | 196 |
| 4.9 | Epsilon Greedy [AK:Img01] | 197 |
| 4.10 | Decaying Epsilon Beispiel | 198 |

| | | |
|------|--|-----|
| 4.11 | Selbsterstellter Suchbaum | 199 |
| 4.12 | MCTS Methoden [AK:Web13] | 200 |
| 4.13 | Actor Critic Beispiel [AK:Web16] | 204 |
| 4.14 | Screenshot vom Asteroids Spiel | 206 |
| 4.15 | Training Ergebnis Plot | 209 |
| 4.16 | SAC Algorithmus [AK:Img02] | 218 |
| 4.17 | Einsatz von PPO in einer LLM [AK:Web21] | 220 |
| 4.18 | CARLA Simulation | 221 |
| 4.19 | Plot nach 500 Iterationen | 222 |
| 4.20 | Plot nach 2000 Iterationen | 222 |

Tabellenverzeichnis

| | | |
|-----|---|-----|
| 1.1 | Klassifizierungsflags [PF:Web12] | 11 |
| 1.2 | Klassifizierungswerte von LAS [PF:Web12] | 11 |
| 1.3 | Bewertungsskala | 41 |
| 1.4 | Bewertung der Lidar-Sensoren [PF:Web41, PF:Web42] | 41 |
| 2.1 | Wichtige Symbole der Mengenlehre [EZ:Web23, EZ:Web24] | 56 |
| 2.2 | Adjazenzliste | 61 |
| 2.3 | Die Platzkomplexitäten der Adjazenzliste und -matrix [EZ:Web40] | 62 |
| 2.4 | Adjazenzmatrix des Graphen aus <i>Abbildung 2.15</i> | 62 |
| 2.5 | Gewichtete Adjazenzliste des generierten Graphen aus <i>Listing 2.5</i> | 67 |
| 2.6 | Vergleich der Zeitkomplexitäten des Min- und Fibonacci-Heaps [EZ:Web55] | 97 |
| 2.7 | Die Vor- und Nachteile der verschiedenen Pathfinding-Algorithmen | 123 |

Listings

| | | |
|------|--|----|
| 1.1 | Konstanten/ constants.py | 43 |
| 1.2 | Methoden/ functions.py | 44 |
| 1.3 | Anzeige der empfangenen Daten/ lidar_visualizer.py | 44 |
| 2.1 | Graph.java | 64 |
| 2.2 | ModifiableGraph.java | 65 |
| 2.3 | Implementierung einer flexiblen Graph-Klasse in Java | 69 |
| 2.4 | ModifiableGraphRandomizer | 72 |
| 2.5 | Beispielanwendung der Klasse aus <i>Listing 2.4</i> | 74 |
| 2.6 | EndCondition.java | 76 |
| 2.7 | SearchAlgorithm.java | 77 |
| 2.8 | PathfindingAlgorithm.java | 78 |
| 2.9 | PathTracer.java | 80 |
| 2.10 | CycleException.java | 81 |
| 2.11 | BFS-Algorithmus in Java | 83 |
| 2.12 | Iterativer DFS-Algorithmus in Java | 86 |
| 2.13 | Rekursiver DFS-Algorithmus in Java | 89 |
| 2.14 | BestFirstSearch.java | 92 |
| 2.15 | AbstractBestFirstSearch.java | 94 |

| | |
|---|-----|
| 2.16 Priority-Queue mit Comparator für Best-first search | 97 |
| 2.17 Aktualisierung der Prioritäten ohne Knoten mehrfach einzufügen | 98 |
| 2.18 decreaseKey-Methode aus FibonacciHeap.java | 98 |
| 2.19 checkPriority-Methode aus FibonacciHeap.java | 98 |
| 2.20 Dijkstra-Algorithmus in Java | 100 |
| 2.21 A*-Algorithmus in Java | 101 |
| 2.22 Heuristic.java | 101 |
| 2.23 BidiBestFirstSearch.java | 105 |
| 2.24 Pathfinder.java | 107 |
| 2.25 Typinferenz | 108 |
| 2.26 Repräsentation des 15-Puzzles als Graph | 117 |
| 2.27 MemoizedGraph.java | 118 |
| 2.28 Berechnung der minimalen Anzahl an benötigten Zügen | 119 |
| 2.29 FifteenPuzzleHeuristic.java | 119 |
| 2.30 Erzeugung eines Graphen aus einer CARLA-Map | 125 |
| 2.31 Berechnung des kürzesten Pfades mittels NetworkX | 126 |
| 4.1 Hyperparameter DQN | 207 |
| 4.2 DQNAgent Konstruktor | 207 |
| 4.3 Average Return von DQN | 208 |
| 4.4 Training Loop DQN | 209 |
| 4.5 Action Space DQN | 211 |
| 4.6 Observations DQN | 212 |
| 4.7 PPOAgent Konstruktor | 215 |

| | |
|----------------------------------|-----|
| 4.8 Network Aufbau SAC | 216 |
|----------------------------------|-----|

Literaturverzeichnis

[PF:Web01] <https://flyguys.com/the-evolution-of-lidar/>
The Evolution of LiDAR
15.11.2023

[PF:Web02] <https://www.artec3d.com/de/learning-center/what-is-lidar>
Wie funktioniert LiDAR?
22.11.2023

[PF:Web03] https://en.wikipedia.org/wiki/LAS_file_format
LAS-Dateiformat
27.11.2023

[PF:Web04] <https://www.artec3d.com/de/learning-center/what-is-lidar>
Zusammenfassung
22.11.2023

[PF:Web05] <https://www.netzwelt.de/abkuerzung/179046-bedeutet-lidar-erklaerung-definition.html>
Was bedeutet LiDAR
22.11.2023

[PF:Web06] <https://www.researchgate.net/figure/The-schematic-diagram-of-a-time-of-flight-TOF-sensor-fig4-3353175601>
Figure 4 - available via license: CC BY
22.11.2023

[PF:Web07] <https://www.all-electronics.de/elektronik-entwicklung/die-top-5-der-lidar-anwendungen-ausserhalb-des-autos.html>
Die Top 5 der Lidar-Anwendungen außerhalb des Autos
18.11.2023

[PF:Web08] <https://desktop.arcgis.com/de/arcmap/latest/manage-data/las-dataset/types-of-lidar.htm>
LIDAR-Typen
23.11.2023

[PF:Web09] <https://www.cadden.fr/en/lidar-technology/>

Was ist der Unterschied zwischen 2D-Lidar und 3D-Lidar?

24.11.2023

[PF:Web10] <https://www.generationrobots.com/blog/de/was-ist-die-lidar-technologie/>

Die verschiedenen Sichtsysteme des LiDAR

25.11.2023

[PF:Web11] https://cdn.sick.com/media/docs/5/25/425/whitepaper_lidar_de_im0079425.pdf

Linear messende Sensoren (1D)

25.11.2023

[PF:Web12] <https://desktop.arcgis.com/de/arcmap/latest/manage-data/las-dataset/lidar-point-classification.htm>

Klassifizierung von LIDAR-Punkten

25.11.2023

[PF:Web13] <https://de.wikipedia.org/wiki/Sonar>

Aktives Sonar

30.11.2023

[PF:Web14] <https://geo-plus.com/de/kurzgeschichte-uber-die-lidar-technologie/>

Das Wachstum

30.11.2023

[PF:Web15] <https://seminex.com/de/lidar/>

LIDAR

30.11.2023

[PF:Web16] <https://www.all-electronics.de/automotive-transportation/lidar-sensoren-automotive-575.html>

Was ist Lidar und für was steht es?

30.11.2023

[PF:Web17] <https://www.all-electronics.de/automotive-transportation/lidar-sensoren-automotive-575.html#Vor-%20und%20Nachteile>

Was sind die Vor- und Nachteile von Lidar?

25.12.2023

[PF:Web18] <https://desktop.arcgis.com/de/arcmap/latest/manage-data/las-dataset/storing-lidar-data.htm>

Speichern von LIDAR-Daten

25.12.2023

[PF:Web19] <https://www.trend.at/tech/lidar>

Die Nachteile von Laserstrahlen für die Abstandsmessung

28.12.2023

[PF:Web20] <https://miltomiltiadou.blogspot.com/2017/08/the-structure-of-las13-file-format-used.html>

The structure of the LAS1.3 file format used to store full-waveform LiDAR data

28.12.2023

[PF:Web21] <https://docs.fileformat.com/de/gis/las/>

Was ist eine LAS-Datei?

28.12.2023

[PF:Web22] https://en.wikipedia.org/wiki/LAS_file_format

LAS file format

28.12.2023

[PF:Web23] <https://www.cbcity.de/fahrzeugumfeldsensorik-ueberblick-und-vergleich-zwischen-lidar-radar-video>

Arten der Umfeldsensorik

02.01.2024

[PF:Web24] <https://flyguys.com/lidar-vs-radar/>

LiDAR vs. RADAR: Was ist der Unterschied?

03.01.2024

[PF:Web25] <https://de.wikipedia.org/wiki/Radar>

Radar

03.01.2024

[PF:Web26] <https://www.radartutorial.eu/01.basics/Entfernungsmessung%20mit%20Radar.de.html>

Entfernungsmessung mit Radar

04.01.2024

[PF:Web27] <https://de.wikipedia.org/wiki/Ultraschall>

Ultraschall

04.01.2024

[PF:Web28] <https://www.microsonic.de/de/service/ultraschallsensoren/prinzip.htm>

Ultraschallprinzip

04.01.2024

[PF:Web29] <https://maxbotix.com/blogs/blog/ultrasonic-sensors-vs-lidar-which-one-should-you-use>

Ultrasonic Sensors vs. LiDAR: Which One Should You Use?

04.01.2024

[PF:Web30] https://www.keyence.de/ss/products/measure/measurement_library/type/ultrasonic/#::text=Wenn%20der%20gemessene%20Abstand%20L,T%20x%20

C%20berechnet%20werden.
Ultraschall-Wegmesssensoren
04.01.2024

[PF:Web31] <https://seminex.com/lidar/#:text=By%20comparing%20the%20phase%20difference,which%20the%20power%20was%20modulated.>
PHASE SHIFT
26.01.2024

[PF:Web32] <https://www.terabee.com/a-brief-introduction-to-time-of-flight-sensing-part-2-indirect-tof-sensors/>
A brief introduction to Time-of-Flight sensing. Part 2 Indirect ToF Sensors
26.01.2024

[PF:Web33] <https://scholar.uwindsor.ca/cgi/viewcontent.cgi?article=9300&context=etd>
3D LiDAR Point Cloud Processing Algorithms
27.01.2024

[PF:Web34] <https://en.study-robot.jp/sensor/lidar/>
Polygon mirror type
27.01.2024

[PF:Web35] <https://radar-blog.innosent.de/die-10-aussergewoehnlichsten-radaranwendungen/>
Die 10 außergewöhnlichsten Radaranwendungen
30.01.2024

[PF:Web36] <https://radar-blog.innosent.de/radartechnik-im-strassenverkehr/>
8. Abstrafende Verkehrstechnik
30.01.2024

[PF:Web37] <https://www.oxts.com/de/what-is-a-pointcloud/>
Was ist eine Punktewolke?
30.01.2024

[PF:Web38] <https://www.blickfeld.com/de/blog/smart-lidar-bewegungserkennung/>
On-Device Bewegungserkennung minimiert LiDAR-Datenübertragung
30.01.2024

[PF:Web39] <https://worldofvr.de/was-ist-eine-point-cloud/#:~:text=Die%20Punktwolken%20stellen%20die%20Grundlage,vorgegebene%20Position%20im%20dreidimensionalen%20Raum.>
Was sind Point Cloud Daten?
31.01.2024

- [PF:Web40] https://www.amazon.de/-/en/WayPonDEV-A2M12-Distance-Intelligent-Avoidance/dp/B0B76NT595/ref=sr_1_5?keywords=WayPonDEV&qid=1694439957&sr=8-5
WayPonDEV Slamtec RPLIDAR A2M12 360 Degree 2D Lidar Sensor Kit, 15Hz Scan Rate and 12 Meter Distance Module for Intelligent Obstacle Avoidance/Robot/Maker Training
03.02.2024
- [PF:Web41] https://www.amazon.de/-/en/WayPonDEV-DTOF-D300-Distance-Intelligent/dp/B0B1VD2PJH/ref=sr_1_8?keywords=WayPonDEV&qid=1694439957&sr=8-8
WayPonDEV DTOF D300 360 Degree 2D Laser Distance Sensor Kit, 10Hz Scan Rate and 12M Distance Lidar Scanner Module for Intelligent Obstacle Robot Maker Training Indoor Outdoor
03.02.2024
- [PF:Web42] https://www.amazon.de/Scanradius-LIDAR-Sensorscanner-Vermeidung-Hindernissen-Navigation/dp/B07VLFGT27/ref=asc_df_B07VLFGT27/?tag=&linkCode=df0&hvadid=407389247407&hvpos=&hvnetw=g&hvrand=1457886404614269112&hvpone=&hvptwo=&hvqmt=&hvdev=c&hvdvcmdl=&hvlocint=&hvlccphy=1000772&hvtargid=pla-843524520219&th=1&ref=&adgrpid=88734954362
Slamtec RPLIDAR A1 2D 360 Degree 12 Meter Scanning Radius LIDAR Sensor Scanner for Avoiding Obstacles and Navigating Robots
03.02.2024
- [PF:Web43] <https://de.wikipedia.org/wiki/Ultraschall#:text=Ultraschall%20wird%20in%20der%20industriellen,Fl%C3%BCssigkeiten%20oder%20der%20zerst%C3%BCrungsfreien%20Materialpr%C3%BCfung.>
Ultraschall
04.02.2024
- [PF:Web44] <https://arxiv.org/pdf/1612.00593v2.pdf>
PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation
07.02.2024
- [PF:Web45] https://repositorium.hs-ruhrwest.de/frontdoor/deliver/index/docId/620/file/Bachelorarbeit_Niclas_Huewe_10007508.pdf
PointNet++
10.02.2024
- [PF:Web46] <https://stanford.edu/rqi/pointnet2/>
PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space
07.02.2024

- [PF:Web47] <https://www.bosch-mobility.com/de/loesungen/sensoren/sensordatenfusion/>
Sensordatenfusion
10.02.2024
- [PF:Web48] <https://medium.com/dorlecontrols/sensor-fusion-242bc70e7332>
Sensorfusion
10.02.2024
- [PF:Web49] <https://de.wikipedia.org/wiki/Sensordatenfusion>
Sensordatenfusion
10.02.2024
- [PF:Web50] <https://www.linkedin.com/pulse/sensor-fusion-advancing-adas-functions-safer-more-efficient-yousif>
Sensor Fusion: Advancing ADAS Functions for Safer and More Efficient Driving
10.02.2024
- [PF:Web51] <https://www.sensortips.com/featured/how-many-types-of-radar-are-there-faq/#:~:text=With%20a%20range%20of%20up,medium%20to%20low%20resolution.>
Sensor Fusion: Advancing ADAS Functions for Safer and More Efficient Driving
10.02.2024
- [PF:Web52] <https://datasolut.com/neuronale-netzwerke-einfuehrung/>
Künstliche Neuronale Netzwerke: Definition, Einführung, Arten und Funktion
11.02.2024
- [PF:Web53] <https://datascientest.com/de/perceptron#:~:text=Was%20ist%20ein%20Perceptron%3F,eine%20Einheit%20eines%20neuronalen%20Netzes.>
Künstliche Neuronale Netzwerke: Definition, Einführung, Arten und Funktion
11.02.2024
- [PF:Web54] https://de.wikipedia.org/wiki/Orthogonale_Matrix
Orthogonale Matrix
11.02.2024
- [PF:Web55] <https://welt-der-bwl.de/Nutzwertanalyse>
Nutzwertanalyse
11.02.2024
- [PF:Web56] <https://de.wikipedia.org/wiki/Pygame>
Pygame
11.02.2024
- [PF:Web57] <https://de.wikipedia.org/wiki/TOF-Kamera>
TOF-Kamera
02.03.2024

[PF:Web58] <https://erc-bpgc.github.io/handbook/electronics/Sensors/lidar/>
Komponenten eines grundlegenden Lidar-Sensors
30.12.2023

[EZ:Web01] <https://www.ionos.at/digitalguide/online-marketing/web-analyse/pathfinding>
Pathfinding: Wegfindung in der Informatik
03.11.2023

[EZ:Web02] <https://www.geeksforgeeks.org/a-search-algorithm/>
A*-Algorithm
28.10.2023

[EZ:Web03] <https://arxiv.org/pdf/physics/0510162.pdf>
Structural Properties of Planar Graphs of Urban Street Patterns
03.11.2023

[EZ:Web04] <https://www.baeldung.com/cs/weighted-vs-unweighted-graphs>
Weighted vs. Unweighted Graphs
28.10.2023

[EZ:Web05] <https://happyCoding.io/tutorials/libgdx/pathfinding>
Pathfinding
04.11.2023

[EZ:Web06] <https://studyflix.de/informatik/grundbegriffe-der-graphentheorie-1285>
Grundbegriffe der Graphentheorie
13.11.2023

[EZ:Web07] <https://blog.viking-studios.net/wp-content/uploads/2013/04/Pathfinding-Algorithmen-in-verschiedenen-Spielegenres.pdf>
Pathfinding-Algorithmen in verschiedenen Spielegenres
14.11.2023

[EZ:Web08] <https://yuminlee2.medium.com/a-search-algorithm-42c1a13fcf9f>
A* Search Algorithm
14.11.2023

[EZ:Web09] http://gitta.info/Accessibilit/de/html/NetworkChara_learningObject1.html
Charakterisierung von Netzwerken
20.11.2023

[EZ:Web10] <https://hyperskill.org/learn/step/5645>
Weighted Graph
14.11.2023

[EZ:Web11] <https://mathworld.wolfram.com/GraphCycle.html>

Graph Cycle

20.11.2023

[EZ:Web12] <https://www.geeksforgeeks.org/what-is-cyclic-graph/>

What is Cyclic Graph

15.11.2023

[EZ:Web13] <https://files.ifi.uzh.ch/cl/siclemat/lehre/hs07/ecl1/script/html/scriptse50.html>

Graphen

16.11.2023

[EZ:Web14] <https://mathworld.wolfram.com/CycleGraph.html>

Cycle Graph

20.11.2023

[EZ:Web15] <https://www.geeksforgeeks.org/degree-of-a-cycle-graph/>

Degree of a Cycle Graph

16.11.2023

[EZ:Web16] [https://en.wikipedia.org/wiki/Cycle_\(graph_theory\)](https://en.wikipedia.org/wiki/Cycle_(graph_theory))

Cycle (graph theory)

16.11.2023

[EZ:Web17] [https://de.wikipedia.org/wiki/Zyklus_\(Graphentheorie\)](https://de.wikipedia.org/wiki/Zyklus_(Graphentheorie))

Zyklus (Graphentheorie)

16.11.2023

[EZ:Web18] https://en.wikipedia.org/wiki/A*_search_algorithm

A* search algorithm

16.11.2023

[EZ:Web19] https://de.wikipedia.org/wiki/Bipartiter_Graph

Bipartiter Graph

20.11.2023

[EZ:Web20] https://de.wikipedia.org/wiki/Gerichteter_Graph

Gerichteter Graph

20.11.2023

[EZ:Web21] <https://www.geeksforgeeks.org/implementing-generic-graph-in-java>

Implementing Generic Graph in Java

20.11.2023

[EZ:Web22] https://de.wikipedia.org/wiki/Kantengewichteter_Graph

Kantengewichteter Graph

24.11.2023

[EZ:Web23] <https://www.mathe-online.at/symbole.html>

Mathematische Symbole und Abkürzungen

24.11.2023

[EZ:Web24] <https://www.mathsisfun.com/sets/symbols.html>

Set Symbols

24.11.2023

[EZ:Web25] [https://en.wikipedia.org/wiki/Graph_\(discrete_mathematics\)](https://en.wikipedia.org/wiki/Graph_(discrete_mathematics))

Graph (discrete mathematics)

24.11.2023

[EZ:Web26] https://de.wikipedia.org/wiki/Einfacher_Graph

Einfacher Graph

24.11.2023

[EZ:Web27] [https://de.wikipedia.org/wiki/Schleife_\(Graphentheorie\)](https://de.wikipedia.org/wiki/Schleife_(Graphentheorie))

Schleife (Graphentheorie)

24.11.2023

[EZ:Web28] [https://en.wikipedia.org/wiki/Loop_\(graph_theory\)](https://en.wikipedia.org/wiki/Loop_(graph_theory))

Loop (graph theory)

24.11.2023

[EZ:Web29] [https://de.wikipedia.org/wiki/Graph_\(Graphentheorie\)](https://de.wikipedia.org/wiki/Graph_(Graphentheorie))

Graph (Graphentheorie)

24.11.2023

[EZ:Web30] <https://de.wikipedia.org/wiki/Idempotenz>

Idempotenz

27.11.2023

[EZ:Web31] https://en.wikipedia.org/wiki/Adjacency_list

Adjacency list

27.11.2023

[EZ:Web32] <https://www.scaler.com/topics/data-structures/graph-in-data-structure/>

Graph in Data Structure

27.11.2023

[EZ:Web33] https://de.wikipedia.org/wiki/Vollständiger_Graph

Vollständiger Graph

27.11.2023

[EZ:Web34] https://en.wikipedia.org/wiki/Complete_graph

Complete graph

27.11.2023

[EZ:Web35] https://de.wikipedia.org/wiki/Knotengewichteter_Graph
Knotengewichteter Graph
28.11.2023

[EZ:Web36] <https://de.wikipedia.org/wiki/Adjazenzmatrix>
Adjazenzmatrix
28.11.2023

[EZ:Web37] https://en.wikipedia.org/wiki/Adjacency_matrix
Adjacency matrix
28.11.2023

[EZ:Web38] https://de.wikipedia.org/wiki/Reductio_ad_absurdum
Reductio ad absurdum
28.11.2023

[EZ:Web39] [https://de.wikipedia.org/wiki/Kante_\(Graphentheorie\)](https://de.wikipedia.org/wiki/Kante_(Graphentheorie))
Kante (Graphentheorie)
29.11.2023

[EZ:Web40] <https://www.educative.io/answers/what-is-an-adjacency-list>
What is an adjacency list?
29.11.2023

[EZ:Web41] <https://cseweb.ucsd.edu/~kube/cls/100/Lectures/lec11/lec11-8.html>
Adjacency matrices
29.11.2023

[EZ:Web42] <https://iq.opengenus.org/adjacency-matrix/>
Adjacency Matrices Explained: A Representation of Graphs
29.11.2023

[EZ:Web43] <https://people.engr.tamu.edu/djimenez/ut/utsa/cs1723/lecture16.html>
Weighted Graphs
29.11.2023

[EZ:Web44] <https://refactoring.guru/design-patterns/builder>
Builder
30.11.2023

[EZ:Web45] <https://networkx.org/>
NetworkX
1.12.2023

[EZ:Web46] <https://jgrapht.org/>
JGraphT
1.12.2023

[EZ:Web47] https://en.wikipedia.org/wiki/Dijkstra's_algorithm

Dijkstra's algorithm

29.01.2024

[EZ:Web48] https://de.wikipedia.org/wiki/Open_Shortest_Path_First

Open Shortest Path First

06.02.2024

[EZ:Web49] https://en.wikipedia.org/wiki/Open_Shortest_Path_First

Open Shortest Path First

06.02.2024

[EZ:Web50] <https://stackoverflow.com/questions/19894509>

What is the purpose of the visited set in Dijkstra?

15.02.2024

[EZ:Web51] <https://de.wikipedia.org/wiki/Greedy-Algorithmus>

Greedy-Algorithmus

15.02.2024

[EZ:Web52] https://en.wikipedia.org/wiki/Greedy_algorithm

Greedy algorithm

15.02.2024

[EZ:Web53] <https://keithschwarz.com/interesting/code/?dir=fibonacci-heap>

FibonacciHeap.java

20.02.2024

[EZ:Web54] <https://www.geeksforgeeks.org/fibonacci-heap-set-1-introduction/>

Fibonacci Heap | Set 1 (Introduction)

23.02.2024

[EZ:Web55] <https://de.wikipedia.org/wiki/Fibonacci-Heap>

Fibonacci-Heap

23.02.2024

[EZ:Web56] <https://stackoverflow.com/questions/6065710>

How does Java's PriorityQueue differ from a min-heap?

25.02.2024

[EZ:Web57] https://de.wikipedia.org/wiki/A*-Algorithmus

A*-Algorithmus

27.02.2024

[EZ:Web58] https://en.wikipedia.org/wiki/Bidirectional_search

Bidirectional search

03.03.2024

[EZ:Web59] <https://de.wikipedia.org/wiki/Memoisation>

Memoisation

09.03.2024

[EZ:Web60] <https://www.geeksforgeeks.org/heap-data-structure/>

Heap Data Structure

09.03.2024

[EZ:Web61] <https://medium.com/nerd-for-tech/3cb9b53f94b5>

Rubik's Cube algorithms for machines

15.03.2024

[EZ:Web62] <https://www.baeldung.com/cs/bidirectional-search>

Bidirectional Search for Path Finding

15.03.2024

[EZ:Web63] <https://www.youtube.com/watch?v=A60q6dcoCjw>

The hidden beauty of the A* algorithm

15.03.2024

[EZ:Web64] <https://www.youtube.com/watch?v=wL3uWO-KLUE>

The trick that solves Rubik's Cubes and breaks ciphers

16.03.2024

[EZ:Web65] <https://refactoring.guru/design-patterns/strategy>

Strategy

21.03.2024

[MM:Book01] Peter Norvig, Stuart J. Russell: *Artificial Intelligence: A Modern Approach* Prentice Hall

28.4.2020

ISBN: 0-13-461099-7

Seite 2-3

[MM:Book02] Paul Wilmott *Machine Learning, An Applied Mathematics Introduction*

Panda Ohana Publishing

2019

ISBN: 9781916081604

[MM:Book03] Peter Norvig, Stuart J. Russell: *Artificial Intelligence: A Modern Approach* Prentice Hall

28.4.2020

ISBN: 0-13-461099-7

Seite 2-5

[MM:Web01] https://www.tutorialspoint.com/artificial_intelligence/artificial_intelligence_agents_and_environments.htm
AI - Agents & Environments
06.11.2023

[MM:Web02] https://en.wikipedia.org/wiki/Unsupervised_learning
Unsupervised Learning
30.11.2023

[MM:Web03] https://en.wikipedia.org/wiki/Supervised_learning
Supervised Learning
30.11.2023

[MM:Web04] <https://www.ibm.com/topics/machine-learning>
Machine Learning
30.11.2023

[MM:Web05] https://en.wikipedia.org/wiki/Deep_learning
Deep Learning
30.11.2023

[MM:Web06] <https://saturncloud.io/blog/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way/>
Convolutional Neural Network
21.12.2023

[MM:Web07] <https://medium.com/ravenprotocol/everything-you-need-to-know-about-neural-networks-6fcc7a15cb4>
Layers
21.12.2023

[MM:Web08] https://www.gabormelli.com/RKB/Artificial_Neuron_Layers
21.12.2023

[MM:Web09] <https://medium.com/@shrutijadon/survey-on-activation-functions-for-deep-learning-9689331ba092>
Activation Function
29.12.2023

[MM:Web10] <https://www.ibm.com/topics/neural-networks#:~:text=Neural%20networks%2C%20also%20known%20as,neurons%20signal%20to%20one%20another.>
Neural Networks
29.12.2023

[MM:Web11] <https://cims.nyu.edu/donev/Teaching/WrittenOral/Projects/XintianHan-WrittenAndOral.pdf>

MDP

29.12.2023

[MM:Web12] https://books.google.at/books?hl=en&lr=&id=ZEID3w9STOUC&oi=fnd&pg=PP7&dq=neural+networks&ots=sAHwEYWHZN&sig=f_4BZScECmeRDM9wfxH2OY9Ax-Y&redir_esc=y#v=onepage&q=neural%20networks&f=false
Neural Networks

29.12.2023

[MM:Web13] <https://medium.com/@metehankozan/supervised-and-unsupervised-learning-an-intuitive-approach-cd8f8f64b644>
Supervised-unsupervised Learning
29.12.2023

[MM:Web14] <https://www.mathworks.com/help/reinforcement-learning/ug/what-is-reinforcement-learning.html>
Reinforcement Learning
29.12.2023

[MM:Web15] <https://cims.nyu.edu/donev/Teaching/WrittenOral/Projects/XintianHan-WrittenAndOral.pdf>
Reinforcement Learning
29.12.2023

[MM:Web16] <https://arxiv.org/abs/2206.09328>
Fan-Ming Luo, Tian Xu, Hang Lai, Xiong-Hui Chen, Weinan Zhang, Yang Yu
A survey on model-based reinforcement learning.
05.01.2024

[MM:Web17] <https://link.springer.com/article/10.1007/s10462-022-10335-w>
Aske Plaat, Walter Kosters & Mike Preuss
High-accuracy model-based reinforcement learning, a survey. Artificial Intelligence
05.01.2024

[MM:Web18] <https://ieeexplore.ieee.org/document/10007800>
Thomas M. Moerland; Joost Broekens; Aske Plaat; Catholijn M. Jonker
Model-based reinforcement learning: A survey.
05.01.2024

[MM:Web19] <https://www.geeksforgeeks.org/ml-monte-carlo-tree-search-mcts/>
Rahul Roy
Monte Carlo Search Tree
05.01.2024

[MM:Web20] <https://medium.com/the-official-integrate-ai-blog/understanding-reinforcement-learning-93d4e34e5698>
MBRL Flow-Diagram
05.01.2024

[MM:Web21] <https://medium.com/@shrutijadon/survey-on-activation-functions-for-deep-learning-9689331ba092>
Activation Functions
05.01.2024

[MM:Web22] https://en.wikipedia.org/wiki/Model_predictive_control
MPC
05.01.2024

[MM:Web23] [https://www.mdpi.com/2075-1680/11/2/80#: text=Hence%2C%20each%20artificial%20neuron%20may,b%20\)%20%20%20see%20Figure%201.](https://www.mdpi.com/2075-1680/11/2/80#:text=Hence%2C%20each%20artificial%20neuron%20may,b%20)%20%20%20see%20Figure%201.)
Neural Networks / Neurons
05.03.2024

[MM:Web24] https://escholarship.org/content/qt9sj5258g/qt9sj5258g_noSplash_580e643a2ee0bd41f9c0fb4bb91cc930.pdf
Control in MBRL | Ph.D Nathan Lambert
15.03.2024

[MM:Web25] <https://www.youtube.com/watch?v=H5Q1UAEkhZY>
MPC in MBRL | Ph.D Nathan Lambert
15.03.2024

[MM:Web26] <https://arxiv.labs.arxiv.org/html/2309.11089>
Probabilistic MBRL
15.03.2024

[MM:Web27] https://www.tu-ilmenau.de/fileadmin/Bereiche/IA/neurob/Publikationen/conferences_int/2009/Hans-ICANN-2009.pdf
MPC
15.03.2024

[MM:Web28] <https://www.mdpi.com/2504-446X/7/4/228>
Uncertainty Propogation
15.03.2024

[MM:Web29] <https://www.youtube.com/watch?v=uja8sxJbplg>
Planning and Learning
15.03.2024

- [MM:Web30] <https://www.researchgate.net/publication/368281498/figure/>
fig5/AS:11431281175426403@1689731781533/Model-Based-Planning-and-
Learning.png
Planning and Learning
15.03.2024
- [MM:Web31] https://en.wikipedia.org/wiki/Artificial_neuron
Artificial Neurons
15.03.2024
- [MM:Web32] <https://builtin.com/data-science/gradient-descent>
Gradient Descent
16.03.2024
- [MM:Web33] [https://www.mltut.com/stochastic-gradient-descent-a-super-easy-
complete-guide/](https://www.mltut.com/stochastic-gradient-descent-a-super-easy-complete-guide/)
Gradient Descent
16.03.2024
- [MM:Web34] [https://www.analyticsvidhya.com/blog/2020/11/popular-classification-
models-for-machine-learning/](https://www.analyticsvidhya.com/blog/2020/11/popular-classification-models-for-machine-learning/)
Classification and Regression
16.03.2024
- [MM:Web35] https://www.youtube.com/watch?v=W01tlRP_Rqs
Supervised und Unsupervised Learning
16.03.2024
- [MM:Web36] [https://medium.com/@manilwagle/association-rules-unsupervised-
learning-in-retail-69791aef99a](https://medium.com/@manilwagle/association-rules-unsupervised-learning-in-retail-69791aef99a)
Association
16.03.2024
- [MM:Web37] <https://www.geeksforgeeks.org/clustering-in-machine-learning/>
Clustering
16.03.2024
- [MM:Web38] <https://www.youtube.com/watch?v=3uxOyk-SczU>
Dimensional reduction
16.03.2024
- [MM:Web39] [https://www.analyticsvidhya.com/blog/2020/11/popular-classification-
models-for-machine-learning/](https://www.analyticsvidhya.com/blog/2020/11/popular-classification-models-for-machine-learning/)
Clustering
16.03.2024

[MM:Web40] https://en.wikipedia.org/wiki/Monte_Carlo_tree_search
MCTS
16.03.2024

[MM:Web41] <https://courses.cs.washington.edu/courses/cse599i/18wi/resources/lecture19/lecture19.pdfh>
MCTS
16.03.2024

[MM:Web42] <https://towardsdatascience.com/neural-networks-backpropagation-by-dr-lihi-gur-arie-27be67d8fdce>
Backpropogation and Forward Pass
21.03.2024

[MM:Web43] <https://www.jmlr.org/papers/volume21/19-060/19-060.pdf>
Model Based and Model Free Combined
21.03.2024

[MM:Web44] <https://openreview.net/pdf?id=HkPCrEZ0Z>
Model Based and Model Free Algorithm
21.03.2024

[AK:Web01] <https://towardsdatascience.com/reinforcement-learning-explained-visually-part-3-model-free-solutions-step-by-step-c4bbb2b72dcf>
Reinforcement Learning Explained Visually (Part 3): Model-free solutions, step-by-step
23.11.2023

[AK:Web02] <https://www.linkedin.com/pulse/reinforcement-learning-sumit-s-patil>
Reinforcement Learning
23.11.2023

[AK:Web03] <https://bdtechtalks.com/2022/06/13/model-free-and-model-based-rl/>
A gentle introduction to model-free and model-based reinforcement learning
29.11.2023

[AK:Web04] [https://en.wikipedia.org/wiki/Model-free_\(reinforcement_learning\)](https://en.wikipedia.org/wiki/Model-free_(reinforcement_learning))
Model-free (reinforcement learning)
23.11.2023

[AK:Web05] https://en.wikipedia.org/wiki/Neural_network
Neural network
31.12.2023

[AK:Web06] <https://de.wikipedia.org/wiki/Markow-Entscheidungsproblem>
Markov-Entscheidungsproblem
2.1.2024

- [AK:Web07] https://en.wikipedia.org/wiki/Markov_decision_process
Markov decision process
8.1.2024
- [AK:Web08] https://en.wikipedia.org/wiki/Reinforcement_learning
Reinforcement Learning
8.1.2024
- [AK:Web09] <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3995763/>
The Exploration-Exploitation Dilemma: A Multidisciplinary Framework
9.1.2024
- [AK:Web10] <https://towardsdatascience.com/the-exploration-exploitation-dilemma-f5622fbe1e82>
The exploration-exploitation trade-off: intuitions and strategies
16.1.2024
- [AK:Web11] https://en.wikipedia.org/wiki/Monte_Carlo_method
Monte Carlo Method
24.1.2024
- [AK:Web12] https://en.wikipedia.org/wiki/Monte_Carlo_tree_search
Monte Carlo Tree Search
24.1.2024
- [AK:Web13] <https://www.scaler.com/topics/artificial-intelligence-tutorial/monte-carlo-tree-search/>
Monte Carlo tree search
30.1.2024
- [AK:Web14] <https://en.wikipedia.org/wiki/Q-learning>
Q-learning
1.2.2024
- [AK:Web15] <https://medium.com/@evertongomede/deep-q-networks-dqn-bridging-the-gap-between-deep-learning-and-reinforcement-learning-5cd73d644c7>
Deep Q-Networks (DQN)
5.2.2024
- [AK:Web16] <https://huggingface.co/blog/deep-rl-a2c>
Advantage Actor Critic (A2C)
6.2.2024
- [AK:Web17] <https://towardsdatascience.com/soft-actor-critic-demystified-b8427df61665>
Soft Actor-Critic Demystified
8.2.2024

[AK:Web18] <https://towardsdatascience.com/evaluation-of-rl-policies-14a9443d3554>
Evaluation of Reinforcement Learning Policies
9.2.2024

[AK:Web19] https://en.wikipedia.org/wiki/Proximal_Policy_Optimization
Proximal Policy Optimization
10.2.2024

[AK:Web20] <https://huggingface.co/blog/deep-rl-ppo>
Proximal Policy Optimization (PPO)
10.2.2024

[AK:Web21] <https://medium.com/@oleglatypov/a-comprehensive-guide-to-proximal-policy-optimization-ppo-in-ai-82edab5db200>
A Comprehensive Guide to Proximal Policy Optimization (PPO) in AI 10.2.2024

[AK:Lit01] https://is.mpg.de/fileadmin/user_upload/files/publications/Peters_EOMLA_submitted_01.pdf
Policy Gradient Methods
6.2.2024

[AK:Img01] https://www.researchgate.net/figure/Epsilon-greedy-method-At-each-step-a-random-number-is-generated-by-the-model-If-the_fig2_334741451
Epsilon Greedy Method
8.2.2024

[AK:Img02] https://www.researchgate.net/figure/Diagram-of-the-soft-actor-critic-SAC-algorithm-for-the-real-time-control-of-a-wave_fig3_345014007
Figure 3
15.3.2024

[AK:Img03] <https://www.javatpoint.com/exploitation-and-exploration-in-machine-learning>
Exploitation and Exploration in Machine Learning
15.03.2024

Kapitelzuordnung

| Kapitel | Autor |
|------------------------------------|----------------|
| Hinderniserkennung und -einplanung | Philip Fenk |
| Pathfinding | Emilio Zottel |
| Model Based Reinforcement Learning | Marco Molnár |
| Model Free Reinforcement Learning | Adrián Kalapis |

Arbeitsprotokolle

Philip Fenk

| KW | Beschreibung | Stunden |
|---------------|---|---------|
| 2023 | | |
| 36 | LaTeX-Einführung, Recherche Aufbau Diplomarbeit | 10,5 |
| 40 | Grobe Erstellung Inhaltsverzeichnis | 2 |
| 41 | Verfeinerung Inhaltsverzeichnis, Overleaf-Setup | 5 |
| 44 | Recherche Frühere Anwendungen und ihre Fortschritte | 8 |
| 46 | Umstrukturierungen, Aktuelle Anwendungen | 9 |
| 47 | Aufbau/ Einteilung LiDAR-Sensor, Klassifizierung | 13,5 |
| 48 | Funktionsweise | 7,5 |
| 51 | Ergänzung Aufbau Sensor | 2 |
| 52 | Vorteile/ Nachteile LiDAR, Datenspeicherung, Aufbau Sensor | 6 |
| 2024 | | |
| 1 | Sensortechnologien, Radar/ Ultraschall Funktionsweise | 13 |
| 2 | Radar Vergleich LiDAR | 3 |
| 4 | Phasenverschiebung LiDAR, Polygonal Type LiDAR Sensor | 10,5 |
| 5 | Radar Einsatzgebiet, Punktfolge, Code Projektbezug | 15 |
| 6 | PointNet, PointNet++, Sensordatenfusion, Auswahl LiDAR-Sensor | 28 |
| 7 | Verfeinerung Diplomarbeit | 5 |
| 8 | Recherche TOF-Kamera | 4,5 |
| 9 | TOF-Kamera Funktionsweise/ Anwendungsbereich | 9 |
| 10 | TOF-Kamera Vorteile/ Nachteile | 2,5 |
| 11 | Verfeinerung Projektbezug, Einleitung | 5 |
| 12 | Feinschliff + Drucken | 9 |
| Gesamtaufwand | | 168 |

Emilio Zottel

| KW | Beschreibung | Stunden |
|---------------|-------------------------------|---------|
| 2023 | | |
| 36 | TeXstudio-Setup | 2 |
| 37 | Inhaltsverzeichnis | 1 |
| 42 | Umstieg auf Overleaf | 2 |
| 43 | Einleitung | 5 |
| 44 | Graphentheorie | 2 |
| 45 | Titelseite | 4 |
| 46 | Graphentheorie | 14 |
| 47 | Algorithmen | 12 |
| 48 | Graph-Implementierung | 16 |
| 52 | Test-Merge | 1 |
| 2024 | | |
| 1 | Breiten- und Tiefensuche | 2 |
| 2 | Dijkstra-Algorithmus | 15 |
| 3 | Breitensuche | 4 |
| 4 | A*-Algorithmus | 4 |
| 5 | Dijkstra-Algorithmus | 6 |
| 6 | Dijkstra-Algorithmus | 8 |
| 7 | 15-Puzzle-Implementierung | 19 |
| 8 | Bidirektionale Bestensuche | 42 |
| 9 | Projektbezug | 16 |
| 10 | Benchmarking | 22 |
| 11 | Zusammenfassung + Feinschliff | 24 |
| 12 | Feinschliff + Drucken | 17 |
| Gesamtaufwand | | 238 |

Marco Molnár

| KW | Beschreibung | Stunden |
|---------------|--|---------|
| 2023 | | |
| 36 | Setup Diplomarbeit Overleaf / Recherche | 3 |
| 37 | Inhaltsverzeichnis | 2 |
| 42 | Recherche / Quellensuche | 5 |
| 43 | Einleitung | 5 |
| 44 | Neural Networks | 8 |
| 45 | AI Einleitung | 4 |
| 46 | Neural Networks / AI Konzepte | 8 |
| 47 | AI Konzepte | 12 |
| 48 | AI Grundlagen | 17 |
| 52 | Test-Merge | 1 |
| 2024 | | |
| 1 | Implementierung AI Setup | 20 |
| 2 | DQN Agent implementieren | 10 |
| 3 | Model Based Planning | 18 |
| 4 | MPC | 5 |
| 5 | Dynamics Models | 7 |
| 6 | Neural Networks / Activation Functions | 8 |
| 7 | Implementierung Flappy Bird AI | 30 |
| 8 | Environment Setup / MBRL Agent / AI Training | 31 |
| 9 | Projektbezug | 2 |
| 10 | Conclusion AI Implementierung | 3 |
| 11 | Feinschliff | 10 |
| 12 | Korrekturlesen und Verbessern | 10 |
| Gesamtaufwand | | 219 |

Adrián Kalapis

| KW | Beschreibung | Stunden |
|---------------|--|---------|
| 2023 | | |
| 36 | Grundrecherche + Editor Setup | 2 |
| 37 | Themenaufstellung | 2 |
| 41 | Themenbearbeitung | 2 |
| 42 | Recherche | 6 |
| 46 | Kapitel 1 Model Free Reinforcement Learning | 2 |
| 46 | Kapitel 2 Konzepte der Model Free Reinforcement Learning | 3 |
| 47 | Kapitel 2 Konzepte der Model Free Reinforcement Learning | 3 |
| 50 | Kapitel Neural Networks | 4 |
| 2024 | | |
| 2 | Kapitel 4 Grundlagen des Reinforcement Learning | 5 |
| 3 | Kapitel 4 Grundlagen des Reinforcement Learning | 3 |
| 4 | Kapitel 5 Methoden der Machine Learning | 6 |
| 4 | Überarbeitung der derzeitigen Kapitel | 4 |
| 5 | Implementierung von Asteroids | 6 |
| 5 | Bug Fixes für Asteroids | 4 |
| 6 | Implementierung DQN Agent | 13 |
| 6 | Training DQN Agent | 3 |
| 7 | Training DQN Agent | 6 |
| 8 | Kapitel 7 Deep Q-Netzwerke | 10 |
| 8 | Implementierung PPO Agent | 5 |
| 8 | Training PPO Agent | 3 |
| 9 | Kapitel 6 Policy Gradients | 8 |
| 9 | Kapitel 8 Proximal Policy Optimization | 6 |
| 9 | Training und Benchmarking DQN und PPO Agenten | 20 |
| 10 | Kapitel 9 Soft Actor Critic | 5 |
| 10 | Training und Benchmarking DQN und PPO Agenten | 9 |
| 10 | Überarbeitung der Diplomarbeit | 4 |
| 11 | Überarbeitung der Diplomarbeit nach Feedback | 3 |
| 12 | Finale Überarbeitung | 5 |
| Gesamtaufwand | | 152 |

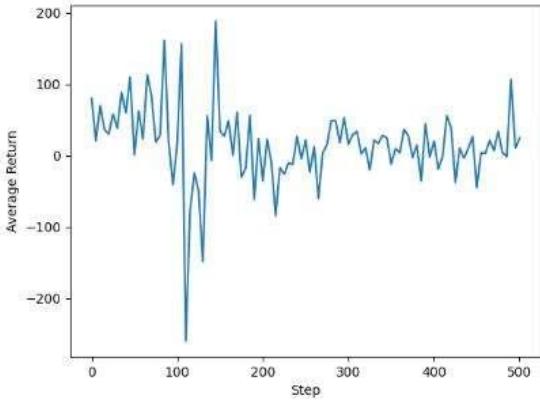
**DIPLOMARBEIT
DOKUMENTATION**

| | |
|---------------------------|--|
| Namen der Verfasser/innen | Philip Fenk Emilio Zottel Marco Molnár Adrián Kalapis |
| Jahrgang Schuljahr | 5AHIF 2023/24 |
| Thema der Diplomarbeit | Einsatz von LiDAR im autonomen Fahren |
| Kooperationspartner | HTBLuVA St. Pölten |

| | |
|------------------|--|
| Aufgabenstellung | Ein optimales Ergebnis, eines autonom fahrenden Fahrzeuges in der Simulationsumgebung CARLA zu erzielen. |
|------------------|--|

| | |
|--------------|--|
| Realisierung | Zum Erreichen des Ziels wurde hauptsächlich die Programmiersprache Python eingesetzt. Zusätzlich wurde das Deep Learning Framework Tensorflow verwendet, um den AI-Agent zu implementieren. Für die Verwendung des LiDAR Sensors und Visualisierung dessen Daten wurde C++ und Processing 4.3 (Java) verwendet. Als Simulationsumgebung, welche es ermöglichte, den Agent effizient zu trainieren, wurde CARLA eingesetzt. Des Weiteren wurde die Python-Library NetworkX angewandt, um dem Agent bestimmte Knoten entlang eines Pfades zu übergeben, welche den kürzesten Pfad zum Ziel repräsentieren. Die Daten wurden in Python bereinigt und dem Agent für das Training übergeben. Für die Visualisierungen wurden die Bibliotheken Matplotlib und Seaborn verwendet. |
|--------------|--|

| | |
|------------|--|
| Ergebnisse | <ul style="list-style-type: none">• Visualisierung und Auswertung der LiDAR Daten• Implementierung eines SAC-Agents in CARLA• Verwendung von Pathfinding in der Simulationsumgebung• Erforschung von verschiedenen RL-Ansätzen• Optimierung der Umgebung für den Agent |
|------------|--|

| | |
|---|---|
| |  |
| Typische Grafik, Foto etc. (mit Erläuterung) |  <p>Das erste Bild zeigt die Trainingssimulation des Autos in CARLA. Die Knoten des Graphen, auf dem das Auto navigiert, sind als farbige Rechtecke visualisiert. Schwarz stellt hier Knoten da, die Teil des Pfades sind, dem es folgen soll. Dieser wird mithilfe des Dijkstra-Algorithmus von der Python-Library NetworkX berechnet.</p> <p>Beim zweiten Bild wird der durchschnittliche Reward über die 500 Episoden angezeigt. Eine Episode dauert an, bis das Auto kollidiert.</p> |

| | |
|--|--|
| Teilnahme an Wettbewerben, Auszeichnungen | |
|--|--|

| | |
|--|--|
| Möglichkeiten der Einsichtnahme in die Arbeit | Die Arbeit befindet sich in der HTBLuVA St. Pölten / Abteilung Informatik (Waldstraße 3, 3100 St. Pölten). |
|--|--|

| | | |
|---------------------------------------|-----------------|--|
| Approbation (Datum / Unterschrift) | Prüfer/Prüferin | Direktor/Direktorin Abteilungsvorstand/Abteilungsvorständin |
|---------------------------------------|-----------------|--|

DIPLOMA THESIS

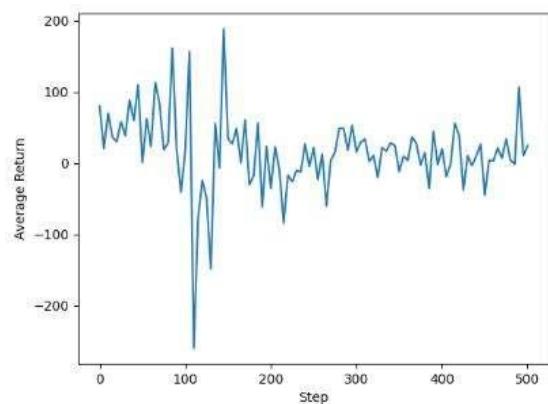
Documentation

| | |
|-----------------------|--|
| Author(s) | Philip Fenk Emilio Zottel Marco Molnár Adrián Kalapis |
| Form Academic year | 5AHIF 2023/24 |
| Topic | Utilization of LiDAR in Autonomous Driving |
| Co-operation partners | HTBLuVA St. Pölten |

| | |
|---------------------|---|
| Assignment of tasks | Attain optimal performance for an autonomous vehicle operating within the CARLA simulation environment. |
|---------------------|---|

| | |
|-------------|--|
| Realisation | To achieve the goal, the programming language Python was predominantly utilized. Additionally, the Deep Learning framework Tensorflow was employed to implement the AI agent. For the utilization of the LiDAR sensor and visualization of its data, C++ as well as Processing 4.3 (Java) were used As a simulation environment that facilitated efficient agent training, CARLA was employed. Furthermore, the Python library NetworkX was used to pass certain nodes along a path to the agent, which represent the shortest path to the goal. The data was cleaned in Python and passed to the agent for training. The libraries matplotlib and seaborn were used for visualizations. |
|-------------|--|

| | |
|---------|--|
| Results | <ul style="list-style-type: none">• Visualization and analysis of LiDAR data• Implementation of an SAC-Agent in CARLA• Utilization of pathfinding in the simulation environment• Exploration of various Reinforcement Learning approaches• Optimization of the environment for the agent |
|---------|--|

| | |
|--|--|
| |  |
| Illustrative graph, photo (incl. explanation) |  <p>The first image shows the training simulation of the car in CARLA. The nodes of the graph on which the car navigates are visualized as colored rectangles. Black represents nodes that are part of the path it should follow. The path is calculated using the Dijkstra algorithm from the Python library NetworkX.</p> <p>In the second image, the average reward over the 500 episodes is displayed. An episode lasts until the car collides.</p> |

| | |
|---|--|
| Participation in competitions Awards | |
|---|--|

| | |
|------------------------------------|---|
| Accessibility of diploma thesis | The work is available at the HTBLuVA St. Pölten / Department of Computer Science (Waldstraße 3, 3100 St. Pölten). |
|------------------------------------|---|

| | | |
|--------------------------------|----------|------------------------------|
| Approval (date / signature) | Examiner | Head of College / Department |
|--------------------------------|----------|------------------------------|