

universität
wien

# DIPLOMARBEIT

Titel der Diplomarbeit

## Optimierendes Lernen unter besonderer Berücksichtigung des Problems des kürzesten Pfades


Verfasser

Judith Fechter


angestrebter akademischer Grad

Magistra der Naturwissenschaften (Mag.rer.nat)

Wien, im Dezember 2012

| | |
|---|---|
| Studienkennzahl lt. Studienblatt: | A 190 406 299 |
| Studienrichtung lt. Studienblatt: | Mathematik |
| Betreuer: | Prof. Dr. Arnold Neumaier |

# Ehrenwörtliche Erklärung

Ich versichere, dass ich die eingereichte Diplomarbeit selbstständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt, und mich auch sonst keiner unerlaubten Hilfsmittel bedient habe. Ich erkläre weiters, dass ich die eingereichte Diplomarbeit weder im Inland, noch im Ausland in irgendeiner Form als wissenschaftliche Arbeit veröffentlicht habe.

Wien, Jänner 2013

Judith Fechter

# Danksagung

Ich möchte mich an dieser Stelle bei all jenen Personen bedanken, die mich bei der Erarbeitung dieser Diplomarbeit unterstützt haben. Mein herzlicher Dank gilt Herrn Prof. Dr. Arnold Neumaier, dem Betreuer meiner Arbeit. Durch seine große Begeisterung für das Forschungsgebiet der Optimierung verbunden mit der Numerischen Mathematik und durch seine zahlreichen Publikationen hat er mein Interesse an diesem Gebiet verstärkt, weshalb ich diesen Themenbereich der Mathematik für meine Diplomarbeit gewählt habe.
Ein großes Dankeschön gilt meinen Eltern und meinen Geschwistern. Sie haben mir nicht nur das Studium finanziell ermöglicht, sondern mich auch in schwierigen Zeiten unterstützt und durch ihre kritischen Anmerkungen und durch ihre intra– und interdisziplinären Denkanstöße und Forderungen einen großen Teil zu meiner Entwicklung beigetragen.
An dieser Stelle möchte ich auch die Korrekturleser und Korrekturleserinnen Karin Fechter, Markus Riedl und Krzysztof Paruch dankend erwähnen, welche sich meiner Arbeit sehr gewissenhaft angenommen haben.
Abschließend möchte ich mich bei meinem Freundeskreis und meiner Kollegenschaft, insbesondere bei Markus Riedl, bedanken. Durch meine Freunde war es mir stets möglich, die Studienzeit zu genießen und den Spaß an der Weiterbildung zu behalten.

Vielen Dank,

Judith Fechter

# Abstract

In recent years the computer has become more and more important in the field of Optimization. As a consequence many methods of solving optimization problems have arisen which take advantage of Numerical Analysis. This diploma thesis gives a short introduction to various methods like Dynamic Programming, Monte Carlo methods and Temporal Difference Learning by working out differences and similarities found in the above mentioned methods. In the final part ways of practical application are presented.

keywords: *optimization, dynamic programming, monte carlo method, temporal difference learning, TD–learning, network flow problem, shortest path problem, algorithm of Bellman, algorithm of Dijkstra*

Abstract

Im Laufe der Zeit gewinnt der Computer als Hilfsmittel im Bereich der Optimierung immer mehr an Bedeutung. So nutzen nun auch viele Optimierungsmethoden die Vorteile der Numerischen Mathematik. Diese Diplomarbeit gibt einen überblicksartigen Einblick in die verschiedenen Methoden des Dynamischen Programmieren, in Monte Carlo Methoden und in Zeitliche – Differenz Methoden. Dabei wurde besonderes Augenmerk auf die Unterschiede und Ähnlichkeiten der eben erwähnten Methoden gelegt und zum Abschluss einige praktsiche Anwendungen behandelt.

Schlüsselwörter: Optimierung, Dynamisches Programmieren, Monte Carlo Methode, zeitliches Differenz Lernen, Netzwerk Fluss Problem, Problem des kürzesten Pfades, Bellman – Algorithmus, Algorithmus von Dijkstra

# Inhaltsverzeichnis


**1 Einführung** **8**

**2 Lernen durch Verstärkung** **10**

2.1 Lernen durch Interaktion mit der Umwelt . . . . . 10

2.2 Elemente von RL . . . . . 12

2.3 Zusammenhang zwischen RL und optimaler Steuerung . . . . . 14

**3 RL als mathematisches Problem** **15**

3.1 Markovsches Entscheidungsproblem . . . . . 15

3.2 Strategie und Wertefunktion . . . . . 18

3.3 Bellman–Gleichungen . . . . . 21

**4 Dynamisches Programmieren** **24**

4.1 Strategieiteration . . . . . 24

4.2 Werteiteration . . . . . 26

**5 Monte Carlo Methoden** **27**

**6 Zeitliche Differenz Methoden als praktischer Lösungsansatz** **29**

6.1 TD Vorhersage . . . . . 30

6.2 TD(0) . . . . . 31

6.3 Agent–Kritiker–System . . . . . 32

6.4 Q–Lernen . . . . . 32

6.5 Ausprobieren und Ausnutzen . . . . . 34

**7 Eine Anwendung: Das optimale Pfad–Problem** **35**

7.1 Das Netzwerk–Fluss Problem . . . . . 36

7.2 Das Problem des kürzesten Pfades . . . . . 38
7.2.1 Bellman–Ford Algorithmus . . . . . 42
7.2.2 Der Algorithmus von Dijkstra . . . . . 45
7.2.3 Darstellungen praktischer Anwendungen . . . . . 47
Literaturverzeichnis . . . . . 64

# Abbildungsverzeichnis


2.1 Interaktion von Agent und Umwelt . . . . . 12

6.1 Agent–Kritiker–System . . . . . 32

7.1 Netzwerk von Pfaden . . . . . 35
7.2 Netzwerk 1 . . . . . 50
7.3 Input Netzwerk 1 . . . . . 50
7.4 Output Netzwerk 1 . . . . . 50
7.5 Netzwerk 2 . . . . . 51
7.6 Input Netzwerk 2 . . . . . 52
7.7 Output Netzwerk 2 . . . . . 52
7.8 Netzwerk 3 . . . . . 53
7.9 Output Netzwerk 3 . . . . . 54
7.10 Input Netzwerk 3 . . . . . 55
7.11 Netzwerk 4 . . . . . 59
7.12 Input Netzwerk 4 . . . . . 60
7.13 Output Netzwerk 4 . . . . . 60
7.14 Netzwerk 5 . . . . . 61
7.15 Input Netzwerk 5 . . . . . 62
7.16 Output Netzwerk 5 . . . . . 63

*„In most sciences one generation tears down what another has built, and what one has established, another undoes. In Mathematics alone each generation adds a new storey to the old structure.“*

**Hermann Hankel**
German mathematician; 1839–1873.

# Kapitel 1

# Einführung

Hermann Hankel, wie in dem Zitat dargelegt, war der Meinung, dass die Mathematik die einzige Wissenschaft sei, in der eine Generation auf dem Wissen, das eine vorhergehende Generation geschaffen hat, aufbauen kann, statt deren Werk zu widerlegen. Dieser Anspruch scheint sich auch im Gebiet des optimierenden Lernens zu bestätigen, denn bei einer historischen Betrachtung stößt man auf zwei Stränge, welche sich unabhängig voneinander stets weiterentwickelt haben. Gegen Ende des 20. Jahrhunderts entstand aus einer Vereinigung dieser Stränge das Gebiet des Optimierenden Lernens.
Einer der Stränge befasste sich mit der Entwicklung der optimalen Steuerung von Prozessen. Zur Lösung diverser Prozesse wurden Kenntnisse über die Dynamische Programmierung und Wissen über die Approximation von Wertefunktionen herangezogen. Der andere Strang beschäftigte sich mit dem Lernen aus Versuch und Irrtum. Hier stand das Lernen im Vordergrund, im Gegensatz zum erst genannten Zweig. So wurden neue Erkenntnisse auch zum Großteil aus der psychologischen Forschung gewonnen und das Lernverhalten von Tieren wurde untersucht.
Beide Stränge entwickelten sich unabhängig voneinander, trafen jedoch bereits bei der Entstehung der Methode des Zeitlichen–Differenz Lernens aufeinander, wie wir in Kapitel 2.3 erfahren werden. Der sich daraus ergebende Strang bildete sodann ab den 1980er Jahren mit den oben genannten beiden Richtungen das Gebiet des Optimierenden Lernens. Diesem Fachgebiet haben Sutton und Barto durch ihre Werke, in denen sie sich mit der Entwicklung komplexer Algorithmen und neuer Theorien zur Approximation von Wertefunktionen

beschäftigten[22][23][24], einen großen Aufschwung verliehen.
Der Strang, welcher sich mit Lernen durch Versuch und Irrtum beschäftigte, ist uns sehr vertraut. In unserer täglichen Erfahrungswelt stoßen wir oft auf Situationen, in denen unsere Lernfähigkeit gefragt ist und wir das Prinzip des Versuchs und Irrtums anwenden. Hat Ihnen je jemand erklärt, in welche Richtung Sie eine Schraube drehen sollen oder in welche Richtung Sie eine Türe öffnen? Auch in der Wissenschaft der Psychologie bildet diese Form des Lernens einen wichtigen Forschungsbereich, der schon eine lange Geschichte durchlaufen hat. Denken wir dabei nur an die zahlreichen Tierversuche, bei denen beispielsweise eine Katze den Ausgang aus einem Käfig finden soll oder eine Ratte versucht, den richtigen Knopf zu finden, um Futter zu erhalten.
Der zweite Strang beschäftigt sich mit dem mathematischen Gebiet der optimalen Steuerung. Dieses Gebiet befasst sich mit der Entwicklung von Abläufen in einem gegebenen dynamischen System, welche ein Optimalitätskriterium erfüllen sollen. Einer der Pioniere auf diesem Gebiet war Richard Bellman, der 1953 einen Lösungsansatz für Probleme der optimalen Steuerung über die Verwendung von Wertefunktionen entwickelte. Er definierte eine optimale Rückgabefunktion, welche auch unter dem Namen Bellman–Gleichung (siehe Kapitel 3.3) bekannt ist.
Da das Gebiet des optimierenden Lernens noch sehr jung ist, gibt es noch viel zu entdecken und zu erforschen. So bot es sich an, dieses Thema für meine Studienabschlussarbeit zu wählen. Das Ziel meiner Arbeit ist es, eine Sammlung von Methoden des optimierenden Lernens für verschiedenste Problemstellungen zu erstellen. Hierzu werde ich mich nach der Darlegung der beiden, oben erwähnten, Stränge und deren Zusammenhang näher mit der Theorie des Dynamischen Programmierens, der Monte Carlo Methode und des zeitlichen–Differenz Lernens befassen, um im Anschluss eine praktische Anwendung des optimierenden Lernens vorzustellen.

# Kapitel 2

# Lernen durch Verstärkung

Dieses Kapitel befasst sich mit der grundlegenden Idee von Lernen durch Verstärkung, die einfach und verständlich dargestellt wird. Für dieses Unterfangen wurde folgende Literatur herangezogen: [23], [6], [12], [27], [17], [14].

## 2.1 Lernen durch Interaktion mit der Umwelt

Lernen durch Interaktion ist eine zentrale Idee, die sehr vielen Theorien der künstlichen Intelligenz und des natürlichen Lernens zugrunde liegt. Betrachtet man ein Kind, das die ersten Schritte versucht, einen jungen Erwachsenen, der sich im Autofahren übt oder eine Katze, die eine Maus schnappen möchte, dann erkennt man, dass all diesen Situationen gemeinsam ist, dass keine expliziten Anweisungen erfolgen, jedoch eine Verbindung zur Umwelt besteht. Durch Anwenden und Üben dieser Verbindung erlangt der Agent Information über Ursache und Wirkung, über Konsequenzen von Handlungen, und ebenso darüber, wodurch ein erwünschtes Ziel erreicht werden kann.
Lernen durch Interaktion bedeutet also, dass der Agent verschiedene Handlungen ausprobiert, Erfolge, Misserfolge und Auswirkungen auf das Umfeld wahrnimmt und anhand dieser Beobachtungen seine Handlungsweisen verbessert. Ein anfängliches Herumirren im Lösen von unbekannten Problemen führt so zu strukturiertem Vorgehen im Erreichen von gesetzten Zielen.
Dieser Prozess findet, wie bereits erwähnt, ohne explizite Anweisungen eines Trainers oder einer Trainerin statt. Der Agent muss daher durch Versuch und Irrtum (engl. trial and error) selbst herausfinden, welche Aktionen in be-

stimmten Situationen zielführend, und welche ineffizient sind. Eine natürliche Rückkopplung hilft dem Agenten bei der Evaluierung von getätigten Handlungen: richtige Aktionen erfahren eine positive Reaktion, falsche Aktionen erfahren eine negative Reaktion. Verlagert ein Kleinkind bei den ersten Versuchen im Fahrradfahren den Schwerpunkt zu sehr nach rechts, wird es unweigerlich umkippen und zu Boden fallen. Dasselbe passiert, wenn der Schwerpunkt zu sehr nach links verlagert wird. Versucht nun das Kleinkind, den Schwerpunkt in der Mitte und so die Balance zu halten, wird es Erfolg haben und die negative Folge „Schmerz“ vermeiden.
In der Psychologie und Verhaltensforschung wurden in der Erforschung von Lernen schon viele Ergebnisse vorgestellt. Von THORNDIKE etwa stammt das „Gesetz der Wirkung“[27]. Dieses besagt, dass Handlungen, welche positive Konsequenzen hervorbringen, in ähnlichen Situationen wahrscheinlich wiederholt werden. Handlungen hingegen, welche eine negative Rückkopplung erzeugen, weisen eine geringere Auftretenswahrscheinlichkeit in der Zukunft auf. Diese Zusammenhänge liegen dem Begriff „Lernen durch Verstärkung“ zugrunde, dessen theoretische Basis beschreibt, wie Verhalten durch Belohnung und Bestrafung optimiert werden kann.
*Optimierendes Lernen* (RL) (engl. Reinforcement Learning) ist nun eine formale Modellierung von dieser Art des Lernens. RL beinhaltet Methoden des maschinellen Lernens, wo Verhalten durch eine Verbindung von Situation und Aktion beschrieben wird. Der lernende Agent beobachtet die Situation und sucht sodann nach einer Aktion, die ihm hilft, in einen neuen, für ihn günstigen Zustand zu gelangen. Die Regel, unter der von Beobachtungen auf Aktionen geschlossen wird, nennt man *Strategie* (engl. policy). Die Methoden des RL versuchen nun, das Vorgehen beim Treffen von Entscheidungen zu optimieren. In vielen Fällen jedoch erhält der Agent nicht nach jeder einzelnen Aktion eine Rückmeldung, sondern muss erst eine Reihe von Aktionen durchführen und auch eine Reihe von Situationen durchlaufen, um eine Reaktion zu erhalten. Stellen wir uns ein Spiel von Backgammon vor. Ein Spieler muss erst viele Spielzüge durchführen, um zu gewinnen oder zu verlieren. Doch es ist klar, dass nicht nur der letzte Spielzug das Spiel entscheidet, sondern viele einzelne Brettsituationen für den Ausgang des Spiels günstig oder ungünstig sind. In solchen Fällen spricht man von *verzögerter Belohnung* (engl. delayed reward). Eine andere Eigenschaft von RL ist, dass Auswirkungen von Aktionen nicht

deterministisch sind. Auch die künstliche Intelligenz muss mit stochastischen Ergebnissen zurechtkommen, wie sie auch in der realen Umwelt alltäglich sind. Hier ist der entscheidende Faktor die Erfahrung. Erfahrungen verarbeiten, bewerten und wiedereinsetzen zu können, hilft dem Agenten, die Konsequenzen seiner Aktionen auf lange Sicht einzuschätzen. Hierbei ist zu bedenken, dass auf der einen Seite jede Aktion die nächste Situation und somit die weiteren Handlungsmöglichkeiten bestimmt, und andererseits eine Aktion in ähnlichen, einander gleichenden Situationen nicht die gleichen Auswirkungen haben muss. Erst durch wiederholtes Ausprobieren und die dadurch gewonnene Erfahrung ist es möglich, solche Situationen auf lange Sicht einzuschätzen.

## 2.2 Elemente von RL

Im RL ist das Bezugssystem des lernenden Agenten seine Umgebung. Er beobachtet die Situation seines Umfelds und wählt, basierend auf dieser Wahrnehmung, eine Aktion aus. Der Agent führt diese Handlung durch, ändert somit den Status der Umgebung und erhält eine numerische Auszahlung als Ergebnis. Positive Werte werden als Belohnung betrachtet, negative Werte als Bestrafung. Es ist die Aufgabe des Agenten, den Durchschnitt der Auszahlungen auf lange Sicht zu maximieren.

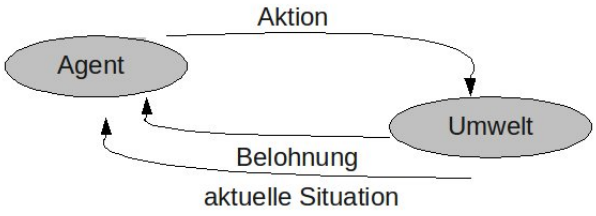



Abbildung 2.1: Interaktion von Agent und Umwelt

Abbildung 2.1 zeigt den Zyklus, in dem Agent und Umwelt interagieren. Der Agent beobachtet die aktuelle Situation und führt eine Aktion aus. Dies führt zum Übergang in einen neuen Zustand und zum Erhalt einer Belohnung.
Die Auszahlungen können vom Agenten subjektiv betrachtet werden, in dem Sinne, dass er die Möglichkeit hat, seine Erfahrungen eigens anhand der erhaltenen Auszahlungen zu evaluieren. Natürlich ist es unerlässlich, dass der

Agent eine vage Vorstellung von dem zu erreichenden Ziel hat. Nur dann ist es auch möglich, von den erhaltenen Werten Rückschlüsse auf die Ziele des Agenten zu ziehen. Um einen möglichst großen Vorteil aus dem Lernprozess ziehen zu können, muss es dem Agenten möglich sein, dieselben oder zumindest ähnliche Situationen bei verschiedenen Gelegenheiten zu durchlaufen.
Üblicherweise weiß der Agent im Vorhinein nicht, welche Auszahlung er für die Durchführung einer Aktion erhalten, oder welche Auswirkungen diese auf den Status seines Umfelds haben wird. Vor allem weiß er nicht, welche Handlung ihn am ehesten zum Ziel führt. Es existiert aber eine sogenannte *Belohnungsfunktion* (engl. reinforcement function), die allen Situation–Aktion Paaren einen numerischen Wert zuordnet. Diese Funktion wird von der Umwelt je nach Lernproblem und Lernaufgabe definiert. Da diese Auszahlung nicht unmittelbar erfolgt, gibt sie keinen Aufschluss über den langfristigen Wert einer Situation. Stellen wir uns wieder ein Backgammon–Spiel vor. Eine Position einer Spielfigur bzw. ein einzelner Spielzug sagt noch nichts über den Spielausgang aus. Das Ende des Spiels wird durch die in den jeweiligen Situationen ausgewählten Aktionen bestimmt. Die Regel, nach welcher der Agent Aktionen auswählt, nennt man, wie bereits in Kapitel 2.1 erwähnt, Strategie. Der langfristige Wert einer Situation wird nun durch die Summe der Auszahlungen bestimmt, die der Agent erhält, wenn er in dieser Situation startet und bestimmte Aktionen durchführt. Die Aufgabe des Agenten ist es daher, eine optimale Strategie zu finden. Das bedeutet, dass er eine Abbildung von Situationen zu Aktionen so definieren muss, dass der Durchschnitt der erzielten Auszahlungen maximiert wird, wenn der Agent in einer beliebigen Situation startet und solange Aktionen durchführt, bis er einen Zielstatus erreicht hat.
Die Funktion, welche angibt, welche Belohnung der Agent, ausgehend von einer bestimmten Situation, erwarten kann, wird *Wertefunktion* (engl. value function) genannt. Es ist eine Abbildung von Situationen auf die Werte der Situationen. Es gilt, diese zu maximieren. Da sie dem Agenten jedoch im Vorhinein nicht bekannt ist, muss er sie durch Ausprobieren von verschiedenen Aktionen in verschiedenen Situationen und durch Beobachten der erhaltenen Auszahlungen schätzen. Tatsächlich ist dies auch das Problem, das alle RL–Algorithmen behandeln: eine möglichst effiziente Approximation der Wertefunktion.

## 2.3 Zusammenhang zwischen RL und optimaler Steuerung

Bisher haben wir den Begriff „Lernen“ aus der Sicht der Psychologie und Verhaltensforschung diskutiert. Wenn wir dieses Verständnis von Lernen aus der mathematischen Sicht betrachten möchten, stoßen wir alsbald auf den Begriff *optimale Steuerung* (engl. optimal control), welcher in der Mitte der 1950er Jahre durch Richard Bellman geprägt wurde.[1]
Das Optimalitätsprinzip von Bellman ist eine grundlegende Eigenschaft einer optimalen Wertefunktion. Es besagt, dass ein Optimierungsproblem in einer rekursiven Funktionalgleichung dargestellt werden kann, deren Lösung sich aus den optimalen Lösungen der einzelnen Schritte zusammensetzt. Diese Gleichung stellt eine Verbindung zwischen dem Wert einer Situation mit den Werten aller anderen Situationen her. Sie heißt *Bellman–Gleichung* (siehe Kapitel 3.3). Methoden, welche jene Gleichung zur Lösung von optimalen Steuerungsproblemen heranziehen, werden als *dynamisches Programmieren* (DP) (engl. Dynamic Programming) bezeichnet.[1]
Algorithmen der dynamischen Programmierung basieren also auf dem Bellmanschen Optimalitätsprinzip. Es wird dabei davon ausgegangen, dass bereits ein Modell für die Umwelt bekannt ist, was in unserem Fall bedeuten würde, dass dem Agenten die Folgen jeder Aktion bekannt sein müssen. Von Interesse ist aber vielmehr das Lösen und Durchführen von unbekannten Aufgaben. Wir suchen also nach Algorithmen, welche auch ohne ein vorher bekanntes Modell zielführend arbeiten.
Eine mögliche Herangehensweise ist, ein solches Modell für die Umwelt durch Erfahrung anzunähern. Wir werden in den folgenden Kapiteln tatsächlich sehen, dass auf ein vorher bekanntes Modell unter bestimmten Voraussetzungen verzichtet werden kann und nur jene Ergebnisse von Bedeutung sind, welche durch Ausprobieren verschiedener Handlungsmöglichkeiten und Beobachten der jeweiligen Konsequenzen gewonnen werden.
Diesen Zusammenhang zwischen RL und optimaler Steuerung werden wir in dem Sinne nutzen, dass wir die Methoden von RL durch klassisches dynamisches Programmieren erarbeiten werden.

# Kapitel 3

# RL als mathematisches Problem

Jeder Agent hat die Aufgabe, eine Abfolge von Aktionen zu wählen, welche ihn möglichst effizient zu seinem erwünschten Ziel führt. Formal ist ein Agent in RL nun durch ein *Markovsches Entscheidungsproblem* modelliert, das die Geschichte der Umwelt gut in einer hinreichenden Statistik zusammenfasst.[18]

## 3.1 Markovsches Entscheidungsproblem

Ein *Markovscher Entscheidungsprozess* (MDP) (engl. Markov Decision Process) besteht aus 5 Komponenten: Zustände, Aktionen, Abbildungs– und Belohnungsfunktion und dem Diskontierungsfaktor.
Der *Zustand* zur Zeit $t$, für welchen wir die Notation $s(t)$ verwenden, beschreibt die momentane Situation des Agenten in seinem Umfeld. Betrachten wir ein Backgammon–Spiel: der Zustand eines Agenten zum Zeitpunkt $t$ wäre die Position der Figur des Agenten am Spielbrett nachdem $t$ Züge getan worden sind. Die Menge der Zustände trägt die Bezeichnung $\mathcal{S}$. Sie kann diskret oder stetig sein.
Die *Aktion*, welche der Agent zum Zeitpunkt $t$ durchführt, um in einen anderen Zustand zu gelangen, bezeichnen wir mit $a(t)$. Auch die Menge der Aktionen $\mathcal{A}$ kann diskret oder stetig sein.
Der Übergang von einem Zustand in den nächsten wird durch eine *Abbildungs-*

*funktion* $T: \quad S \times A \to \Pi(S)$ mit

$$T^a(s, s') := Pr(s_{t+1} = s' \mid s_t = s, a_t = a) \tag{3.1}$$

für alle $(s, a) \in S \times A$ definiert, wobei $\Pi(S)$ die Wahrscheinlichkeitsverteilung über $\mathcal{S}$ ist und $Pr(\cdot \mid \cdot, \cdot)$ eine bedingte Wahrscheinlichkeit darstellt, welche beschreibt, dass der Agent, wenn er in einem Zustand $s$ die Aktion $a$ ausführt, in den Zustand $s'$ gelangt. Falls $\mathcal{S}$ diskret ist, definieren wir $T^a(s, s')$ als eine Auftretenswahrscheinlichkeit. Falls $\mathcal{S}$ stetig ist, wird $T^a(s, s')$ als Dichtefunktion verstanden.
Eine weitere Konsequenz, nachdem eine Aktion durchgeführt wurde, ist der Erhalt einer numerischen Auszahlung. Eine *Belohnungsfunktion* (siehe Kapitel 2.2) ist durch $R: \quad S \times S \times A \to \mathbb{R}$ mit

$$R^a(s, s') := \mathbb{E}(r_{t+1} \mid s_t = s, s_{t+1} = s', a_t = a) \tag{3.2}$$

definiert, wobei $\mathbb{E}(\cdot \mid \cdot, \cdot, \cdot)$ den bedingten Erwartungswert darstellt, welcher die vom Agenten erhaltenen Werte beschreibt. Die erwartete Belohnung $r_{t+1}$ wird als eine Funktion des aktuellen Zustandes $s_t$, des nachfolgenden Zustandes $s_{t+1}$ und der durchgeführten Aktion $a_t$ dargestellt.
Da es in RL erwünscht ist, dass der Agent möglichst schnell an das vorgegebene Ziel gelangt, existiert ein sogenannter *Diskontierungsfaktor* (engl. discount factor) $\gamma \in (0, 1)$, welcher durchgeführte Aktionen je nach Durchführungszeitpunkt unterschiedlich gewichtet. Je größer $\gamma$ gewählt wird, umso wichtiger sind entfernte Auszahlungen und umso schwieriger ist üblicherweise das Optimierungsproblem.
Als ein Beispiel für einen Markovschen Entscheidungsprozess stellen wir uns ein hypothetisches Experiment vor, in dem eine Ratte in einem Käfig Knöpfe drücken muss, die Licht an- oder ausschalten, um Futter zu erhalten. Angenommen das Licht ist ausgeschalten, so bewirkt das Betätigen des Knopfes $A$, dass das Licht mit einer bestimmten Wahrscheinlichkeit eingeschalten wird. Betätigen des Knopfes $B$ hat keinen Effekt. Angenommen das Licht ist eingeschalten, so hat das Betätigen des Knopfes $A$ keinen Effekt. Das Betätigen des Knopfes $B$ liefert mit einer bestimmten Wahrscheinlichkeit Futter und schaltet das Licht wieder aus. In dieser einfachen Umgebung gibt es nun zwei relevante Situationen: „Licht an“ und „Licht aus“. Knopf $A$ bewirkt vielleicht einen Übergang von „Licht aus“ zu „Licht an“; Knopf $B$ liefert vielleicht Futter. Die einzige Information, welche die Ratte benötigt, um eine Entscheidung

zu treffen, ist, ob das Licht ein- oder ausgeschalten ist. Die optimale Politik ist hier einfach: in „Licht aus“ drücke Knopf $A$, in „Licht an“ drücke Knopf $B$.
Der Agent interagiert also mit seiner Umwelt (siehe Abb. 2.1). Er nimmt zu jedem Zeitpunkt $t$ einer diskreten Folge $t = 1, 2, 3, ...$ seine Umwelt im Zustand $s(t) \in \mathcal{S}$ wahr und führt eine Aktion $a(t) \in \mathcal{A}$ aus. Die Umwelt reagiert mit einer sofortigen Auszahlung $r_t \in [0, 1]$ und geht mit der Wahrscheinlichkeit $T^a(s, s')$ in einen Zustand $s_{t+1}$ über.
Der Vollständigkeit halber möchte ich folgende Begriffsbestimmungen anführen:

- Ein MDP wird *endlich* genannt, wenn sowohl die Menge der Zustände $\mathcal{S}$, als auch die Menge der Aktionen $\mathcal{A}$ endlich ist; im Gegensatz dazu wird ein MDP als *stetig* bezeichnet, wenn entweder die Menge der Zustände $\mathcal{S}$ oder die Menge der Aktionen $\mathcal{A}$ stetig ist.
- Ein MDP wird oft als *unbeschränkt* oder auch als *Markov Kette* bezeichnet, falls $|A| = 1$.
- Ein MDP heißt *stochastisch*, falls die Belohnungen und Übergänge zufällig sind; ansonsten ist es *deterministisch*.

Für die Theorie muss ein MDP die *Markov–Eigenschaft* besitzen. Diese setzt voraus, dass

$$\begin{aligned} T^a(s, s') &= Pr(s_{t+1} = s', r_{t+1} = r \,|\, s_t, a_t, r_t, s_{t-1}, a_{t-1}, r_{t-1}, ..., r_1, s_0, a_0) \\ &= Pr(s_{t+1} = s', r_{t+1} = r \,|\, s_t, a_t) \end{aligned}$$

und besagt, dass alle Informationen, welche für den nächsten Zustand und die Belohnung benötigt werden, in der Erklärung des aktuellen Zustandes und der folgenden Aktion gespeichert sind. Das bedeutet, dass ein Zustand also nur von seinem unmittelbaren Vorgänger abhängt.
Das Ziel des Agenten beim Lösen eines MDP ist es, die Auszahlungen auf lange Sicht zu maximieren. Eine übliche Vorgehensweise in der Mathematik ist, diese Auszahlungen auf lange Sicht als diskontierte Summe der erhaltenen Belohnungen darzustellen. Die Auszahlungen an den Agenten zum Zeitpunkt $t$, notiert als $R_t$, definieren wir als:

$$R_t := \sum_{\tau=0}^{\infty} \gamma^\tau r_{\tau+t} \tag{3.3}$$

Da sich der Umgang mit unendlichen Summen als schwierig erweisen kann, verwenden wir folgendes Lemma, das besagt, dass die unendliche Summe durch eine „geringe“ Anzahl von vorderen Termen approximiert werden kann.

**Lemma 3.1.** *Sei $R_t(H)$ eine H-fach diskontierte Belohnung zum Zeitpunkt t, definiert als*

$$R_t(H) := \sum_{\tau=0}^{H-1} \gamma^\tau r_{\tau+t}$$

*Für ein beliebiges $\epsilon > 0$ und t gilt $0 \leq (R_t - R_t(H)) \leq \epsilon$, falls*

$$H \geq \frac{1}{1-\gamma} \ln \frac{1}{\epsilon(1-\gamma)}. \tag{3.4}$$

**Beweis.** Es ist klar, dass $R_t \geq R_t(H)$, weil alle sofortigen Belohnungen nichtnegativ sind. Auf der anderen Seite gilt, falls H die Gleichung 3.4 erfüllt, dann folgt aus der Ungleichung $\gamma \geq 1 + ln\gamma$

$$H \geq \frac{\ln \frac{1}{\epsilon(1-\gamma)}}{\ln \frac{1}{\gamma}} = \log_\gamma(\epsilon(1-\gamma)),$$

und daher

$$R_t - R_t(H) = \sum_{\tau=H}^{\infty} \gamma^\tau r_{t+\tau} \leq \sum_{\tau=H}^{\infty} \gamma^\tau = \frac{\gamma^H}{1-\gamma} \leq \frac{\epsilon(1-\gamma)}{1-\gamma} = \epsilon$$

sind alle sofortigen Belohnungen von oben durch 1 beschränkt. □

Die Belohnung $R_t$ ist eine zufällige Variable, welche von der Abbildungs- und Belohnungsfunktion des MDP abhängt, und ebenso von den Entscheidungen des Agenten. Sowohl die Abbildungs- als auch die Belohnungsfunktion hängen von der Beschaffenheit der Umwelt ab und können daher nicht vom Agenten beeinflusst werden. Jedoch kann der Agent infolge der Regel, nach der Entscheidungen getroffen werden, seiner Strategie, den Durchschnitt der Belohnungen kontrollieren.

## 3.2 Strategie und Wertefunktion

Formal definiert eine *stationäre Strategie* $\pi \in \mathcal{A}^\mathcal{S}$, auch *Politik* (engl. policy) genannt, eine Aktion–Auswahl–Regel, wobei $\pi(s)$ die ausgewählte Aktion beschreibt, wenn der aktuelle Zustand $s$ ist. Eine stationäre Strategie $\pi$ bestimmt

für einen beliebigen MDP $M$ die Verteilung der Belohnungsfolge und somit die Verteilung der Belohnungen. Daher verwenden wir in Zukunft den Erwartungswert der Belohnung, um eine Strategie zu evaluieren. Wir definieren die *Wertefunktion* $V^\pi(s)$, um Zustände und Aktionen bewerten zu können. Es ist eine Abbildung von Zuständen auf reelle Zahlen. $V^\pi : \quad S \to \mathbb{R}$ mit

$$V^\pi(s) := \mathbb{E}_\pi[R_1 \mid s_1 = s] \tag{3.5}$$

$V^\pi(s)$ beschreibt den Wert eines Zustandes $s$ unter der Strategie $\pi$ , wobei sich $E_\pi$ auf die Wahrscheinlichkeitsverteilung der Belohnungsfolge bezieht, welche sowohl durch die Dynamik des MDP, als auch durch die Politik ($a_t = \pi(s_t)$ für alle $t$) entsteht.
Als ein Beispiel einer Wertefunktion möchte ich einen einfachen Markovschen Entscheidungsprozess mit 16 Zuständen betrachten. Wir betrachten eine 4x4–Gitterwelt, wobei jedes Quadrat einen Zustand repräsentiert. Die Belohnungsfunktion gibt immer den Wert $-1$ zurück. Das bedeutet, dass der Agent für jeden Übergang in einen anderen Zustand $-1$ erhält. In jedem Zustand sind 4 Aktionen möglich: nach oben, nach rechts, nach unten oder nach links. Die zu erreichenden Zustände sind der 'Eckzustand' links oben und der 'Eckzustand' rechts unten. In Abb. 3.1 sehen wir die Wertefunktion für eine beliebige Strategie: in jedem Zustand wird eine der 4 Aktionen zufällig ausgewählt. Die Zahlen in den Zuständen geben den erwarteten Wert der Situation an. Angenommen der Agent startet in der linken Ecke unten, welche den Wert 22 hat. Das bedeutet, dass der Agent im Durchschnitt 22 Zustandsübergänge vor sich hat, bevor er einen der Zielzustände erreicht.

| | | | |
|---|---|---|---|
| 0 | −14 | −20 | −22 |
| −14 | −18 | −22 | −20 |
| −20 | −22 | −18 | −14 |
| −22 | −20 | −14 | 0 |

Tabelle 3.1: beliebige Wertefunktion

Die optimale Wertefunktion ist nun in Abb. 3.2 zu sehen. Wieder startet der Agent in der linken Ecke unten und berechnet den Durchschnitt der Belohnungen. Wenn der Agent die optimale Strategie verfolgt, also jene Strategie, welche den Durchschnitt der Belohnungen maximiert, erhält er als Wert des

Zustandes $-3$, weil nur 3 Zustandsübergänge benötigt werden, um ein Ziel zu erreichen.

| | | | |
|---|---|---|---|
| 0 | $-1$ | $-2$ | $-3$ |
| $-1$ | $-2$ | $-3$ | $-2$ |
| $-2$ | $-3$ | $-2$ | $-1$ |
| $-3$ | $-2$ | $-1$ | 0 |

Tabelle 3.2: optimale Wertefunktion

Falls die optimale Wertefunktion bekannt ist, ist es sehr einfach, die optimale Strategie zu finden. Unter der Annahme, dass der Agent in einem beliebigen Zustand in Abb. 3.2 startet, müsste er einfach jene Aktion auswählen, durch welche er die größte sofortige Belohnung erhält.
Diese Tatsache wirft die Frage auf, welcher Algorithmus uns effizient die optimale Wertefunktion liefert. Wir werden später darauf zurückkommen.
Ähnlich wie die Wertefunktion definieren wir auch die $Q$ - *Funktion* (engl. Q–function) $Q^\pi(s,a)$ als den Erwartungswert nach Auswählen der Aktion $a$ in Zustand $s$ und Folgen einer Strategie $\pi$. $Q^\pi : \quad (S,A) \to \mathbb{R}$ mit

$$Q^\pi(s,a) := \mathbb{E}_\pi[R_1 \mid s_1 = s, a_1 = a] \tag{3.6}$$

$Q^\pi(s,a)$ beschreibt den Nutzen einer Aktion $a$ in einer Situation $s$.
Um den Durchschnitt der Belohnungen zu maximieren, sucht der Agent nach einer optimalen Strategie $\pi^*$, mit den Wertefunktionen $V^*(s)$ und $Q^*(s,a)$, welche folgende Bedingungen erfüllen:

$$\begin{aligned} V^*(s) &= \max_\pi V^\pi(s) \\ Q^*(s,a) &= \max_\pi Q^\pi(s,a) \end{aligned}$$

Zwischen den optimalen Werten einer Situation $V^*(s)$ und den optimalen Zustands–Aktions–Werten $Q^*(s,a)$ besteht folgender Zusammenhang:

$$V^*(s) = \max_a Q^*(s,a)$$

Eine *gierige Strategie* (engl. greedy policy) $\pi_V$ oder $\pi_Q$ bezüglich einer Wertefunktion V oder Q legt für jede Situation eine Aktion fest und maximiert die

sofortige Belohnung und den diskontierten erwarteten Wert des Folgezustandes. Wir definieren für alle $s \in \mathcal{S}$:

$$\pi_V(s) : = \operatorname*{argmax}_{a \in \mathcal{A}} \left( R(s,a) + \gamma \sum_{s \in \mathcal{S}} T(s' \mid (s,a)) V(s') \right) \tag{3.7}$$

$$\pi_Q(s) : = \operatorname*{argmax}_{a \in \mathcal{A}} Q(s,a) \tag{3.8}$$

Eine wichtige Tatsache ist, dass eine optimale Strategie gierig bezüglich der optimalen Wertefunktionen sein muss:

$$\pi^*(s) = \operatorname*{argmax}_{a \in \mathcal{A}} \left( R(s,a) + \gamma \sum_{s \in \mathcal{S}} T(s' \mid (s,a)) V^*(s') \right) \tag{3.9}$$

$$= \operatorname*{argmax}_{a \in \mathcal{A}} Q^*(s,a) \tag{3.10}$$

## 3.3 Bellman–Gleichungen

Bellman–Gleichungen[1] charakterisieren die Struktur einer optimalen Wertefunktion. Sie stellen eine rekursive Beziehung zwischen dem Wert einer Situation und den Werten der möglichen Nachfolgesituationen her und dienen zur Berechnung der optimalen oder einer gegebenen Strategie.
Wir definieren den *Bellman–Operator* $\mathcal{B} \in (\mathbb{R}^{\mathcal{S}})^{\mathbb{R}^{\mathcal{S}}}$ durch

$$\mathcal{B}V(s) := \max_{a \in \mathcal{A}} \mathcal{B}^a V(s), \forall s \in \mathcal{S} \tag{3.11}$$

wobei der Operator $\mathcal{B}^a$, verlinkt mit einer Aktion $a$, wie folgt definiert ist

$$\mathcal{B}^a V(s) := R(s,a) + \gamma \sum_{s' \in \mathcal{S}} T(s' \mid (s,a)) V(s'), \forall s \in \mathcal{S} \tag{3.12}$$

$\mathcal{B}^a V$ stellt also eine neue Wertefunktion dar, wobei sich der Wert $\mathcal{B}^a V(s)$ einer Situation $s$ aus den Wahrscheinlichkeiten für alle möglichen Aktionen $a$ ergibt, dass durch die Strategie $\pi$ jene Aktion $a$ ausgewählt wird, durch die der Agent von Situation $s$ in Situation $s'$ gelangt. Es ergibt sich für $|\mathcal{S}|$ Zustände ein Gleichungssystem von $|\mathcal{S}|$ Gleichungen und $|\mathcal{S}|$ Unbekannten, welches $\mathcal{B}^a V(s)$ als eindeutige Lösung hat. $\mathcal{B}V(s)$ wird durch jene Aktion $a$ bestimmt, welche die gewichtete Summe über die sofortigen Belohnungen und die Werte der Nachfolgezustände maximiert. Jene Gleichung ist in der Literatur als *Bellman Optimalitätsgleichung* bekannt.[1]

Die Eigenschaften von $\mathcal{B}$ bleiben für $\mathcal{B}^a$ erhalten, da $\mathcal{B}^a$ als Bellman–Operator eines neuen MDP betrachtet werden kann, dessen Menge an Aktionen sich auf eine einzelne Aktion $\{a\}$ beschränkt.
Wir werden die Notation $\mathcal{B}$ auch für den Operator der Zustands–Aktions–Funktionen verwenden. Für eine beliebige Q–Funktion $Q \in \mathbb{R}^{S \times A}$ definieren wir also:

$$\mathcal{B}Q(s,a) := R(s,a) + \gamma \sum_{s' \in \mathcal{S}} T(s' \mid (s,a)) \max_{a' \in \mathcal{A}} Q(s',a') \tag{3.13}$$

Eine grundlegende Eigenschaft des Bellman–Operators ist dessen Kontraktion, welche wir formal im folgenden Lemma angeben:

**Lemma 3.2.** *Seien $V_1, V_2 \in \mathbb{R}^{\mathcal{S}}$ zwei beschränkte Zustands–Werte–Funktionen, dann gilt*

$$||\mathcal{B}V_1 - \mathcal{B}V_2||_\infty \leq \gamma ||V_1 - V_2||_\infty$$

*Seien $Q_1, Q_2 \in \mathbb{R}^{\mathcal{S} \times \mathcal{A}}$ zwei beschränkte Zustands–Aktions–Funktionen, dann gilt*

$$||\mathcal{B}Q_1 - \mathcal{B}Q_2||_\infty \leq \gamma ||Q_1 - Q_2||_\infty$$

Die Kontraktionseigenschaft des Bellman–Operators garantiert die Existenz eines Fixpunktes, nach dem Banachschen Fixpunkttheorem.[18] Der Fixpunkt stimmt mit der optimalen Wertefunktion $V^*$ oder $Q^*$ überein. Daher ist das Berechnen von $V^*$ oder $Q^*$ äquivalent zum Lösen der sogenannten Bellman–Gleichung:

$$V = \mathcal{B}V \tag{3.14}$$

$$Q = \mathcal{B}Q \tag{3.15}$$

Falls die Wertefunktion die Bellman–Gleichung nicht exakt löst, $V^* \neq V$ oder $Q^* \neq Q$, so tritt ein *Bellman–Residuum* ungleich Null auf:

$$E(s;V) : = \mathcal{B}V(s) - V(s) \tag{3.16}$$

$$E(s,a;Q) : = \mathcal{B}Q(s,a) - Q(s,a) \tag{3.17}$$

Ein wichtiger Spezialfall der Bellman–Gleichungen ist die Verwendung einer fixen Strategie. Der entsprechende Bellman–Operator wird linear und der MDP

wird zu einer induzierten *Markovkette* (engl. Markov chain); also zu einem stochastischen Prozess mit diskretem Zustandsraum, der durch die Eigenschaft gekennzeichnet ist, dass die Zukunft des Systems nur von der gegenwärtigen Situation abhängt, nicht jedoch von der Vergangenheit. Ein Beispiel dafür wäre ein MDP, welcher eine endliche Anzahl an Zuständen hat und eine fixe Strategie verwendet, dann lösen die Funktionen $V^\pi$ folgendes lineares Gleichungssystem:

$$V^\pi(s) = R(s, \pi(s)) + \gamma \sum_{s' \in \mathcal{S}} T(s' \mid s, \pi(s)) V(s') \tag{3.18}$$

und ähnlich für $Q^\pi$

$$Q^\pi(s, a) = R(s, a) + \gamma \sum_{s' \in \mathcal{S}} T(s' \mid s, a) Q(s', \pi(s')) \tag{3.19}$$

Oft ist die Anwendung der Bellman–Gleichungen jedoch zu aufwendig, da für jeden Zustand eine Gleichung aufgestellt werden muss, wodurch die Komplexität des zu lösenden Gleichungssystems sehr schnell steigen kann. In einem Backgammon–Spiel würden sich beispielsweise ungefähr $10^{20}$ Gleichungen ergeben.[26]

# Kapitel 4

# Dynamisches Programmieren

Dynamisches Programmieren (DP)[1] bietet systematische Methoden zur Lös–ung von Optimierungsproblemen, wobei jenes rasche Ansteigen der Anzahl an Gleichungen vermieden wird, indem die *Markov–Eigenschaft* ausgenutzt wird (siehe Kapitel 3.3).
Allgemein ist DP ein iterativer Prozess und bezeichnet jene Klasse von Algorithmen, welche eine Lösung durch Rückgriff auf Teillösungen berechnen.
Gleichung 3.10 lässt den Schluss zu, dass der Agent optimal agieren kann, sofern er die optimale Wertefunktion $Q^*$ kennt. Tatsächlich versuchen die meisten Algorithmen zur Lösung von einem MDP $Q^*$ zu approximieren.
In den folgenden Kapiteln werde ich zwei der wichtigsten Methoden des DP beschreiben: Strategie– und Werteiteration.

## 4.1 Strategieiteration

Eine Strategie ist eine Vorschrift von Aktionen zu Situationen; beispielsweise eine Anweisung „drücke Knopf A, falls das Licht aus ist“ und „drücke Knopf B, falls das Licht an ist“. Im Allgemeinen können Strategien stochastisch sein und die Wahrscheinlichkeit des Durchführens einer Aktion in jeder Situation beschreiben.
Strategieiteration, oft auch als Politikiteration bezeichnet, ist ein Algorithmus, der direkt nach der optimalen Strategie $\pi^*$ in der Menge der Strategien sucht. Er startet mit einer beliebigen Strategie und geht dann iterativ vor: in jedem Schritt wird die Strategie ausgewertet und im Anschluss verbessert.

Im Auswertungsschritt wird die Wertefunktion mit der aktuellen Strategie ausgewertet. Dieser Vorgang ist äquivalent zum Lösen des folgenden Gleichungssystems:

$$V_{t+1}(s) = R(s, \pi(s)) + \gamma \sum_{s' \in \mathcal{S}} T(s' \mid s, \pi(s)) V(s') \tag{4.1}$$

Der Wert $V_{t+1}(s)$ einer Situation $s$ nach $t+1$ Iterationen ergibt sich also dadurch, dass für alle Strategien und für alle möglichen nachfolgenden Situationen die zu erwartenden Belohnungen $R(s, \pi(s))$ und die geschätzten Werte der Zustände $V_t(s')$ aufsummiert werden.
Um die Strategie $\pi$ zu verbessern, werden die Werte von $V(s)$ herangezogen, um eine neue Strategie $\pi_t$ zu erstellen, welche zumindest so gut ist wie die vorhergehende. Die neue Strategie soll die gierige Strategie bezüglich der Wertefunktion der aktuellen Strategie sein; das bedeutet, dass jene Aktion $a$ ausgewählt wird, welche für die aktuelle Situation $s$ die gewichtete Summe der sofortigen Belohnungen und aktuellen Schätzungen der Werte der Nachfolgezustände maximiert. Wir erhalten:

$$Q(s, a) = R(s, a) + \gamma \sum_{s' \in \mathcal{S}} T(s' \mid s, a) Q(s', \pi(s')) \tag{4.2}$$

Die neue Strategie ist also:

$$\pi'(s) = \underset{a \in \mathcal{A}}{\operatorname{argmax}}\, Q(s, a) \tag{4.3}$$

Der Pseudocode der Strategieiteration ist in Algorithmus 4.1 gegeben:

**Algorithmus 4.1.** *(Strategieiteration)*
***Inputs**: $M=< \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma >$*
*initialisiere $\pi_1 \in \mathcal{A}^{\mathcal{S}}$ beliebig*
***for** $t = 1, 2, 3, \ldots$ **do***
*Strategieauswertung: $V(s) \leftarrow R(s, \pi(s)) + \gamma \sum_{s' \in \mathcal{S}} T(s' \mid s, \pi(s)) V(s')$*
*Strategieverbesserung: $\pi(s) \leftarrow \operatorname{argmax} \left( R(s, a) + \gamma \sum_{s' \in \mathcal{S}} T(s' \mid s, a) V(s') \right)$*
***end for***

Es ist bekannt, dass jede Strategie $\pi_t$ alle vorhergegangenen Strategien $\pi_\tau$ für $\tau < t$ dominiert. Da die Anzahl an deterministischen Strategien in einem endlichen MDP endlich ist ($|\mathcal{A}|^{|\mathcal{S}|}$), ist garantiert, dass der Algorithmus nach endlich vielen Iterationsschritten endet. Sobald zwei aufeinanderfolgende Strategien gleich sind, ist die optimale Strategie gefunden.[18]

## 4.2 Werteiteration

Werteiteration stellt eine sehr einfache Art dar, einen MDP zu lösen. Dieser Algorithmus arbeitet im Raum der Wertefunktionen. Er startet mit einer beliebigen, anfänglich beschränkten Schätzung einer Wertefunktion und wendet auf diese wiederholt den Bellman–Operator (siehe Kapitel 3.3) an. Das geschieht solange, bis sich die Schätzung der Wertefunktion der optimalen Wertefunktion angenähert hat.
Eine einfache Form der Werteiteration ist in Algorithmus 4.2 zu sehen. Der Pseudocode initialisiert die Wertefunktion als Null in allen Zuständen.
Sei $V_t$ eine Schätzung einer Wertefunktion vor der $t$-ten Iteration. Für den Algorithmus der Wertefunktion ist bekannt, dass die Folge der geschätzten Wertefunktionen $[V_t]_{t\in\mathbb{N}}$ sich geometrisch an $V^*$ annähert, was aus der Kontraktionseigenschaft des Bellman–Operators (siehe Lemma 3.2) folgt. Auch die gierige Strategie $\pi_t$ von $V_t$ konvergiert in endlich vielen Iterationen gegen $\pi^*$, auch wenn $V_t$ noch nicht $V^*$ exakt gleicht.
Es sollte erwähnt werden, dass, auch wenn $V_t$ erst am Ende gegen $V^*$ konvergiert, der Algorithmus in der Praxis bereits nach $t$ Iterationen ans Ziel geführt werden kann. Sobald die maximale Änderung nach einem Schritt, also $||V_{t+1} - V_t||_\infty$, unter einen vorgegebenen Grenzwert, ein Abbruchkriterium, sinkt, ist der Algorithmus beendet.[18]

**Algorithmus 4.2.** *(Werteiteration)*
***Inputs**: $M=<\mathcal{S},\mathcal{A},\mathcal{T},\mathcal{R},\gamma>$*
*initialisiere $V(s)\leftarrow 0$ für alle $s\in\mathcal{S}$*
***for** $t=1,2,3,...$ **do***
$V\leftarrow\mathcal{B}V$
***end for***

# Kapitel 5

# Monte Carlo Methoden

Monte Carlo Methoden (MC)[23] setzen, anders als die Methoden des DP, kein vollständiges Wissen über das System der Umwelt voraus. Sie benötigen nur Erfahrung in Form von Beispielen, wie Folgen von Zuständen, Aktionen und Belohnungen, welche der Agent durch Interaktion mit der Umwelt erlangt. Auch wenn ein Modell für die Umwelt benötigt wird, muss dieses nicht, wie im Fall von DP, die vollständige Wahrscheinlichkeitsverteilung von allen möglichen Übergängen enthalten, sondern nur einzelne Übergangswahrscheinlichkeiten. Es ist in überraschend vielen Fällen einfach, einzelne Übergangswahrscheinlichkeiten bezüglich einer gewünschten Verteilung zu erstellen, jedoch sehr schwer, die Verteilung in expliziter Form anzugeben.
Monte Carlo Methoden errechnen also nicht den Durchschnitt aller Rückgabewerte, sondern nur die einer Auswahl oder einer Folge. Um sicher zu stellen, dass die Rückgabewerte definiert sind, werden Monte Carlo Methoden nur für *Episoden* definiert. Zerfällt die Interaktion des Agenten mit seiner Umwelt in Teilfolgen, wobei jede Teilfolge eine endliche Anzahl an Zuständen besitzt, so nennt man eine solche Teilfolge „Episode“. Das bedeutet, dass in den einzelnen Episoden Erfahrungen gesammelt werden. Jede Episode terminiert, unabhängig davon, welche Aktionen ausgewählt werden. Daher ist die Schätzung der Wertefunktion und die Veränderung der Strategie nur vom Abschluss einer Episode abhängig. Das heißt, Monte Carlo Methoden arbeiten schrittweise im Sinne einzelner Episoden.
DP und Monte Carlo gehen beim Lösen eines RL–Problems ähnlich vor. Wie auch bei DP wird die Strategie zuerst ausgewertet, werden die Wertefunk-

tionen $V^\pi$ und $Q^\pi$ für eine beliebige Strategie $\pi$ berechnet, wird dann die Strategie verbessert und wird letztendlich die Strategieiteration generalisiert. Der Unterschied liegt jedoch darin, dass DP vollständige Kenntnis über die Umwelt benötigt, Monte Carlo jedoch nur Erfahrung in Form von Beispielen verwendet.
Angenommen wir möchten die Wertefunktion $V^\pi(s)$ schätzen, also den Wert einer Situation $s$ unter einer Strategie $\pi$. So wird eine Menge an Episoden durchlaufen, wobei einer Strategie $\pi$ gefolgt, und eine Situation $s$ besucht wird. Hier können nun zwei Arten der Schätzung der Wertefunktion $V^\pi(s)$ unterschieden werden:

- Die Wertefunktion $V^\pi(s)$ wird aufgrund des Durchschnitts der nach jeder einzelnen Situation $s$ in jeder Episode erhaltenen Rückgabewerte geschätzt.
- Die Wertefunktion $V^\pi(s)$ wird aufgrund des Durchschnitts der nach jeder ersten Situation $s$ in jeder Episode erhaltenen Rückgabewerte geschätzt. Diese Methode ist in Algorithmus 5.3 zu sehen.

Beide Methoden konvergieren zu der optimalen Wertefunktion $V^*(s)$, falls die Anzahl der durchlaufenen (ersten) Situationen $s$ gegen unendlich wächst.
Um die Strategie zu verbessern, wie auch im Kapitel 4.1 zur Strategieiteration beschrieben, wird die gierige Strategie bezüglich der Wertefunktion berechnet.

**Algorithmus 5.3.** *(Monte Carlo)*
***Inputs**: $M=<\mathcal{S},\mathcal{A},\gamma>$*
*initialisiere $V(s)$ zufällig und $Ret(s)$ als einen leeren Vektor*
*für alle $s \in \mathcal{S}$*
***for** $t = 1, 2, 3, ...$ **do***
*erstelle eine Episode mit $\pi$*
*für jede Situation $s$ in der Episode:*
*$R \leftarrow$ Belohnung nach der ersten Situation $s$*
*füge $R$ zu $Ret(s)$ hinzu*
*$V(s) \leftarrow$ Durchschnitt über $Ret(s)$*
***end for***

# Kapitel 6

# Zeitliche Differenz Methoden als praktischer Lösungsansatz

Das Anwenden der Strategie– oder Werteiteration erfordert, dass dem Agenten die erwarteten Belohnungen und Übergänge bekannt sind. Die Methoden des *zeitlichen–Differenz–Lernens* (TD) (engl. Temporal Difference learning) bilden eine Familie von Algorithmen, welche zur Lösung von RL–Problemen herangezogen wird, wenn

- die Übergangs– und Belohnungsfunktionen vom MDP $M =< \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma >$ nicht im Input gegeben sind.
- der Agent die Dynamik der Übergänge durch Interaktion mit dem MDP $M$ kennenlernen muss, wie zum Beispiel durch Drücken der Knöpfe A oder B.

TD–Methoden nutzen sowohl die Ideen von Monte Carlo, als auch jene vom DP. Wie Monte Carlo können TD–Methoden lernen ohne das System der Umwelt zu kennen; wie DP optimieren sie die Schätzungen basierend auf einem Teil der bisherigen Schätzungen.[23][30]
Methoden des TD sollen eine Vorhersage und Optimierung erarbeiten. Um die optimale Strategie zu finden, verwenden alle drei Methoden (TD, DP, Monte Carlo) eine Form der Strategieiteration. Die Unterschiede liegen im Zugang an das Vorhersage–Problem, also in der Schätzung von $V^{\pi}$.

## 6.1 TD Vorhersage

Sowohl Monte Carlo, als auch TD nutzen die Erfahrung, um das Vorhersage–Problem zu lösen. Gefolgt wird einer gegebenen Strategie $\pi$; die Wertefunktion $V^\pi$ wird in jedem Schritt neu geschätzt. $V(s_t)$ wird aufgrund des folgenden Ereignisses geschätzt; wird also eine Situation $s_t$ zum Zeitpunkt $t$ erreicht, so wartet die Monte Carlo Methode die aktuelle Episode (siehe Kapitel 5) ab und schätzt sodann $V(s_t)$:

$$V(s_t) \leftarrow V(s_t) + \alpha_t \left[R_t - V(s_t)\right], \tag{6.1}$$

wobei $R_t$ der aktuelle Rückgabewert folgend auf den Zeitpunkt $t$, und $\alpha_t$ eine sogenannte *Lernrate* ist. Für die Folge der Lernraten $[\alpha_t]_{t\in\mathbb{N}}$ gilt $\alpha_t \in (0,1)$. Die Lernrate bestimmt die Schrittweite, mit der $V(s_t)$ an $V(s_{t+1})$ und $r_{t+1}$ angepasst wird. Die Lernrate $\alpha$ übernimmt die oben erwähnte Rolle des Diskontierungsfaktors $\gamma$.
Im Gegensatz dazu, muss TD jedoch nur auf den nächsten Schritt warten. Zum Zeitpunkt $t+1$ wird bereits eine Aktualisierung basierend auf der beobachteten Belohnung $r_{t+1}$ vorgenommen und $V(s_{t+1})$ geschätzt:

$$V(s_t) \leftarrow V(s_t) + \alpha \left[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\right] \tag{6.2}$$

Die Aktualisierungen der Monte Carlo Methode basieren also auf $R_t$, während jene der TD Methode aufgrund von $r_{t+1} + \gamma V_t(s_{t+1})$ durchgeführt werden.
Aus Kapitel 3.2 wissen wir:

$$\begin{aligned}
V^\pi(s) &= \mathbb{E}_\pi[R_t \,|\, s_t = s] && (6.3)\\
&= \mathbb{E}_\pi\left[\sum_{\tau=0}^{\infty} \gamma^\tau r_{t+\tau+1} \,|\, s_t = s\right] && (6.4)\\
&= \mathbb{E}_\pi\left[r_{t+1} + \gamma \sum_{\tau=0}^{\infty} \gamma^\tau r_{t+\tau+2} \,|\, s_t = s\right] && (6.5)\\
&= \mathbb{E}_\pi\left[r_{t+1} + \gamma V^\pi(s_{t+1}) \,|\, s_t = s\right]. && (6.6)
\end{aligned}$$

Monte Carlo Methoden schätzen aufgrund der Gleichung 6.3. Da der Erwartungswert in 6.3 nicht bekannt ist, wird die Aktualisierung basierend auf einem geschätzten Rückgabewert durchgeführt. TD Methoden schätzen aufgrund der Gleichung 6.6. Da der Wert $V^\pi(s_{t+1})$ nicht bekannt ist, wird die Aktualisierung basierend auf dem aktuellen Schätzwert $V_t(s_{t+1})$ durchgeführt.

## 6.2 TD(0)

TD(0) ist die einfachste Methode des TD–Lernens. Zunächst wird für eine fixe Strategie $\pi$ die Wertefunktion $V^\pi$ bestimmt. Um die Wertefunktion in jedem Schritt zu aktualisieren, wird folgende Regel angewandt:

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \tag{6.7}$$

Der Agent lernt also nach dem Auswählen der Aktion anhand der beobachteten Zustandsübergänge und des erhaltenen Rückgabewertes; $V(s_t)$ wird aufgrund der erhaltenen Belohnung $r_{t+1}$, sowie der Differenz zum geschätzten Wert der Nachfolgesituation $V(s_{t+1})$ aktualisiert.[23]
Für jede Strategie $\pi$ konvergieren die Werte von $V(\cdot)$ im Mittel gegen $V^\pi(\cdot)$, falls die Lernrate $\alpha$ hinreichend klein und die Varianz der Rückgabewerte beschränkt ist.[23]
Um die optimale Strategie $\pi^*$ und die optimale Wertefunktion $V^*$ zu ermitteln, wird ebenso die Regel 6.7 angewandt. Die Strategie ist jedoch nicht mehr fix, sondern wird an die optimale Strategie iterativ angenähert; dabei wird sie, wie auch schon in Kapitel 4.1 beschrieben, entsprechend der gierigen Strategie bezüglich der geänderten Wertefunktion verändert:

$$\pi^*(s) = \underset{a \in \mathcal{A}}{\operatorname{argmax}} \left( R(s,a) + \gamma \sum_{s \in \mathcal{S}} T(s' \mid (s,a)) V^*(s') \right) \tag{6.8}$$

Die TD Methode TD(0) ist in Algorithmus 6.4 zu sehen:

**Algorithmus 6.4.** *(TD(0))*
***Inputs**: $M=<\mathcal{S}, \mathcal{A}, \gamma>$*
*initialisiere $V(s)$ zufällig für alle $s \in \mathcal{S}$*
***for** $t = 1, 2, 3, \ldots$ **do***
*führe Aktion $a_t$ durch, gegeben durch Strategie $\pi$ für Situation $s_t$*
$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$
***end for***

## 6.3 Agent–Kritiker–System

*Agent–Kritiker–Lernen* (engl. actor–critic–learning) ist eine TD–Variante der Strategieiteration. Der Agent entspricht der Strategie, da jener die Aktionen auswählt. Der Kritiker entspricht der geschätzten Wertefunktion, da jener die vom Agenten ausgewählten Aktionen kritisiert und auswertet. Der Kritiker bewertet die Aktionen durch ein skalares Signal, welches der einzige Ausgabewert des Kritikers ist, wodurch er sowohl das Lernen des Agenten, als auch jenes des Kritikers lenkt, wie es in Abb. 6.1 zu sehen ist:

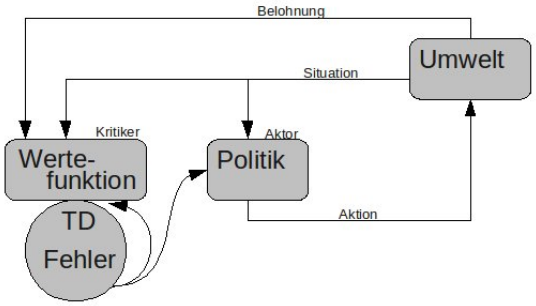



Abbildung 6.1: Agent–Kritiker–System

Üblicherweise entspricht der Kritiker einer Zustands–Werte–Funktion; er wertet nach jeder Aktion die neue Situation aus. Die Bewertung ist über den *TD–Fehler* definiert:

$$\delta_t := r_{t+1} + \gamma V(s_{t+1}) - V(s_t), \tag{6.9}$$

wobei $V(\cdot)$ die, vom Kritiker gebildete, aktuelle Wertefunktion ist. Der TD–Fehler bewertet eine eben ausgewählte Aktion $a_t$, welche in Situation $s_t$ angewendet wird. Ein positiver Fehler unterstützt die Auswahl der ausgewerteten Aktion in der Zukunft.

## 6.4 Q–Lernen

*Q–Lernen* ist eine TD–Variante der Werteiteration. Das Ziel ist, die optimale Wertefunktion eines endlichen MDPs anzunähern.[30]
In Algorithmus 6.5 sehen wir die Vorgehensweise beim Q–Lernen. Der Algorithmus benötigt eine Folge von Lernraten, $[\alpha_t]_{t\in\mathbb{N}}$, wobei $\alpha_t \in (0,1)$. Es

ist bekannt, dass die Wertefunktion beim Q–Lernen asymptotisch mit Wahrscheinlichkeit 1 gegen $Q^*$ konvergiert, solange folgende Bedingungen für jedes Zustands–Aktions–Paar $(s, a)$ erfüllt sind:

$$\sum_{i=1}^{\infty} \alpha_{t_i} = \infty \quad und \quad \sum_{i=1}^{\infty} \alpha_{t_i}^2 < \infty, \tag{6.10}$$

wobei $t_i$ jener Schritt ist, in dem Aktion $a_{t_i}$ in Situation $s_{t_i}$ durchgeführt wird.[29] Diese beiden Bedingungen werden häufig für den Beweis der asymptotischen Konvergenz von stochastischen Approximations–Algorithmen benötigt.[19] Die erste Bedingung garantiert, dass die Folge der Lernraten hinreichend groß ist, um den Anfangswert $Q(s, a)$ in Richtung $Q^*(s, a)$ zu verändern; während die zweite Bedingung garantiert, dass die neuen Werte von $Q(s, a)$ schnell genug fallen, sodass sie letztendlich $Q^*(s, a)$ entsprechen.
Da die Lernrate $\alpha_t$ kleiner als 1 ist, folgt aus der ersten Bedingung in Gleichung 6.10, dass jede Aktion $a$ in jeder Situation $s$ unendlich oft ausprobiert werden muss. Diese Anforderung ist für die Konvergenz von Q–Lernen unvermeidbar; es sei denn, es werden weitere Annahmen getroffen:
wenn der Agent die optimale Funktion $Q^*$ exakt in einem stochastischen MDP lernen möchte, muss er jede Aktion in jeder Situation unendlich oft durchführen, um $R(s, a)$ und $T(\cdot \mid s, a)$ zu erfahren. Diese Tatsache bezieht sich auf das sogenannte *Ausprobieren -Ausnutzen* Dilemma (engl. exploration / exploitation–dilemma), welches in Kapitel 6.5 erklärt wird. Beim optimierenden Lernen ist, neben vielen natürlichen Entdeckungs–Strategien, die *$\epsilon$–gierige* Regel eine gängige Methode. Ein $\epsilon$–gieriger Agent wählt eine gierige Aktion bezüglich der aktuellen Schätzung der Wertefunktion mit einer Wahrscheinlichkeit von $1 - \epsilon$, und eine zufällige Aktion mit einer Wahrscheinlichkeit von $\epsilon$ aus.
Sofern das MDP ergodisch ist, wird jede Aktion in jeder Situation unendlich oft angewendet, und somit konvergiert der Algorithmus des Q–Lernens asymptotisch mit Wahrscheinlichkeit 1 gegen $Q^*$.

**Algorithmus 6.5.** *(Q–Lernen)*
***Inputs**: $M{=}<\mathcal{S}, \mathcal{A}, \gamma>$*
*initialisiere $Q(s, a)$ für alle $(s, a) \in \mathcal{S} \times \mathcal{A}$*
***for** $t = 1, 2, 3, \ldots$ **do***
*beobachte aktuelle Situation $s_t$, führe Aktion $a_t$ durch und beobachte*

*Belohnung $r_t$ und Übergang in eine neue Situation $s_{t+1}$*
*Aktualisierung der Q–Funktion durch*

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_t \left( r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a) - Q(s_t, a_t) \right).$$

***end for***

## 6.5 Ausprobieren und Ausnutzen

Bei den Methoden des TD–Lernens ist es sehr wichtig, eine Balance zwischen Ausprobieren und Ausnutzen zu halten. Die Frage ist, wie der Agent den Durchschnitt seiner Belohnungen maximieren kann, wenn er ein unvollständiges Wissen über seine Umwelt hat. Ohne externe Hilfe muss der Agent mit seiner Umwelt interagieren und experimentieren, um mehr Wissen zu erlangen. Auch wenn der Agent den Durchschnitt maximieren möchte, muss er oft Aktionen ausprobieren, die ihm vorerst suboptimal erscheinen, um herauszufinden, ob das dadurch erlangte Wissen in der Zukunft möglicherweise hilfreich im Erreichen seines Ziels ist. Jedoch ist ein Agent, der exzessiv seine Umwelt erforscht und entdeckt, nicht unbedingt wünschenswert, da ihm der Blick für den Nutzen fehlt. Ebenso wenig brauchbar ist ein Agent, welcher ausschließlich jene Aktionen auswählt, die aktuell am besten erscheinen, da es möglicherweise eine suboptimale Strategie im Auswählen der Aktionen zur Folge hat, da zu wenig Wissen über die Umwelt vorhanden ist. Daher ist eine gute Balance zwischen Ausprobieren und Ausnutzen entscheidend.
Jene Theoreme, die beweisen, dass TD–Methoden üblicherweise funktionieren, benötigen viel Experimentieren: alle Aktionen müssen in allen Situationen angewendet werden. In der Praxis ist es daher üblich, jene Strategien auszuwählen, die „Ausprobieren“ enthalten; das bedeutet, dass manchmal eine zufällige Aktion ausgewählt wird, meistens jedoch jene, die aktuell als die optimale Aktion erscheint.

# Kapitel 7

# Eine Anwendung: Das optimale Pfad–Problem

Unter die Kategorie *Problem des optimalen Pfades* fällt eine große Klasse von Netzwerk–Optimierungsproblemen. In solchen Problemen betrachten wir eine Funktion, welche jedem Pfad, der in einem Netzwerk zwei Knoten verbindet, einen Wert zuordnet. Es wird eine Menge $\mathcal{W}$ festgelegt, welche alle möglichen Wege zwischen gegebenen Start– und Zielknoten enthält. Außerdem wird eine Teilmenge $\mathcal{Q}$ von $\mathcal{W}$ bestimmt, sobald sich der Weg $\mathcal{Q}$ als „besser“ als jeder andere Weg von $\mathcal{Q} \setminus \mathcal{P}$ erweist. Was jedoch ein „guter“ Weg ist, wird in der Problemstellung beschrieben – die Kürze, der Kostenfaktor oder auch die Länge könnte als Kriterium gelten.
Betrachten wir als Beispiel folgendes Netzwerk:

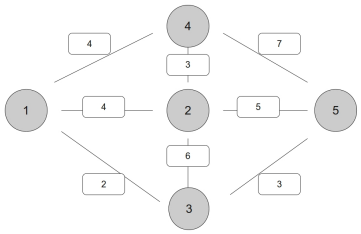



Abbildung 7.1: Netzwerk von Pfaden

Sei nun $\mathcal{W}$ die Menge der Wege von Knoten 1 zu Knoten 5, wobei kein Knoten doppelt oder mehrfach besucht wird. Wir erhalten die Menge
$\mathcal{P} = \{\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}, \mathcal{E}\}$, wobei

- $\mathcal{A} = \{1, (1,3), 3, (3,5), 5\}$,
- $\mathcal{B} = \{1, (1,2), 2, (2,5), 5\}$,
- $\mathcal{C} = \{1, (1,4), 4, (4,5), 5\}$,
- $\mathcal{D} = \{1, (1,3), 3, (3,2), 2, (2,5), 5\}$ und
- $\mathcal{E} = \{1, (1,4), 4, (4,2), 2, (2,5), 5\}$ ist.

Sei nun $d(\mathcal{Q})$ die Distanz des Weges $\mathcal{Q}$ vom Startknoten zum Zielknoten. Wir erhalten $d(\mathcal{A}) = 2 + 3 = 5$, $d(\mathcal{B}) = 4 + 5 = 9$, $d(\mathcal{C}) = 4 + 7 = 11$, $d(\mathcal{D}) = 2 + 6 + 5 = 13$ und $d(\mathcal{E}) = 4 + 3 + 5 = 12$. Angenommen wir suchen nach dem kürzesten Weg, so wäre Weg $\mathcal{A}$ die gesuchte Lösung.
Die eben angeführte Vorgangsweise wäre bei Problemen größerer Dimensionen sehr langwierig und daher nicht geeignet. Daher werden wir in diesem Kapitel Algorithmen zum Lösen solcher Optimierungsprobleme kennenlernen, welche den „besten“ Weg in einer angemessenen Berechnungszeit finden können. Zu diesem Zweck wurden folgende Quellen herangezogen:
[2][4][5][7][8][9][10][11][15][16].

## 7.1 Das Netzwerk–Fluss Problem

Ein *Fluss–Netzwerk* ist ein gerichteter Graph, der aus Knoten und Pfaden besteht. Jeder Pfad verfügt über eine gewisse Kapazität und jeder Pfad erhält einen Fluss an Daten. Ein solches Netzwerk findet im Modellieren von Straßenverkehr oder von Strömungen in einem elektrischen Kreislauf Verwendung. Ein *Netzwerk–Fluss Problem* ist nun ein Optimierungsproblem innerhalb eines Fluss–Netzwerkes. Das Ziel ist, eine bestimmte Größe, wie Kosten, Geschwindigkeit oder Dauer zu minimieren oder zu maximieren. Ein spezielles Problem, welches als Netzwerk–Fluss Problem formuliert werden kann, ist das Problem des kürzesten Pfades. Auf dieses werden wir in Kapitel 7.2 zurück kommen. Zunächst werden wir jedoch ein Fluss–Netzwerk formal definieren. Um einen

umfassenden Überblick zu Netzwerk–Fluss Problemen zu erhalten, sei jedoch auf folgende Schriften verwiesen: [5][8][9][10].
Wir definieren ein *Netzwerk* als einen endlichen, gerichteten Graphen $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, wobei $\mathcal{V}$ die Menge der Knoten und $\mathcal{E}$ die Menge der Pfade bezeichnet. Ein Netzwerk besitzt eine Quelle $s$ (engl. source) und eine Senke $t$ (engl. target) aus $\mathcal{V}$. Jeder Pfad $(u, v) \in \mathcal{E}$, welcher von Knoten $u$ zu Knoten $v$ verläuft, verfügt über eine reellwertige, nicht–negative Kapazität $c(u, v)$, welche durch die Funktion $c: \quad \mathcal{E} \to \mathbb{R}_+$ zugewiesen wird. Falls $(u, v) \notin \mathcal{E}$, wird angenommen, dass $c(u, v) = 0$. Sei nun der *Fluss* innerhalb eines Netzwerks eine reelle Funktion $f: \quad V \times V \to \mathbb{R}$, welche folgende Bedingungen für Netzwerke erfüllt: Eine erste Bedingung sei die Fluss–Erhaltung. Das bedeutet, dass die Menge, welche in einen Knoten fließt plus der, die schon im Knoten ist, gleich der Ausflussmenge sein muss.

$$u + \sum_{j \in \mathcal{I}(u)} f(j, u) = \sum_{j \in \mathcal{O}(u)} f(j, u), \quad \forall u \in \mathcal{V} \tag{7.1}$$

$\mathcal{I}(u)$ bezeichnet dabei die Einflussmenge eines Knotens $u$ und $\sum_{j \in \mathcal{I}(u)} f(j, u)$ den gesamten Fluss in den Knoten $u$. $\mathcal{O}(u)$ bezeichnet die Ausflussmenge eines Knotens $u$ und $\sum_{j \in \mathcal{O}(u)} f(u, j)$ den gesamten Fluss aus dem Knoten $u$.
Eine zweite Bedingung sei, dass der Fluss durch einen Pfad nicht–negativ und nicht größer als dessen Kapazität sein darf.

$$0 \leq f(u, v) \leq c(u, v), \quad \forall (u, v) \in \mathcal{E} \tag{7.2}$$

Ein Fluss $f(u, v)$ ist *zulässig*, falls die zwei eben angeführten Bedingungen erfüllt sind. Ein Fluss–Netzwerk Problem gilt als zulässig, falls es mindestens einen zulässigen Fluss enthält. [9][10]
Bei der Formulierung von Netzwerk–Fluss Problemen können die Knoten und/oder die Pfade zusätzliche Einschränkungen erhalten. Eine mögliche Einschränkung für einen Knoten könnte ein begrenzter Wert der Zufuhr sein. Eine mögliche Einschränkung für einen Pfad könnte eine Begrenzung der Kapazität sein. Im Weiteren werden wir uns einer speziellen Art von Netzwerk–Fluss Problemen zuwenden, dem Problem des kürzesten Pfades.

## 7.2 Das Problem des kürzesten Pfades

Das *Problem des kürzesten Pfades* ist ein klassisches Optimierungsproblem, welches schon seit den 50er Jahren untersucht wird. Um dieses näher zu bearbeiten, wird der Darstellung von [5][9][10][13] gefolgt.
Jeder Pfad $(u, v) \in \mathcal{E}$ des gerichteten Graphen $(V, \mathcal{E})$ besitzt einen reellen Wert $k_{u,v} \in \mathbb{R}$, welcher die Kosten oder die Distanz des Pfades $(u, v)$ beschreibt. Wir betrachten für das Problem des kürzesten Pfades folgende Zielfunktion:

$$k: \quad \mathcal{E} \to \mathbb{R}: \quad (u,v) \to k(u,v) = \sum_{(u,v)\in\mathcal{E}} k_{u,v} \tag{7.3}$$

Es ist das Ziel, einen Pfad $(u, v)^*$ von der Quelle $s$ zur Senke $t$ in $(V, \mathcal{E})$ zu finden, sodass $k^*(u, v)$ minimal wird. Das heißt, dass $k^*(u, v) \leq k(u, v)$ für alle $(u, v) \in \mathcal{E}$ gilt.
Ein Problem des optimalen Pfades heißt *endlich*, falls ein optimaler Pfad existiert, dessen Anzahl an Knoten endlich ist. Ein optimales Pfad Problem heißt *beschränkt*, falls für jede optimale Lösung $(u, v)^*$ der Funktionswert der Kostenfunktion (auch Zielfunktion genannt) $k((u, v)^*) < \infty$ gilt; sonst heißt es *unbeschränkt*.
Mit dem Satz 7.6 werden wir beweisen, dass das Problem des kürzesten Pfades endlich ist, genau dann, wenn das Netzwerk keine Pfade mit negativen Werten enthält.
Um die Notation übersichtlich zu gestalten, bezeichnen wir im Beweis von Satz 7.6 und vom Korollar 7.7 jeden Pfad $(u, v) \in \mathcal{E}$ mit $p \in \mathcal{E}$.

**Satz 7.6.** *Seien $\mathcal{E}_{s,i}$ und $\mathcal{E}_{i,t}$ nicht–leere Mengen für jedes $i \in V$. Das Problem des kürzesten Pfades ist endlich, genau dann, wenn $k(Z) \geq 0$ für jeden Zyklus $Z$ in $(V, \mathcal{E})$.*

**Beweis.** Sei $Z$ ein negativer Zyklus, also $k(Z) < 0$ und sei $i$ ein Knoten von $Z$. Da $\mathcal{E}_{s,i}$ und $\mathcal{E}_{i,t}$ nicht–leer sind, existiert $p_1$, ein Pfad von $s$ nach $i$, und $p_2$, ein Pfad von $i$ nach $t$ in $(V, \mathcal{E})$. Sei nun $p_1 \lhd Z^0 \rhd p_2$ der Pfad von $s$ nach $t$ über $p_1$ und $p_2$. Somit bezeichnet $p_j = p_1 \lhd Z^j \rhd p_2$, mit $j \geq 0$, eine Folge von Pfaden von $s$ nach $t$. Wir erhalten $k(p_j) = k(p_1) + k(Z^j) + k(p_2)$. Daraus kann man schließen

$$lim_{j\to\infty} k(p_j) = k(p_1) + k(Z) lim_{j\to\infty} j + k(p_2) = -\infty \tag{7.4}$$

Somit ist der optimale Pfad $p_\infty = lim_{j\to\infty} p_j$, was bedeutet, dass $p_\infty$ nicht endlich ist. Da $k_{u,v}$ für jeden Pfad $(u, v)$ endlich ist, ist das Problem des kürzesten Pfades nicht endlich.
Sei nun $k(Z) \geq 0$ für jeden Zyklus in $(V, \mathcal{E})$ und sei $p^*$ der optimale Pfad.
Falls $p^*$ keine Zyklen hat, hat $p^*$ auch keine Schleifen und somit auch keine wiederholt besuchten Knoten. Daher ist das Problem endlich.
$p^*$ habe nun zumindest einen Zyklus, somit erhalten wir $p_1 \lhd Z \rhd p_2$. Wenn $k(Z) > 0$ ist, wäre $k(p^*) > k(p_1 \lhd \rhd p_2)$ und somit nicht optimal. Daraus können wir schließen, dass $k(Z) = 0$ gelten muss. Da aber $k(p^*) = k(p_1 \lhd \rhd p_2)$ gilt und somit $p_1 \lhd \rhd p_2$ auch ein optimaler Pfad ist, muss $p^*$ einen weiteren Zyklus, außer $p_1 \lhd \rhd p_2$, besitzen. Wendet man dieses Verfahren wiederholt an, so kann man alle Zyklen aus $p^*$ entfernen und erhält somit einen Pfad ohne Schleifen, woraus sich auf die Endlichkeit des Problems schließen lässt. $p^*$ ist dann die optimale Lösung des Problems. □
Von Satz 7.6 können wir folgende Resultate ableiten.

**Korollar 7.7.** *Ein Problem des kürzesten Pfades ist nicht–endlich, genau dann, wenn es unbeschränkt ist.*

**Beweis.** Falls das Problem des kürzesten Pfades nicht–endlich ist, so existiert ein negativer Zyklus $Z$ in $(V, \mathcal{E})$. Sei $p_\infty = lim_{j\to\infty} p_j$, wobei $p_j = p_1 \lhd Z^j \rhd p_2$ mit $p_1 \in \mathcal{E}_{s,i}$, $p_2 \in \mathcal{E}_{i,t}$ und $i$ ein Knoten in $Z$ ist. Somit gilt $k(p_\infty) = -\infty$ und das Problem ist unbeschränkt. Falls das Problem endlich ist, so hat der optimale Pfad eine endliche Anzahl an Knoten und Wegen und ist somit beschränkt. □

**Korollar 7.8.** *Es existiert ein Pfad ohne Schleifen, welcher die optimale Lösung eines Problem des kürzesten Pfades ist, genau dann, wenn $k(Z) \geq 0$, für jeden Zyklus $Z \in (V, \mathcal{E})$.*

Im Folgenden werden wir das Problem des kürzesten Pfades als ein Netzwerk–Fluss Problem (siehe Kapitel 7.1) betrachten und annehmen, dass keine negativen Zyklen existieren, das heißt $k_{(u,v)} \geq 0$ für alle $(u, v) \in \mathcal{E}$. Außerdem wenden wir uns ab hier dem sogenannten *all–to–one shortest path problem* zu.[9] Dies bedeutet, dass wir den kürzesten aller Pfade zu einem bestimmten Knoten suchen. Dieser Zielknoten wird mit $n$ bezeichnet. Alle von diesem wegführenden Wege haben den Wert 0. Wir nehmen an, dass zumindest ein

Pfad von jedem Knoten $i \neq n$ zum Knoten $n$ existiert, das heißt, dass das Problem zulässig (siehe Kapitel 7.1) ist.
Bezüglich der Kapazität eines jeden Pfades nehmen wir keine Einschränkung vor. Das heißt, dass jeder Pfad $(i, j) \in \mathcal{E}$ im Netzwerk $(V, \mathcal{E})$ eine nach oben unbeschränkte Kapazität $c(i, j) \geq 0$ hat. Da wir den kürzesten aller Pfade zu dem Zielknoten $n$ suchen, ist jeder Knoten $1, ..., n-1$ eine Quelle, welcher jeweils eine Versorgung mit dem Wert 1 liefert, und der Knoten $n$ die einzige Senke, welche einen Bedarf von $n-1$ hat. Da der Knoten $n$ die einzige Senke ist, und Quellknoten keine Versorgung „behalten“ können, muss der Fluss so lange durch das Netzwerk laufen, bis er den Knoten $n$ erreicht.
Ziel ist es nun, den kürzesten der Pfade von einer Quelle zur Senke zu minimieren. Wir minimieren

$$\sum_{(i,j)\in\mathcal{E}} k_{i,j} f_{i,j} \tag{7.5}$$

über alle zulässigen Flüsse, für jeden Knoten $i \neq n$. Da wir Wege mit negativen Werten ausgeschlossen haben, wird der kürzeste Pfad auch die geringsten Kosten haben.
Beschreiben wir nun mögliche Nebenbedingungen an den Fluss, der die Zielfunktion erfüllen soll, näher. Schon in Kapitel 7.1 haben wir das sogenannte Fluss–Erhaltungsgesetz beschrieben, welches besagt, dass die Einflussmenge in einen Knoten gleich der Ausflussmenge sein muss. Konkret für unser Problem formulieren wir diese Bedingung an die Quellknoten und an den Senkknoten wie folgt:

$$\sum_{\{j|(i,j)\in\mathcal{E}\}} f_{j,i} - \sum_{\{j|(i,j)\in\mathcal{E}\}} f_{i,j} = 1, \quad i = 1, ..., n-1 \tag{7.6}$$

$$\sum_{\{j|(i,j)\in\mathcal{E}\}} f_{j,n} - \sum_{\{j|(i,j)\in\mathcal{E}\}} f_{n,j} = 0 \tag{7.7}$$

$\sum_{\{j|(i,j)\in\mathcal{E}\}} f_{j,i}$ bezeichnet den gesamten Fluss, der in den Knoten $i$ fließt und $\sum_{\{j|(i,j)\in\mathcal{E}\}} f_{i,j}$ bezeichnet den gesamten Fluss, der aus dem Knoten $i$ fließt.
Im Hinblick auf die Algorithmen, welche wir zum Lösen von Problemen des kürzesten Pfades heranziehen und in den Unterkapiteln 7.2.1 und 7.2.2 betrachten werden, definieren wir nun die sogenannte Baum–Lösung (engl. tree solution).
Ein Vektor $\mathbf{f}$ heißt *Baum–Lösung*, falls jener Vektor alle Knoten mit dem

Zielknoten verbindet, sodass die Kosten bzw. die Distanz minimal sind. Eine Baum–Lösung kann folgendermaßen konstruiert werden:

- Wähle eine Menge $\mathcal{T} \subset \mathcal{E}$ von $n-1$ Wegen, welche einen Baum formen. Dabei ist die Richtung der Wege unwichtig.
- Setze $f_{i,j} = 0$, für alle $(i,j) \notin \mathcal{T}$.
- Bestimme mit dem Fluss–Erhaltungsgesetz (siehe Gleichung 7.6 und 7.7) $A\mathbf{f} = \mathbf{b}$ die Fluss–Variablen $f_{i,j}$ für alle $(i,j) \in \mathcal{T}$.

Eine Lösung $f_{i,j} \geq 0$ für alle $(i,j) \in \mathcal{E}$ heißt eine *zulässige* Baum–Lösung.[5][9]
An dieser Stelle möchte ich hinzufügen, dass jedes Optimierungsproblem, unter der Voraussetzung, dass alle aufgestellten Nebenbedingungen erfüllt sind, als ein *duales Problem* neu dargelegt werden kann. Die neue Formulierung beinhaltet eine Variable für jede Nebenbedingung des ursprünglichen Problems. Der Wert der dualen Variable wird so gewählt, dass die zugehörige ursprüngliche Nebenbedingung erfüllt wird. Die Nebenbedingungen des dualen Problems bestehen aus zwei Einheiten.
Der duale Graph beinhaltet einen Knoten für jede duale Variable und Pfade für Beziehungen zwischen den dualen Variablen. [20][3]
Mit folgendem Satz über die Lösbarkeit eines Problems sei auf [9] verwiesen.

**Satz 7.9.** *Gegeben sei ein Problem des kürzesten Pfades als ein Netzwerk–Fluss Problem mit einem gerichteten Graphen $(V, \mathcal{E})$, wobei keine negativen Zyklen enthalten sind, also $k_{i,j} \geq 0, \forall (i,j) \in \mathcal{E}$. Von jedem Knoten $i \neq n$ existiert mindestens ein Pfad zum Zielknoten $n$. $n$ besitzt keine wegführenden Wege. Dann gilt:*

1. *Wenn eine zulässige Baum–Lösung optimal ist, dann gibt diese Lösung den kürzesten Pfad an.*
2. *Sei $\boldsymbol{p}$ eine duale Variable. Wenn $p_n = 0$ gilt, so hat das duale Problem eine eindeutige Lösung $\boldsymbol{p}^*$ und $p_i^*$ ist die Länge des kürzesten Pfades von Knoten $i$ zu $n$.*

**Beweis.**

1. kann durch folgende Gedankenführung bewiesen werden: aufgrund der Voraussetzung $k_{i,j} \geq 0, \forall (i,j) \in \mathcal{E}$ sind die Wege von den „Blättern“ zu

der „Wurzel“, dem Zielknoten, gerichtet. Angenommen es existiert ein Pfad von einem Knoten $i$ zu dem Knoten $n$, welcher kürzer ist als jener in der Baum–Lösung. Somit wäre die Baum–Lösung nicht optimal, da ein Fluss schneller über Knoten $i$ geführt werden könnte. Daher liefert die optimale, zulässige Baum–Lösung die kürzesten Pfade.

2. Laut Voraussetzung sei $p_n^* = 0$. Sei $(p_1^*, ..., p_{n-1}^*)$ ein Vektor der dualen Variablen. Da, laut Voraussetzung, alle Wege zum Zielknoten $n$ gerichtet sind, erhalten wir die Länge des Pfades von $i$ nach $n$ durch Aufsummieren der Kosten $k_{i,j}$ der einzelnen Wege. Für jeden Weg im Baum gilt: $p_i^* = c_{i,j} + p_j^*$. Wir erhalten mit $c_{i,j} = p_i^* - p_j^*$:

$$\begin{aligned} \sum c_{i,j} &= \sum p_i^* - p_j^* && (7.8)\\ &= \sum p_i^* - p_0^* && (7.9)\\ &= p_i^* && (7.10) \end{aligned}$$

Da die Lösung eine optimale, zulässige Baum–Lösung ist, ist jeder Pfad der kürzeste Pfad von einem bestimmten Knoten. So ist $p_i^*$ die Länge des kürzesten Pfades von $i$ nach $n$. Außerdem ist die Lösung nicht entartet, da jeder Weg $(i, j)$ die Versorgung von Knoten $i$ zu tragen hat. Daher ist die Lösung eindeutig. □

### 7.2.1 Bellman–Ford Algorithmus

Der Bellman–Ford Algorithmus basiert auf dem Optimalitätsprinzip von Bellman (siehe Kapitel 3.3).[1] Dieses stammt von Richard Bellman ab. Der Algorithmus wurde schrittweise von Fulkerson[9][10] und Moore[15] weiterentwickelt und wird daher auch als Ford–Fulkerson Algorithmus bezeichnet. Der Algorithmus zieht zum Lösen des Problems die Bellman–Gleichung (siehe Kapitel 3.3) heran.
Der Bellman–Ford Algorithmus ist ein gieriger Algorithmus (siehe Kapitel 3.2). Das bedeutet, dass er Lösungsmethoden heranzieht, welche überprüfen, ob eine bessere Langzeitstrategie existiert. Er startet bei einem „Wurzel“–Knoten und erforscht in den folgenden Schritten die benachbarten Knoten. Diese Vorgehensweise wird *Breitensuche* (engl. breadth–first search, BFS) genannt. Das Vorgehen des Bellman–Ford Algorithmus kann wie folgt beschrieben werden:

- Finde den kürzesten Pfad von einem festgelegten Knoten unter Beachtung der Nebenbedingung, dass maximal ein Verbindungsglied verwendet werden darf.
- Finde den kürzesten Pfad unter der Nebenbedingung, dass maximal zwei Verbindungsstücke besucht werden dürfen.
- Erhöhe die Anzahl der maximal besuchten Verbindungsglieder so lange, bis der Algorithmus endet.

Bei diesem Verfahren sind Wege mit negativen Werten $k_{i,j} \leq 0$, $\forall (i,j) \in \mathcal{E}$ zugelassen, vorausgesetzt das Netzwerk beinhaltet keine Zyklen, deren Gesamtwert negativ ist. Der Bellman–Ford Algorithmus kann sowohl zur Berechnung des kürzesten Pfades zu allen Knoten von einem festgelegten Startknoten herangezogen werden, als auch zur Berechnung des kürzesten Pfades von allen Knoten zu einem festgelegten Zielknoten. Letztgenannte Aufgabenstellung haben wir bereits in Kapitel 7.2 näher behandelt.
Stellen wir nun das Problem des kürzesten Pfades als ein duales Problem dar. Das Problem des kürzesten Pfades lautet:

$$min \sum_{(i,j)\in\mathcal{K}} k_{i,j} f_{i,j} \tag{7.11}$$

mit den Nebenbedingungen

$$\begin{aligned} s.t. \quad \sum_{\{j|(i,j)\in\mathcal{K}\}} f_{j,i} - \sum_{\{j|(i,j)\in\mathcal{K}\}} f_{i,j} &= 1, \quad i = 1, ..., n-1 && (7.12)\\ \sum_{\{j|(i,j)\in\mathcal{K}\}} f_{j,n} - \sum_{\{j|(i,j)\in\mathcal{K}\}} f_{n,j} &= 0 && (7.13)\\ f_{i,j} &\geq 0, \quad \forall (i,j) \in \mathcal{K} && (7.14) \end{aligned}$$

Das duale Problem sei, mit $p_n = 0$, folgendermaßen formuliert:

$$max \sum_{i=1}^{n-1} p_i \tag{7.15}$$

mit der Nebenbedingung

$$s.t. \quad p_i \leq k_{i,j} + p_j, \quad \forall i,j \in \mathcal{E} \tag{7.16}$$

Wenn nun alle Komponenten von $\mathbf{p}$, ausgenommen $p_i$, festgelegte Werte sind, so sollte $p_i$ den größten Wert haben. Das bedeutet unter Berücksichtigung der Nebenbedingung

$$p_i = min_{j \in \mathcal{O}(i)} k_{i,j} + p_j \tag{7.17}$$

Die optimale Lösung $\mathbf{p}^*$ erfüllt somit die Gleichung

$$p_i^* = min_{j \in \mathcal{O}(i)} k_{i,j} + p_j^*, \quad i = 1, .., n-1 \tag{7.18}$$

wobei $p_n^* = 0$. Somit erhalten wir ein System von $n-1$ nicht–linearen Gleichungen mit $n-1$ Unbekannten, welches als die Bellman–Gleichung (siehe Kapitel 3.3) bekannt ist.
Falls alle Zyklen positive Werte haben, so ist die Lösung der Bellman–Gleichung, welche dem kürzesten Pfad entspricht, eindeutig. Falls alle Zyklen nicht–negative Werte haben, so ist die Länge des kürzesten Pfades die maximale Lösung der Bellman–Gleichung.

**Algorithmus 7.10.** *(Bellman–Ford Algorithmus)*

***Inputs**: $G = (V, \mathcal{E}), s \in V, k : \mathcal{E} \to \mathbb{R}$, Startknoten $s$, Zielknoten $t$*
*initialisiere* $p(s) \leftarrow 0, v(s) \leftarrow Nil$
***foreach*** $i \in V \setminus s$ ***do***
$p(i) \leftarrow \infty, v(i) \leftarrow Nil$
*end* ***foreach***
***for*** $n = 1, ..., \mid V \mid -1$ ***do***
***foreach*** $(i, j) \in \mathcal{E}$ ***do***
***if*** $p(i) > p(j) + k(j, i)$ ***do***
$p(i) = p(j) + k(j, i)$
$v(i) = j$
*end* ***if***
*end* ***foreach***
*end* ***for***
***for*** $(j, i) \in \mathcal{E}$ ***do***
***if*** $p(i) < p(j) + k(i, j)$
***return*** *false*
*end* ***if***
*end* ***for***
***return*** *true*

Da jeder Pfad ($\mid V \mid -1$)mal untersucht wird, beträgt die Laufzeit des Bellman–Ford Algorithmus $\mathcal{O}(\mid V \mid -1 \times \mathcal{E}) = \mathcal{O}(V\mathcal{E})$.[28]

### 7.2.2 Der Algorithmus von Dijkstra

Der Algorithmus von Dijkstra ist einer der effizientesten Algorithmen für das Lösen des Problems des kürzesten Pfades. Im Vergleich zum Bellman–Algorithmus ist er zwar ebenso ein gieriger Algorithmus, liefert jedoch nur für gerichtete Graphen mit nicht–negativen Pfaden ein korrektes Ergebnis. Daher sei ab nun $k_{i,j} \geq 0$ für alle $(i,j) \in \mathcal{E}$ vorausgesetzt.[8][28]
Der Algorithmus von Dijkstra geht iterativ vor. Jeder Knoten trägt eine Bezeichnung, welche aus zwei Elementen besteht. Das erste Element speichert die Länge des kürzesten Pfades, welcher bis zur vorhergehenden Iteration gefunden wurde. Falls keine Verbindung bis zu diesem Punkt gefunden wurde, so wird die Länge mit $\infty$ bezeichnet. Sobald jedoch das erste Element der Bezeichnung einen endlichen Wert besitzt, speichert das zweite Element den Vorgänger des Knotens. Der Vorgänger des Quellknotens $s$ wird mit $s$ bezeichnet. Diese Bezeichnung wird als 2–dimensionaler Vektor weitergegeben.
Im Wesentlichen startet der Algorithmus bei einem Knoten, besucht jeden Knoten in seiner Umgebung so lange, bis er den Knoten mit der kürzesten Entfernung gefunden hat. Die Grundidee ist, die Knoten nach der kürzesten Entfernung zum Startknoten zu bezeichnen, wodurch die Länge des kürzesten Pfades im Fortschreiten des Algorithmus berechnet wird.
In jeder Iteration kann die Menge der Knoten $V$ in zwei Teilmengen $V_1, V_2$ geteilt werden. $V_1$ beinhaltet die *dauerhaften* Knoten, über welche ausgesagt werden kann, dass der Pfad, der vom Startknoten zu diesem Knoten gefunden wurde, der kürzeste Pfad ist. $V_2$ beinhaltet alle anderen, die sogenannten *temporären* Knoten. Da $k_{i,j}$ für alle $(i,j) \in \mathcal{E}$ nicht–negativ sind, ist in der Menge $V_1$ stets der Quellknoten $s$ enthalten und somit ist $V_1$ niemals leer. [5][21][28]
In jeder Iteration untersucht der Algorithmus alle Knoten $v \in V_2$, um zu überprüfen, ob der von $s$ nach $v$ gefundene Pfad verbessert werden könnte. Hierbei wird überprüft, ob sich ein Knoten $u \in V_1$ befindet, sodass ein Pfad $(u,v) \in \mathcal{E}$ von $u$ nach $v$ besteht, sodass $p_v > p_u + k_{u,v}$ erfüllt ist. Falls kein solcher Knoten $u \in V_1$ existiert, wird während dieser Iteration die Bezeichnung in $V_2$ beibehalten. Falls ein solcher Knoten $u_0$ existiert, wählt der Algorithmus diesen aus und überschreibt den Wert von $p_v$ mit dem neuen Wert $p_{u_0} + k_{u_0,v}$.

Wenn dieser Vorgang während einer Iteration abgeschlossen wird, wählt der Algorithmus einen Knoten $v_0$ aus $V_2$ aus, dessen Wert $p_{v_0}$ der Kleinste unter allen Knoten in $V_2$ ist. Diesen Knoten transferiert der Algorithmus von den temporären Knoten aus $V_2$ zu den dauerhaften Knoten aus $V_1$. Der Grund für diesen Transfer ist folgender: der kürzeste Pfad von $s$ zu $v_0$ könnte entweder nur aus Knoten von $V_1$ bestehen oder auch zwischenliegende Knoten aus $V_2$ besitzen. Im zweiten Fall würde der Vorgänger von $v$ in $V_1$ liegen, wobei der Algorithmus bereits zu allen Knoten aus $V_1$ einen kürzesten Pfad garantieren kann. Daher ist der Pfad nach $v$ auch ein kürzester Pfad.[21]

**Algorithmus 7.11.** *(Algorithmus von Dijkstra)*

***Inputs**: $G = (V, \mathcal{E}), s \in V, k : \mathcal{E} \to \mathbb{R}$, Startknoten $s$, Zielknoten $t$*

*initialisiere* $p(s) \leftarrow 0, v(s) \leftarrow Nil, V_1 = \{s\}, V_2 = V \setminus \{s\}$

***foreach*** $i \in V_2$ ***do***

$p(i) \leftarrow \infty, v(i) \leftarrow Nil$

*end* ***foreach***

***for*** $n = 1, ..., \mid V \mid -1$ ***do***

***if*** $t \in V_1$ ***do***

*stelle den Pfad nach $s$ her, beginnend bei $v(t)$ und* ***return*** *true*

***else***

***foreach*** $(i, j) \in \mathcal{E}$, *mit* $i \in V_1, j \in V_2$ ***do***

***if*** $p(i) > p(j) + k(j, i)$ ***do***

$p(i) = p(j) + k(j, i)$

$v(i) = j$

$i_0 = argmin\{p(i) : i \in V_2\}$

*transfer* $i_0$ *von* $V_2$ *nach* $V_1$

*end* ***if***

*end* ***foreach***

*end* ***if***

*end* ***for***

***return*** *true*

An dieser Stelle möchte ich erwähnen, dass durch das Anwenden des Algorithmus von Dijkstra in jeder Iteration ein kürzester Pfad gefunden wird, nämlich jener zu dem Knoten $i_0$, der von $V_2$ nach $V_1$ transferiert wird. In einem Netzwerk mit $n$ Knoten wird daher ein kürzester Pfad zwischen zwei beliebigen

Knoten nach maximal $n$ Iterationen gefunden. Außerdem wird in jeder Iteration maximal die Anzahl der Pfade im Netzwerk untersucht. Wir erhalten einen Aufwand von $\mathcal{O}(V^2 + \mathcal{E}) = \mathcal{O}(V^2)$. Daher ist der Algorithmus auch für große Netzwerke, was die Laufzeit betrifft, sehr effizient.[21]
Der Algorithmus von Dijkstra würde jedoch in einem Netzwerk mit negativen Pfaden kein korrektes Ergebnis erzielen, da in der Initialisierung festgelegt wird, dass der kürzeste Pfad zum Startknoten den Wert 0 hat.[21][28]

### 7.2.3 Darstellungen praktischer Anwendungen

Im letzten Kapitel meiner Arbeit möchte ich nun die konkreten Source Codes der diskutierten Algorithmen anführen. Des Weiteren werde ich Beispiele von Netzwerken behandeln, in denen das Problem des kürzesten Pfades mit Hilfe der vorgestellten Algorithmen gelöst wurde.
**Bellman – Algorithmus**

```
#include <stdio.h>
typedef struct
{int u, v, k; /*Startknoten, Zielknoten, Kosten*/
}Netzwerk;

#define INF 999999
#define Graph 1024
int n; /*Anzahl der Knoten*/
int num; /*Anzahl der Pfade*/
long d[Graph]; /*Distanz: d[i] ist die kürzeste Distanz vom Startknoten
                zum Knoten i*/
Netzwerk pfad[Graph];

/*Die Funktion "bellman" untersucht, ausgehend von einem Startknoten,
jeden wegführenden Pfad, geht über zum nächsten Knoten und untersucht
einen weiterführenden Pfad. Im nächsten Schritt wird ein anderer Weg
durch das Netzwerk untersucht. Jedem Knoten werden die Distanzkosten
eingeschrieben, welche bei Finden eines kürzeren Pfades mit den neuen
Kosten überschrieben werden.*/
void bellman(int s)
```

```
{int i,j;
/*die Distanz zum Startknoten beträgt Null, die Distanz zu jedem
anderen Knoten wird als unendlich festgelegt*/
d[s]=0;
for (i = 0; i < n; ++i)
    {d[i]=INF;}

/*Überprüfen, ob ein kürzerer Pfad vorhanden ist und evtl.
überschreiben*/
for (i = 0; i < n-1; ++i)
    {for (j = 0; j < num; ++j)
         {if (d[pfad[j].v] > d[pfad[j].u] + pfad[j].k)
          d[pfad[j].v] = d[pfad[j].u] + pfad[j].k;}
    }
/*Überprüfen, ob das Netzwerke einen negativen Zyklus beinhaltet*/
for (i = 0; i < num; ++i)
    {if (d[pfad[i].v] > d[pfad[i].u] + pfad[i].k)
    {printf("Das Netzwerk beinhaltet negative Zyklen.\n");
     break;}
    }
}

/*Die Funktion "printDistanz" gibt das Ergebnis am Ende des
Algorithmus aus.*/
void printDistanz()
{int i;
 printf("\nDie Kosten vom Startknoten\n");
 for (i = 0; i < n; ++i)
     {printf("zum Knoten %d betragen: %ld\t\n", i+1, d[i]);}
 printf("\n");
}

/*Die "main"-Funktion stellt das Hauptprogramm dar. Es werden die
Werte aus der Graphen-Datei eingelesen und die einzelnen Funktionen
des Algorithmus aufgerufen.*/
```

```
int main(int argc, char *arg[])
{int i,j,k;
 FILE *netz = fopen("netzwerk3.txt", "r");
 fscanf(netz, "%d", &n);
 num=0;
 for (i = 0; i < n; ++i)
     {for (j = 0; j < n; ++j)
      {fscanf(netz, "%d", &k);
      if (k != 0) /*um Knoten auszusparen, zu denen kein Weg führt*/
         {pfad[num].u = i;
      pfad[num].v = j;
      pfad[num].k = k;
      ++num;}
      }
     }
fclose(netz);
bellman(0);
printDistanz();
return 0;
}
```

**Beispiel 7.12.** *(Netzwerk 1)*

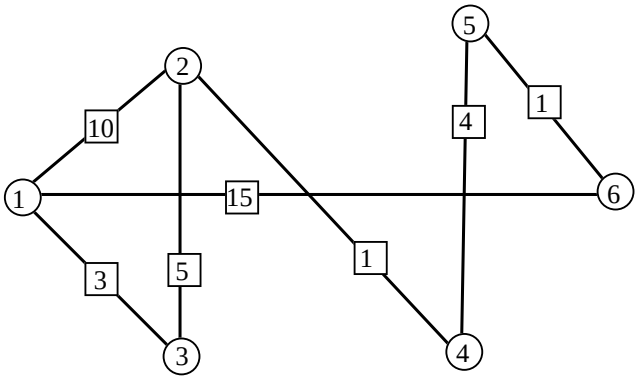



Abbildung 7.2: Netzwerk 1

```
6
0  10  3  0  0  15
10  0  5  1  0  0
3  5  0  0  0  0
0  1  0  0  4  0
0  0  0  4  0  1
15  0  0  0  1  0
```

Abbildung 7.3: Input Netzwerk 1

```
Knoten || Kosten vom Startknoten
 2     ||      8
 3     ||      3
 4     ||      9
 5     ||      13
 6     ||      14
```

Abbildung 7.4: Output Netzwerk 1

**Beispiel 7.13.** *(Netzwerk 2)*

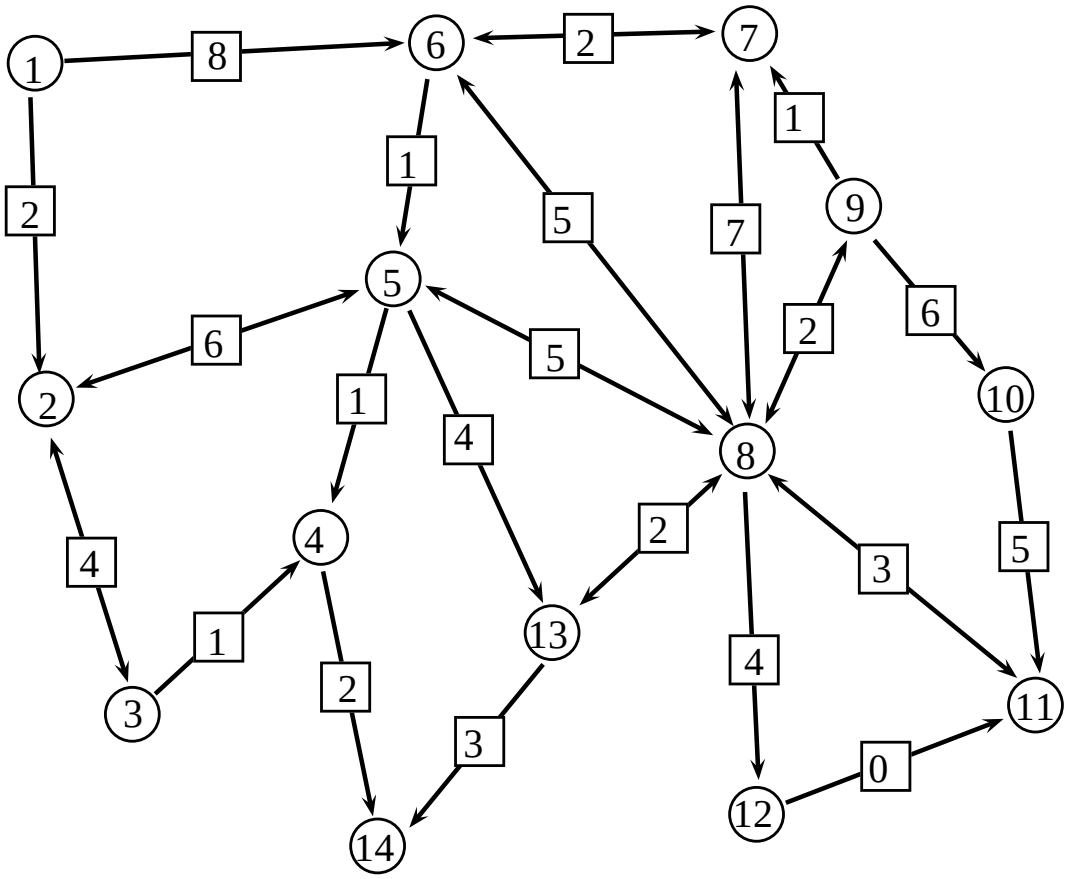



Abbildung 7.5: Netzwerk 2

14

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 2 | 0 | 0 | 0 | 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 4 | 0 | 6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 4 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 |
| 0 | 6 | 0 | 1 | 0 | 0 | 0 | 5 | 0 | 0 | 0 | 0 | 4 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 2 | 5 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 2 | 0 | 7 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 5 | 5 | 7 | 0 | 2 | 0 | 3 | 4 | 2 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 0 | 6 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 5 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 3 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Abbildung 7.6: Input Netzwerk 2

```
Knoten || Kosten vom Startknoten
 2     ||       2
 3     ||       6
 4     ||       7
 5     ||       8
 6     ||       8
 7     ||       10
 8     ||       13
 9     ||       15
 10    ||       21
 11    ||       16
 12    ||       17
 13    ||       12
 14    ||       9
```

Abbildung 7.7: Output Netzwerk 2

**Beispiel 7.14.** *(Netzwerk 3)*

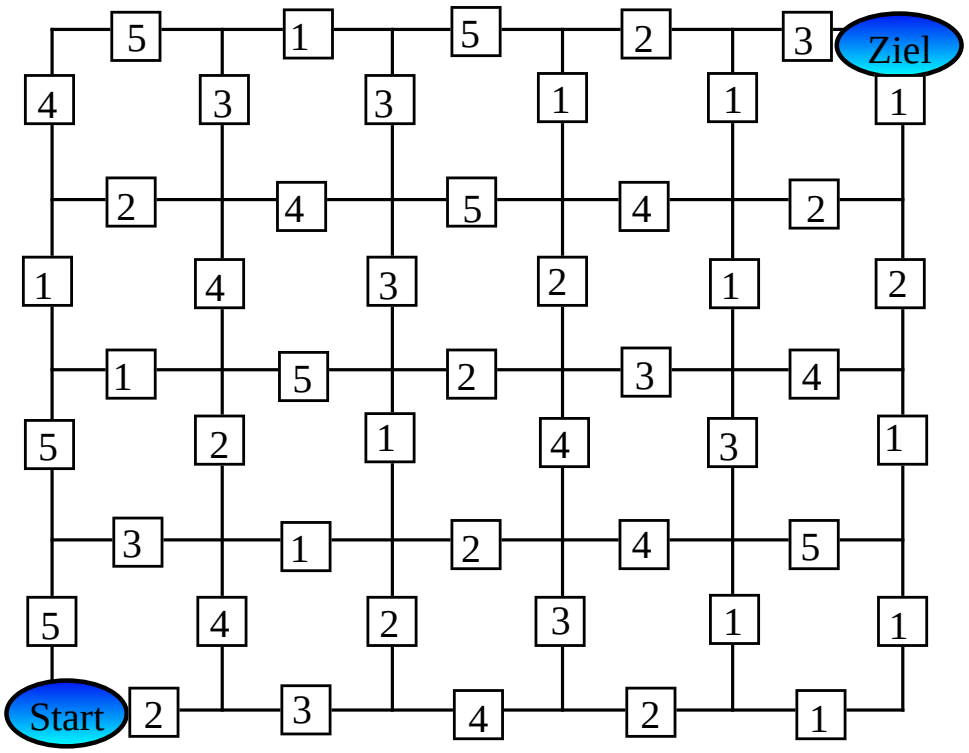



Abbildung 7.8: Netzwerk 3

| Knoten | Kosten vom Startknoten | | |
|---|---|---|---|
| 2 | 2 | 16 | 10 |
| 3 | 5 | 17 | 13 |
| 4 | 9 | 18 | 14 |
| 5 | 11 | 19 | 10 |
| 6 | 12 | 20 | 12 |
| 7 | 5 | 21 | 11 |
| 8 | 6 | 22 | 12 |
| 9 | 7 | 23 | 14 |
| 10 | 9 | 24 | 16 |
| 11 | 12 | 25 | 14 |
| 12 | 13 | 26 | 15 |
| 13 | 9 | 27 | 14 |
| 14 | 8 | 28 | 13 |
| 15 | 8 | 29 | 15 |
| | | 30 | 17 |

Abbildung 7.9: Output Netzwerk 3

30

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 2 | 0 | 0 | 0 | 0 | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 3 | 0 | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 3 | 0 | 4 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 4 | 0 | 2 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 2 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 4 | 0 | 0 | 0 | 0 | 3 | 0 | 1 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 2 | 0 | 0 | 0 | 0 | 1 | 0 | 2 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 2 | 0 | 4 | 0 | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 4 | 0 | 5 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 1 | 0 | 5 | 0 | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 5 | 0 | 2 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 2 | 0 | 3 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 3 | 0 | 4 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 2 | 0 | 4 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 4 | 0 | 5 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 5 | 0 | 4 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 4 | 0 | 2 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 5 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 5 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 1 | 0 | 5 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 5 | 0 | 2 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 2 | 0 | 3 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Abbildung 7.10: Input Netzwerk 3

**Dijkstra – Algorithmus**

```
#include <stdio.h>
typedef struct
{int u, v, k; /*Startknoten, Zielknoten, Kosten*/
}Netzwerk;

#define INF 999999
#define Graph 1024
#define MAX(a, b) ((a > b) ? (a) : (b))

int m; /*Anzahl der Pfade mit Distanz ungleich Null*/
int n; /*Anzahl der Knoten*/
int num; /*Anzahl der Pfade*/
long d[Graph]; /*d[i] ist die Distanz vom Startknoten zum Knoten i*/
long dist[Graph][Graph]; /*dist[i][j] gibt die Distanz von Knoten i zu
Knoten j an*/
Netzwerk pfad[Graph];

/*Die Funktion "Dijkstra" untersucht von einem Knoten ausgehend alle
umliegenden Pfade und benennt die Knoten mit den Kosten. Sodann wird
der eben untersuchte Knoten als "besucht" markiert. Im nächsten Schritt
werden vom nächsten Knoten, der die kürzeste Distanz vom eben besuchten
Knoten hat, alle umliegenden Pfade untersucht und überprüft, ob ein
kürzerer Weg vorhanden ist. Ist das der Fall, so wird dem Knoten die
kürzere Distanz zugeschrieben. Iterativ wählt der Algorithmus den
nächsten Knoten, zu dem die kürzeste Distanz besteht so lange, bis alle
Knoten als "besucht" markiert wurden.*/
void Dijkstra(int s)
{
int i, j, knoten;
int besucht[Graph]; /*besucht[i] gibt mit den Werten 0 oder 1 an, ob
der Knoten i bereits überprüft wurde*/

/*die Distanz zum Startknoten beträgt Null, die Distanz zu jedem
anderen Knoten wird als unendlich festgelegt*/
```

```
d[s] = 0;
for (i = 1; i <= n; ++i)
    {d[i] = INF;
     besucht[i] = 0;}
/*falls der Knoten noch nicht besucht wurde, wird die Distanz zu jedem
umliegenden Knoten überprüft und anschließend als "besucht" markiert*/
for (i = 1; i <= n; ++i)
    {knoten = -1;
    for (j = 1; j <= n; ++j)
    {if (!besucht[j] && ((knoten == -1) || (d[j] < d[knoten])))
             knoten = j;}

    for (j = 1; j <= n; ++j)
    {if (dist[knoten][j] != 0)
         if (d[knoten] + dist[knoten][j] < d[j])
     d[j] = d[knoten] + dist[knoten][j];
    }
besucht[knoten] = 1;
    }
/*überpruefen, ob das Netzwerke negative Pfade beinhaltet*/
for (i = 1; i <= n; ++i)
    {for (j = 1; j <= n; ++j)
 {if (dist[i][j] < 0)
     {printf("\nDas Netzwerk beinhaltet negative Pfade und liefert
             daher möglicherweise ein falsches Ergebnis.\n");
      break;}
 }
     }
}
/*Die Funktion "printDistanz" gibt am Ende des Algorithmus
das Ergebnis aus.*/
void printDistanz()
{
int i;
printf("\n");
```

```
printf("Die Kosten vom Startknoten\n");
for (i = 2; i <= n; ++i)
    {printf("zum Knoten %d betragen: %ld\t\n", i, d[i]);}
printf("\n");
}

/*Die "main"-Funktion stellt das Hauptprogramm dar. Es werden die
Werte aus der Graphen-Datei eingelesen und die einzelnen Funktionen
des Algorithmus aufgerufen.*/
int main(int argc, char *arg[])
{
int i, j, u, v, k;

FILE *netz = fopen("test", "r");
fscanf(netz, "%d", &m);
for (j = 0; j < m; ++j)
    for (i = 0; i < m; ++i)
 dist[j][i] = 0;
for (j = 0; j < m; ++j)
    {fscanf(netz, "%d%d%d", &u,&v,&k);
     dist[u][v] = k;
     n = MAX(u, MAX(v, n));
    }
fclose(netz);

Dijkstra(1);
printDistanz();
return 0;
}
```

**Beispiel 7.15.** *(Netzwerk 4)*

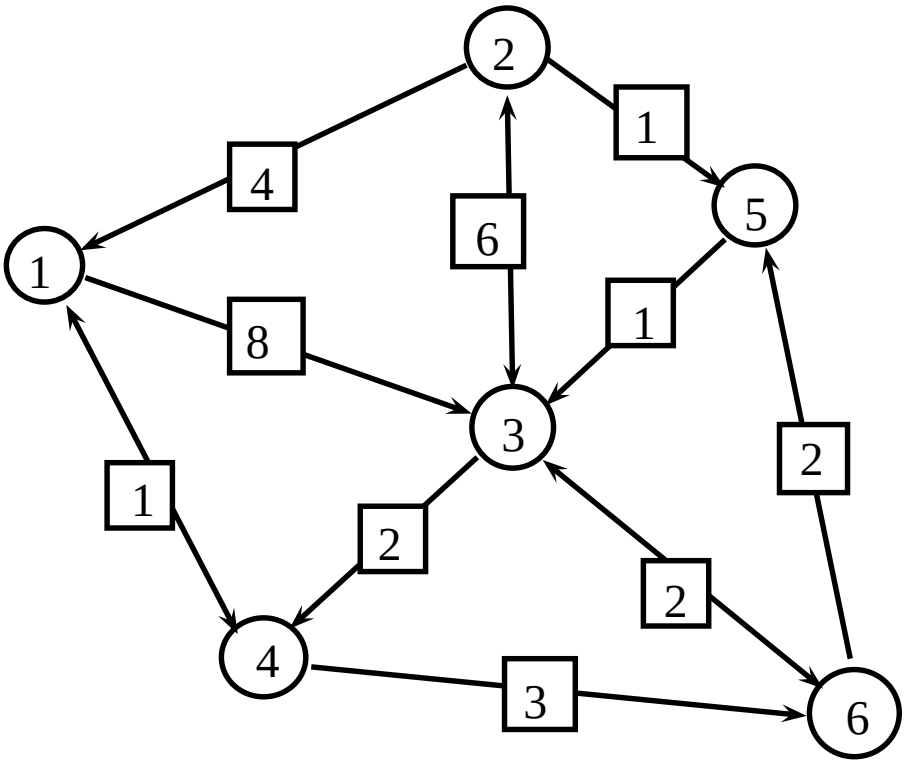



Abbildung 7.11: Netzwerk 4

```
14
1  3  8
1  4  1
2  1  4
2  3  6
2  5  1
3  2  6
3  4  2
3  6  2
4  1  1
4  6  3
5  3  1
6  3  2
6  5  2
```

Abbildung 7.12: Input Netzwerk 4

```
Knoten || Kosten vom Startknoten
 2     ||      12
 3     ||      6
 4     ||      1
 5     ||      6
 6     ||      4
```

Abbildung 7.13: Output Netzwerk 4

**Beispiel 7.16.** *(Netzwerk 5)*

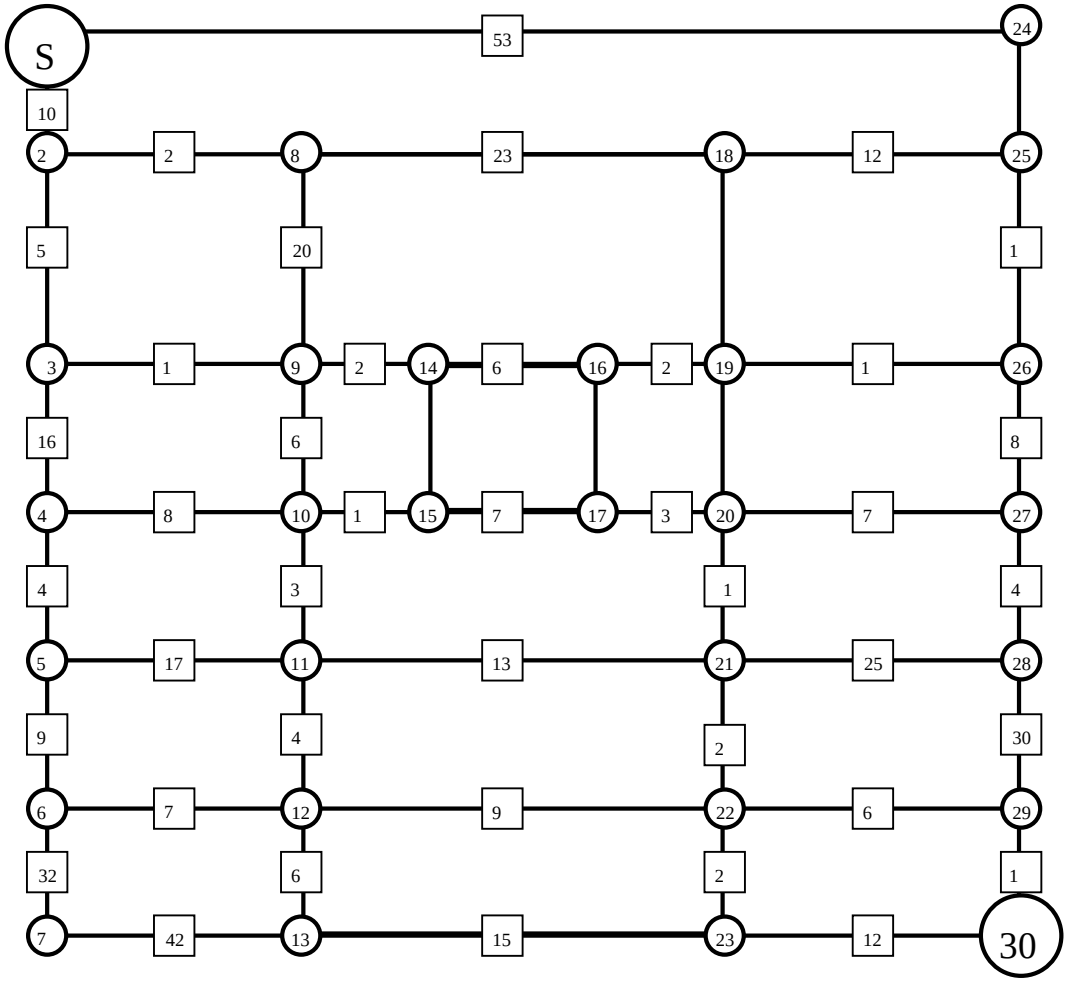



Abbildung 7.14: Netzwerk 5

```
83
1  2  10
1  24 53
2  1  10
2  3  5
2  8  2
3  2  5
3  4  16
3  9  1
4  3  16
4  5  4
4  10 8
5  4  4
5  6  9
5  11 17
6  5  9
6  7  32
6  12 7
7  6  32
7  13 42
8  2  2
8  9  20
8  18 23
9  3  1
9  8  20
9  10 6
9  14 2
10 4  8
10 9  6
10 11 3
10 15 1
11 5  17
11 10 3
11 12 4
11 21 13
12 6  7
12 11 4
12 13 6
12 22 9
13 7  42
13 12 6
13 23 15
14 9  2
14 16 6
15 10 1
15 17 7
16 14 6
16 19 2
17 15 7
17 20 3
18 8  23
18 25 12
19 16 2
19 26 1
20 17 3
20 21 1
20 27 7
21 11 13
21 20 1
21 22 2
21 28 25
22 12 9
22 21 2
22 23 2
22 29 6
23 13 15
23 22 2
23 30 12
24 1  53
25 18 12
25 26 1
26 19 1
26 27 8
27 20 7
27 26 8
27 28 4
28 21 25
28 27 4
28 29 30
29 22 6
29 28 30
29 30 1
30 23 12
30 29 1
```

Abbildung 7.15: Input Netzwerk 5

| Knoten | Kosten vom Startknoten | | |
|---|---|---|---|
| 2 | 10 | 16 | 24 |
| 3 | 15 | 17 | 30 |
| 4 | 30 | 18 | 35 |
| 5 | 34 | 19 | 26 |
| 6 | 36 | 20 | 33 |
| 7 | 68 | 21 | 34 |
| 8 | 12 | 22 | 36 |
| 9 | 16 | 23 | 38 |
| 10 | 22 | 24 | 53 |
| 11 | 25 | 25 | 47 |
| 12 | 29 | 26 | 27 |
| 13 | 35 | 27 | 35 |
| 14 | 18 | 28 | 39 |
| 15 | 23 | 29 | 42 |
| | | 30 | 43 |

Abbildung 7.16: Output Netzwerk 5

# Literaturverzeichnis


[1] R. Bellman. *Dynamic Programming.* Princeton University Press, 1957.

[2] R. Bellman. *On a routing problem.* Quarterly Applied Mathematics, 1: 425-447, 1958.

[3] T. Bradtke. *Grundlagen in Operations Research für Ökonomen*, München, Wien, Oldenbourg, 2003.

[4] B. V. Cherkassky, A. V. Goldberg und T. Radzik. *Shortest path algorithms: Theory and experimental evaluation.* Mathematical Programming, 73: 129-196, 1996.

[5] E. J. Chester. *Fast methods for computing all-to-all geodesic paths for the Eikonal equation.* University of California, Berkeley, 2007.

[6] P. Dayan P und C. Watkins. *Reinforcement learning.* Encyclopedia of Cognitive Science London, England: MacMillan Press, 2001.

[7] E. V. Denardo und B. L. Fox. *Shortest route methods: reaching, pruning and buckects.* Operations Research, 27: 161-186, 1979.

[8] E. Dijkstra. *A note on two problems in connection to graphs.* Numerical Mathematics, 1: 269-271, 395-412, 1959.

[9] L. R. Ford, Jr. und D. R. Fulkerson. *Network Flow and System of Representatives*, Can. J. Math., 10 Vol.1: 78-84, 1958.

[10] L. R. Ford, Jr. und D. R. Fulkerson. *A simple algorithm for finding maximal network flows and an application to the Hitchcock Problem*, Can. J. Math., 9: 210-218, 1957.

[11] G. Gallo und S. Pallotino. *Shortest path methods: a unifying approach.* Mathematical Programming Study, 26: 38-64, 1986.

[12] T. Jung. *Reinforcement Lernen mit Regularisierungsnetzen.* Dissertation, Johannes Gutenberg-Universität Mainz, 2007.

[13] E. Martins, M. Pascoal, D. Rasteiro und J. L. Santos. *The Optimal Path Problem.* Universität von Coimbra, Portugal, 1998.

[14] M. L. Minsky. *Theory of Neural - Analog Reinforcement Systems and its Application to the Brain - Model Problem.* PhD thesis, Princeton University, 1954.

[15] E. F. Moore. *The shortest path through a maze.* Proceedings of the International Symposium on the Theory of Switching, Harvard University Press: 285-292, 1959.

[16] U. Pape. *Implementation and efficiency of moore algorithms for the shortest roote problem.* Mathematical Programming, 7: 212-222, 1974.

[17] P. I. Pavlov. *Conditional Reflexes.* Oxford, London, 1927.

[18] M. Puterman. *Markov Decision Problems.* Wiley, New York, 1994.

[19] H. Robbins und S. Monro. *A stochastic approximation method.* Annals of Mathematical Statistics, 1951.

[20] B. Runzheimer, T. Cleff und W. Schäfer. *Operations Research 1. Lineare Planungsrechnung und Netzplantechnik.* 7.Auflage, Gabler, Wiesbaden, 1999.

[21] G. Sierksma und D. Gosh. *Networks in Action. Text and Computer Exercises in Network Optimization.* In International Series in Operations Research Management Science Vol.140, Stanford University, CA, USA, 2009.

[22] R. Sutton. *Learning to predict by the methods of temporal difference.* Machine Learn- ing 3, 1988.

[23] R. Sutton und A. Barto. *Reinforcement Learning: An Introduction.* MIT Press, 1998.

[24] R. Sutton und A. Barto. *A temporal-difference model of classical conditioning.* Pro- ceedings of the Ninth Annual Conference of the Cognitive Science Society, Seattle, 1987.

[25] R. Sutton, A. Barto und C.J.C.H. Watkins. (1990) *Learning and sequential decision making.* M Gabriel and J Moore (Editors). In Learning and Computational Neuroscience: Foundations of Adaptive Networks. Cambridge, MA: MIT Press, Bradford Books, 1990.

[26] G. Tesauro. *Temporal Difference Learning and TD - Gammon.* In *Communications of the ACM*, Vol.38, 1995.

[27] E. L. Thorndike. *Animal Intelligence.* Hafner, Darien, Connecticut, 1911.

[28] L. T. Wang, Y. W. Chang und K. T. Cheng. *Electronic design automation: synthesis, verification, and test.* Morgan Kaufmann, Massachusetts, 2009.

[29] C.J.C.H. Watkins, P. Dayan. *Q–Learning. Machine Learning.* 8: 279-292, 1992.

[30] C.J.C.H. Watkins. *Learning from Delayed Rewards.* PhD Thesis, University of Cambridge, Cambridge, 1989.

Judith Fechter



# Curriculum Vitae

## Ausbildung

| | |
|---|---|
| Studienjahr 2010/11 | Auslandsaufenthalt an der "University of Edinburgh" (GB) |
| Seit 2006 | Studium des Lehramts für Mathematik und Psychologie und Philosophie an der Universität Wien |
| Seit 2005 | Studium der Technischen Mathematik mit der Spezialisierung Wirtschaftsmathematik |
| 1997- 2005 | Erzbischöfliches Realgymnasium Hollabrunn |
| 1993- 1997 | Volksschule Nappersdorf in Niederösterreich |

## Berufserfahrung

| | |
|---|---|
| Seit 2011 | TU WIEN<br>Lektorin im Fach "Statistik und Wahrscheinlichkeitstheorie" für InformatikerInnen |
| Seit 2011 | MATURASCHULE LERNEN 8<br>Abhalten von Vorbereitungskursen für die Latinum - Prüfung für Studierende der österreichischen Universitäten |

| | |
|---|---|
| Seit 2009 | LERNQUADRAT WIEN<br>Nachhilfetätigkeit in den Fächern Latein und Mathematik<br>(Gruppen- und Einzeltraining) |
| Seit Juli 2008 | FERIENHORT am WOLFGANGSEE<br>Jugendbetreuung im Sommercamp "Ferienhort am Wolfgangsee" mit Teamleiterfunktion |
| 2006-2009 | SCHÜLERHILFE WIEN<br>Nachhilfetätigkeit in den Fächern Latein und Mathematik<br>(Gruppen- und Einzeltraining) |
| Juli 2006 | AUSTRIAN AIRLINES<br>im Bereich Buchhaltung |
| August<br>2005 | IMMOBILIENVERWALTUNG KRAMAS<br>im Bereich Buchhaltung |
| August<br>2004 | EVN<br>im Bereich Vermessungswesen |

## Sprachkenntnisse

| | |
|---|---|
| Deutsch | Muttersprache |
| Englisch | fließend in Wort und Schrift |
| Latein | |

## EDV - Kenntnisse

| | |
|---|---|
| MS Office | sehr gute Kenntnisse |
| LaTex | sehr gute Kenntnisse |
| Matlab | gute Kenntnisse |
| C | gute Kenntnisse |
| R | gute Kenntnisse |

## Hobbies und sonstige Kompetenzen

| | |
|---|---|
| Hobbies | Familie, Freunde, Laufen, Skifahren, Schwimmen, Segeln |
| Sonstiges | B (PKW), Segelschein ÖSV-A (Binnenrevier),<br>Retterschein (ÖWR) |

# MERKBLATT FÜR DEN VERFASSER VON DIPLOMARBEITEN

Der unterzeichnete Verfasser der Diplomarbeit (Titel): .....
.....
.....
nimmt zur Kenntnis:

1) Die Diplomarbeit wird in der Universitätsbibliothek, Nationalbibliothek und dem für das Diplomarbeitsfach zuständige Universitäts-Institut aufgestellt und allgemein zugänglich gemacht.

2) Von erschienenen (d.h. approbierten, durch Druck oder ein anderes Vervielfältigungsverfahren veröffentlichten) Diplomarbeiten dürfen Kopien für den eigenen Gebrauch der Bibliothek in beliebiger Zahl angefertigt werden. Für den Gebrauch anderer Personen dürfen Kopien von der Bibliothek bzw. ihren Benützern nur mit Genehmigung des Autors hergestellt werden.

3) Bei der Benützung der Diplomarbeit im Lesesaal oder durch Entlehnung wird seitens der Bibliothek eine Erklärung des Inhalts verlangt, dass bei jeder Verwendung der Diplomarbeit oder einzelner Teile daraus, der Verfasser und die Fakultät, die die Diplomarbeit approbiert hat, anzugeben sind.

Wien,..... .....
Unterschrift