Einführung in die Methoden der Künstlichen Intelligenz

Sommersemester 2018

Prof. Dr. Manfred Schmidt-Schauß
und
PD Dr. David Sabel

Institut für Informatik
Fachbereich Informatik und Mathematik
Goethe-Universität Frankfurt am Main
Postfach 11 19 32
D-60054 Frankfurt am Main
Email: schauss@ki.informatik.uni-frankfurt.de Email:
sabel@ki.informatik.uni-frankfurt.de

Stand: 13. Juni 2018

Inhaltsverzeichnis

1 Einleitung

1.1 Themen und Literatur

Für die Veranstaltung sind die folgenden Inhalte geplant (kann sich noch leicht ändern).

- Einführung: Ziele der künstliche Intelligenz, intelligente Agenten.
- Suchverfahren: Uninformierte und informierte Suche, Suche in Spielbäumen.
- Evolutionäre (oder auch genetische) Algorithmen.
- Maschinelles Lernen
- Planen

In der Anschlussvorlesung KILOG werden dann die Themen behandelt

- Wissensrepräsentation und Deduktion in der Aussagenlogik.
- Prädikatenlogik
- Qualitatives zeitliches Schließen am Beispiel von Allens Intervalllogik
- Konzeptbeschreibungsschreibungssprachen

Wir werden einige Algorithmen erörtern, wobei wir neben (imperativem) Pseudo-Code auch desöfteren eine entsprechende Implementierung in der funktionalen Programmiersprache Haskell angeben. Die Haskellprogramme dienen eher zum Verständnis und sind daher selbst kein Prüfungsstoff.

1.1.1 Wesentliche Lehrbücher für diese Veranstaltung

D. Poole, and A. Mackworth, *Artificial Intelligence: Foundations of Computational Agents*. Cambridge University Press, 2010. auch online unter http://artint.info/ frei verfügbar

S.J. Russel and P. Norvig. *Artificial Intelligence: A Modern Approach.* Third Edition, Prentice Hall, 2010.

W. Ertel *Grundkurs Künstliche Intelligenz: Eine praxisorientierte Einführung*. Vieweg-Teubner Verlag, 2009. Aus dem Netz der Universität als E-Book kostenlos verfügbar unter http://www.springerlink.com/content/t13128/

D. Poole, A. Mackworth, and R. Goebel. *Computational Intelligence: A Logical Approach*. Oxford University Press, 1998.

Franz Baader, Deborah McGuiness, Daniele Nardi, and Peter Patel-Schneider. *The description logic handbook*. Cambridge university press, 2002. Aus dem Netz der Universität als E-Book kostenlos verfügbar unter http://site.ebrary.com/lib/frankfurtm/docDetail.action?docID=10069975

Christoph Beierle and Gabriele Kern-Isberner. *Methoden wissensbasierter Systeme*. 4. Auflage, Vieweg, 2008. Aus dem Netz der Universität als E-Book kostenlos verfügbar unter http://www.springerlink.com/content/h67586/

P. Winston. *Artificial Intelligence*. Addison Wesley, 1992.

McDermott Charniak. *Introduction to Artificial Intelligence*. Addison Wesley, 1985.

Genesereth and Nilsson. *Logical foundations of artificial intelligence*. Morgan Kaufmann, 1988.

M. Ginsberg. *Essentials of artificial intelligence*. Morgan Kaufmann, 1993.

Wolfgang Bibel, Steffen Hölldobler, and Torsten Schaub. *Wissensrepräsentation und Inferenz. Eine grundlegende Einführung*. Vieweg, 1993.

Joachim Herzberg. *Planen: Einführung in die Planerstellungsmethoden der Künstlichen Intelligenz*. Reihe Informatik. BI Wissenschaftsverlag, 1989.

Michael M. Richter. *Prinzipien der Künstlichen intelligenz*. Teubner Verlag, 1992.

1.1.2 Weitere Lehr- und Handbücher

A. Barr, P.R. Cohen, and E.A. Feigenbaum. *The Handbook of Artificial Intelligence*. Addison Wesley, 1981-1989. 4 Bände.

E. Davies. *Representation of Commonsense Knowledge*. Morgan Kaufmann, 1990.

H.L. Dreyfus. *What Computers Can't Do*. Harper Colophon Books, 1979.

Günter Görz, editor. *Einführung in die künstliche Intelligenz*. Addison Wesley, 1993.

Peter Jackson. *Introduction to expert systems, 3rd ed.* Addison-Wesley, 1999.

George F. Luger and William A. Stubblefield . *Artificial intelligence: structures and strategies for complex problem solving. 3rd ed*. Addison-Wesley, 1998.

Zbigniew Michalewicz. *Genetic algorithms + data structures = evolution programs*. Springer-Verlag, 1994.

Todd C. Moody. *Philosophy and Artificial Intelligence*. Prentice Hall, 1993.

Frank Puppe. *Problemlösungsmethoden in Expertensystemen*. Springer-Verlag, 1990.

Frank Puppe. *Einführung in Expertensysteme*. Springer-Verlag, 1991.

Peter Schefe. *Künstliche Intelligenz, Überblick und Grundlagen*. Reihe Informatik. BI Wissenschaftsverlag, 1991.

S. Shapiro. *Encyclopedia of Artificial Intelligence*. John Wiley, 1989-1992. 2 Bände.

Herbert Stoyan. *Programmiermethoden der Künstlichen Intelligenz*. Springer-Verlag, 1988.

J. Retti u.a. *Artificial Intelligence – Eine Einführung*. Teubner Verlag, 1986.

Vidysagar. *A theory of learning and generalization*. Springer-Verlag.

Ingo Wegener, editor. *Highlights aus der Informatik*. Springer, 1996.

Reinhard Wilhelm, editor. *Informatics – 10 years back – 10 years ahead*. Lecture Notes in Computer Science. Springer-Verlag, 2000.

1.2 Was ist Künstliche Intelligenz?

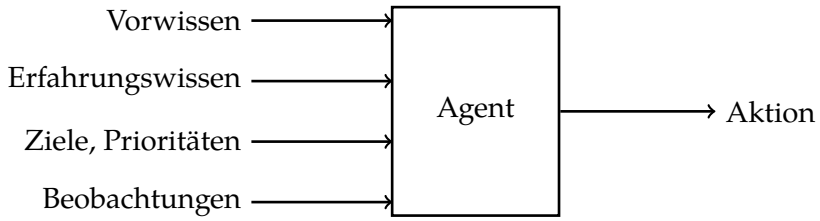
Ziel der Künstlichen Intelligenz ist die Herstellung eines *intelligenten Agenten*. Dies kann man auch als die Herstellung eines möglichst guten (autonomen, lernenden, intelligenten, automatischen) Informationssystems ansehen. Einige Beispiele für solche Systeme sind:

- Taschenrechner, die analog zum Menschen Rechenaufgaben lösen können (dabei aber anders Rechnen als der Mensch).
- Schachspielende Computer wie z.B. Deep Blue, Deep Thought und Deep Fritz
- Go-spielender Computer AlphaGo.
- Sprachübersetzer wie z.B. GoogleTranslate, Babelfish, etc.
- wissensbasierte Systeme in vielen Varianten
- Texterkennungssysteme in Kombinationen mit Hintergrundwissen wie z.B. das Watson-Programm[1], welches bei der amerikanischen TV-Show „Jeopardy“ gegen frühere Champions antrat und gewann.

[1] www.ibm.com/de/watson

- Roboter, z.B. Haushaltsroboter wie Staubsaugerroboter, Industrieroboter, etc.
- intelligente Informationssysteme

Allgemein ist ein intelligenter Agent, der im Kern einen Computer hat und mit der physikalischen Umwelt agiert, ein *Roboter*. Sind die Umgebungen nicht-physikalisch, so wird dies meist durch spezielle Namen gekennzeichnet: Ein Software-Roboter (softbot) hat eine Wissensbasis und gibt Antworten und Ratschläge, aber agiert nicht mit der physikalischen Umwelt. Web-Roboter (webbot) interagieren mit dem WWW, z.B. zum Durchsuchen und Erstellen von Suchdatenbanken im WWW. Ein Chat-Roboter (chatbot) agiert in einem Chat. Ein Agent hat als Eingaben und Zwischenzustände zum einen Vorwissen über die Umgebung (z.B. eine Karte), aber auch erlerntes Wissen, Testfälle etc. Die vom Agent zu erreichenden Ziele sind üblicherweise mit Prioritäten und Wichtigkeiten versehen. Allgemein kann ein Agent Beobachtungen über die Umgebung und auch Beobachtungen über sich selbst machen (und daraus Schlüsse ziehen, beispielsweise Lernen). Aufgrund dieser „Eingaben“ berechnet der Agent als als Ausgabe seine nächste *Aktion*.



Bevor wir auf die genauere Modellierung von Agenten eingehen, wollen wir erörtern, was man unter Künstlicher Intelligenz versteht, d.h. wann ein System (Computer) über künstliche Intelligenz verfügt und welche Eigenschaften eine Methode als KI-Methode auszeichnet.

Allerdings gibt es keine eindeutige allgemeine Definition der künstlichen Intelligenz. Wir werden daher verschiedene Ansätze und Kennzeichen für künstliche Intelligenz erläutern. Gemäß (Russell & Norvig, 2010) kann man die verschiedenen Ansätze entsprechend der zwei Dimensionen:

- menschliches Verhalten gegenüber rationalem Verhalten
- Denken gegenüber Handeln

in vier Klassen von Ansätzen unterteilen: Menschliches Denken, rationales Denken, menschliches Handeln und rationales Handeln. Während „menschlich“ bedeutet, dass das Ziel ist, den Menschen nachzuahmen, meint „rational“ ein vernünftiges Vorgehen, das optimal (bezüglich einer Kostenabschätzung o.ä.) ist. Wir erläutern im Folgenden genauer, was KI entsprechend dieser Klassen meint und als Ziel hat.

1.2.1 Menschliches Handeln

Verwendet man menschliches Handeln als Ziel künstlicher Intelligenz, so steht im Vordergrund, Systeme oder auch Roboter zu erschaffen, die analog zu Menschen handeln. Ein Ziel dabei ist es, Methoden und Verfahren zu entwickeln, die Computer dazu bringen, Dinge zu tun, die momentan nur der Mensch kann, oder in denen der Mensch noch den Computern überlegen ist. Eine Methode, um „nachzuweisen“, ob der Computer entsprechend intelligent handelt, besteht darin, seine Handlungen mit den Handlungen des Menschen zu vergleichen (wie z.B. der Turing-Test, siehe unten).

1.2.2 Menschliches Denken

Dieser Ansatz verfolgt das Ziel, dass ein Computer „wie ein Mensch denkt“. Dabei muss zunächst erforscht werden, wie der Mensch selbst denkt, bevor entsprechende Modelle auf Computer übertragen werden können. Dieses Erforschen kann beispielsweise durch psychologische Experimente oder Hirntomografie erfolgen. Ist eine Theorie aufgestellt, so kann diese in ein System umgesetzt werden. Wenn die Ein- und Ausgaben dieses Systems dem menschlichen Verhalten entsprechen, kann man davon ausgehen, dass dieses System auch im Mensch eingesetzt werden könnte und daher dem menschlichen Denken entspricht. Dieser Ansatz ist jedoch nicht alleinig in der Informatik angesiedelt, sondern entspricht eher dem interdisziplinären Gebiet der Kognitionswissenschaft, die informatische Modelle mit experimentellen psychologischen Versuchen kombiniert, um menschliches Verhalten durch Modelle zu erfassen und nachzuahmen. Beispiele für dieses Gebiet sind Fragen danach, wie der Mensch etwas erkennt (z.B. wie erkennen wir Gesichter), oder Verständnis des Spracherwerbs des Menschen. Ein System ist *kognitiv adäquat*, wenn es strukturell und methodisch wie ein Mensch arbeitet und entsprechende Leistungen erzielt. Z.B. ist ein Taschenrechner nicht kognitiv adäquat, da er anders addiert als der Mensch, die Methodik stimmt also nicht überein. Daher ist dieser Ansatz der *kognitiven Simulation* eher vom Gebiet der *künstlichen Intelligenz* abzugrenzen (also als verschieden anzusehen), da er sehr auf exakte Nachahmung des menschlichen Denkens zielt. Wir werden daher diesen Ansatz nicht weiter verfolgen, sondern ihn der Kognitionswissenschaft überlassen.

1.2.3 Rationales Denken

Rationales Denken versucht Denken durch Axiome und *korrekte* Schlussregeln zu formalisieren. Meistens wird hierzu eine Logik verwendet. D.h. der logische Ansatz (ein großes Teilgebiet der Künstlichen Intelligenz) ist es, aufgrund einer Logik ein *Deduktionssystem* zu implementieren, das sich intelligent verhält. Eine Hürde bei diesem Ansatz ist es, das entsprechende Problem bzw. das Wissen in einer Logik zu formalisieren.

1.2.4 Rationales Handeln

Dies ist der Agenten-Ansatz. Ein Agent ist ein System, dass agiert, d.h. auf seine Umgebung (die Eingaben) eine Reaktion (die Ausgaben) durchführt. Dabei erwartet man im Allgemeinen, dass der Agent autonom operiert, d.h. auf die Umgebung für längere Zeiträume reagiert, sich an Änderungen anpassen kann und ein Ziel verfolgt (und dieses eventuell den Gegebenheiten anpasst). Ein rationaler Agent ist ein Agent, der sein Ergebnis maximiert, d.h. stets das bestmögliche Ergebnis erzielt. Dieser Ansatz ist sehr allgemein und kann auch teilweise die anderen Ansätze miteinbeziehen. So ist es durchaus denkbar, dass ein rationaler Agent eine Logik mitsamt Schlussregeln verwendet, um rationale Entscheidungen zu treffen. Die Wirkung des rationalen Agenten kann menschlich sein, wenn dies eben rational ist. Es ist jedoch keine Anforderung an den rationalen Agent, menschliches Verhalten abzubilden. Dies ist meist ein Vorteil, da man sich eben nicht an einem gegebenen Modell (dem Mensch) orientiert, sondern auch davon unabhängige Methoden und Verfahren verwenden kann. Ein weiterer Vorteil ist, dass Rationalität klar mathematisch definiert werden kann und man daher philosophische Fragen, was intelligentes Verhalten oder Denken ist, quasi umgehen kann.

1.2.5 Ausrichtungen und Ausrichtung der Vorlesung

Die Kognitionswissenschaft versucht das intelligente Handeln und Denken des Menschen zu analysieren, zu modellieren und durch informatische Systeme nachzuahmen. Dafür werden insbesondere das Sprachverstehen, Bildverstehen (und auch Videoverstehen) untersucht und durch Lernverfahren werden entsprechende Systeme trainiert und verbessert. Die *ingenieurmäßig ausgerichtete KI* hingegen stellt Methoden, Techniken und Werkzeuge zum Lösen komplexer Anwendungsprobleme bereit, die teilweise kognitive Fähigkeiten erfordern. Als Beispiele sind hier Deduktionstechniken, KI-Programmiersprachen, neuronale Netze als Mustererkenner, Lernalgorithmen, wissensbasierte Systeme, Expertensysteme zu nennen.

In der Vorlesung werden wir die ingenieurmäßige Ausrichtung verfolgen und uns mit direkt programmierbaren KI-Methoden beschäftigen, die insbesondere Logik in verschiedenen Varianten verwenden.

Einige *Gebiete der Künstlichen Intelligenz* sind

- Programmierung strategischer Spiele (Schach, ...)
- Automatisches/Interaktives Problemlösen und Beweisen
- Natürlichsprachliche Systeme (Computerlinguistik)
- Bildverarbeitung
- Robotik

- (medizinische) Expertensysteme
- Maschinelles Lernen

1.3 Philosophische Aspekte

Wir betrachten kurz einige philosophische Aspekte der künstlichen Intelligenz, indem wir zunächst die philosophischen Richtungen Materialismus, Behaviorismus und Funktionalismus erörtern.

Materialismus Dem Materialismus liegt der Gedanke zu grunde, dass es nichts gibt außer Materie. Diese bewirkt auch den Geist bzw. die Gedanken eines Menschen. Die Implikation daraus ist, dass alles was einen Menschen ausmacht prinzipiell mithilfe naturwissenschaftlicher Methoden (Chemie, Physik, Biologie, ...) analysierbar ist. Als Konsequenz (insbesondere für die künstliche Intelligenz) ergibt sich, dass prinzipiell auch alles konstruierbar ist, insbesondere ist bei diesem Ansatz im Prinzip auch der denkende, intelligente Mensch konstruierbar.

Behaviorismus Der Behaviorismus geht davon aus, dass nur das Beobachtbare Gegenstand der Wissenschaft ist. D.h. nur verifizierbare Fragen und Probleme werden als sinnvoll angesehen. Glauben, Ideen, Wissen sind nicht direkt, sondern nur indirekt beobachtbar. Bewußtsein, Ideen, Glaube, Furcht, usw. sind Umschreibungen für bestimmte Verhaltensmuster. Ein Problem dieser Sichtweise ist, dass man zum Verifizieren evtl. unendlich viele Beobachtungen braucht. Zum Falsifizieren kann hingegen eine genügen. Äquivalenz von Systemen (z.B. Mensch und Roboter) wird angenommen, wenn sie gleiches Ein-/Ausgabe-Verhalten zeigen.

Funktionalismus Der Funktionalismus geht davon aus, dass geistige Zustände (Ideen, Glauben, Furcht, ...) interne (funktionale) Zustände eines komplexen System sind. Einzig die Funktion definiert die Semantik eines Systems. D.h. der Zustand S_1 des Systems A ist funktional äquivalent zu Zustand S_2 des Systems B, gdw. A und B bei gleicher Eingabe die gleiche Ausgabe liefern und in funktional äquivalente Zustände übergehen. Z.B. sind das Röhren-Radio und das Transistor-Radio funktional äquivalent, da sie dieselbe Funktion implementieren. Im Unterschied dazu sind Radio und CD-Player nicht äquivalent. Dieser Ansatz läuft darauf hinaus, dass der Mensch im Prinzip ein endlicher Automat mit Ein/Ausgabe ist, wenn auch mit sehr vielen Zuständen.

1.3.1 Die starke und die schwache KI-Hypothese

Im Rahmen der Künstlichen Intelligenz unterscheidet die Philosophie die schwache und die starke KI-Hypothese. Die *schwache KI-Hypothese* ist die Behauptung, dass Maschinen agieren können, *als ob* sie intelligent wären.

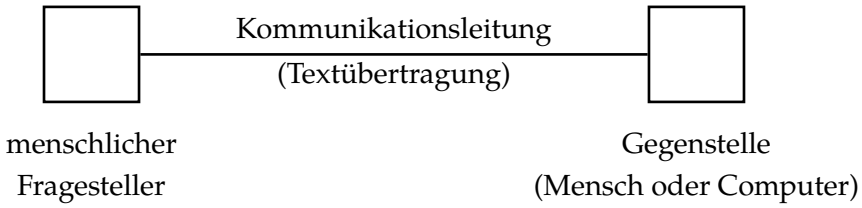
Die *starke KI-Hypothese* geht im Gegensatz dazu davon aus, dass Maschinen wirklich denken können, d.h. sie simulieren nicht nur das Denken.

Die schwache KI-Hypothese wird von den meisten als gegeben hingenommen. Die starke KI-Hypothese steht in der KI-Forschung im Allgemeinen nicht im Vordergrund, sondern eher die pragmatische Sichtweise, dass es irrelevant ist, ob das „Denken“ nur anscheinend oder tatsächlich passiert – hauptsache das System funktioniert.

Trotz allem sollte man sich als KI-Forscher über die ethisch-moralischen Konsequenzen seiner Forschung bewusst sein.

1.3.2 Der Turing-Test

Ein Test, der die starke KI-Hypothese nachweisen könnte, könnte der folgende von Alan Turing vorgeschlagene Test sein. Der sogenannte *Turingtest* besteht darin, dass ein menschlicher Fragesteller schriftliche Fragen an den Computer (die Gegenstelle) stellt und dieser darauf antwortet. Der Test ist bestanden, wenn der Fragesteller nicht erkennen kann, ob die Antworten von einem Computer oder von einem Menschen stammen, d.h. die Gegenstelle sollte wechselweise mal durch den Computer mal durch einen Mensch ersetzt werden.



Ein Problem des Turingtests ist, dass dieser nicht objektiv ist, da er von den Fähigkeiten des menschlichen Fragestellers abhängt, z.B. vom Wissen über aktuelle Fähigkeiten eines Computers. Die Aussagekraft kann durch wiederholen des Experiments mit mehreren Fragestellern erhöht werden.

Der Test ist absichtlich so ausgelegt, dass es keine physische Interaktion zwischen Mensch und Computer gibt (der Mensch sieht den Computer nicht und er stellt die Fragen schriftlich), da diese zusätzlichen Anforderungen nicht die Intelligenz ausmachen.

Hingegen verwendet der *totale Turingtest* zusätzlich ein Videosignal, so dass der Computer anhand dieses Signals auch den menschlichen Befrager analysieren kann (z.B. durch Gestenerkennung), und umgekehrt der Befrager feststellen kann, ob der Befragte über wahrnehmungsorientierte Fähigkeiten verfügt. Zusätzlich kann der Befrager physische Objekte an den Befragten übergeben. Der Computer muss daher über Fähigkeiten zur Objekterkennung und Robotik (Bewegen des Objekts) etc. verfügen.

Ein Wettbewerb, der sich am Turingtest orientiert, ist der seit 1991 vergebene Loebner-Preis[2]. Dabei werden Chat-Bots getestet und der von einer Jury am menschenähnlichsten ausgewählte Chat-Bot wird zum jährlichen Sieger gekürt. In jeder Runde des Tests

[2] http://www.loebner.net/Prizef/loebner-prize.html

kommuniziert ein Jury-Mitglied simultan mit einem Chat-Bot und einem Menschen ausschließlich über Textnachrichten in begrenzter Zeit. Im Anschluss entscheidet der Juror wer Mensch, wer Maschine ist. Es gibt zudem noch einmalige Preise für den ersten Chat-Bot, der es schafft die Jury überzeugend zu täuschen, ein Mensch zu sein, und schließlich den mit 100.000 USD dotierten Preis, für den ersten Chat-Bot, der dies auch bezüglich audio-visueller Kommunikation schafft. Beide einmalige Preise wurden bisher nicht vergeben.

Das Interessante an den Turing-ähnlichen Tests ist, dass sie ein halbwegs einsichtiges Kriterium festlegen, um zu entscheiden, ob eine Maschine menschliches Handeln simulieren kann oder nicht. Gegenargumente für den Turingtest sind vielfältig, z.B. könnte man ein System als riesige Datenbank implementieren, die auf jede Frage vorgefertigte Antworten aus der Datenbank liest und diese präsentiert. Ist ein solches System intelligent?

Andererseits gibt es Programme die manche Menschen in Analogie zum Turingtest täuschen können. Ein solches Programm ist das von J. Weizenbaum entwickelte Programm ELIZA[3], dass (als Softbot) einen Psychotherapeuten simuliert. Zum einen benutzt ELIZA vorgefertigte Phrasen für den Fall, dass das System nichts versteht (z.B. „Erzählen Sie mir etwas aus Ihrer Jugend.“), zum anderen wird mithilfe von Mustererkennung in der Eingabe des Benutzers nach Schlüsselwörtern gesucht und anschließend werden daraus Ausgaben (anhand von Datenbank und Schlussregeln) geniert (z.B. „Erzählen Sie mir etwas über xyz“).

Der Turing-Test wird in der Forschergemeinde nicht stark als angestrebtes Ziel verfolgt, aber aufgrund seiner Einfachheit demonstriert er eindrucksvoll, was künstliche Intelligenz bedeutet.

1.3.3 Das Gedankenexperiment „Chinesischer Raum“

Das von John Searle vorgeschlagenes Gedankenexperiment zum „Chinesischen Raum“ soll als Gegenargument zur starken KI-Hypothese dienen, d.h. es soll ein Nachweis sein, dass ein technisches System kein Verständnis der Inhalte hat, die verarbeitet werden, bzw. dass es keinen verstehenden Computer geben kann.

Das Experiment lässt sich wie folgt beschreiben: Jemand, der kein chinesisch versteht, sitzt in einem Raum. Es gibt dort chinesisch beschriftete Zettel und ein dickes Handbuch mit Regeln (in seiner Muttersprache), die jedoch ausschließlich angeben, welche chinesische Zeichen als Antwort auf andere chinesische Zeichen gegeben werden sollen. Durch einen Schlitz an der Tür erhält die Person Zettel mit chinesischen Zeichen. Anhand des Handbuchs und der bereits beschrifteten Zettel erzeugt die Person neue Zettel auf seinem Stapel und einen neuen beschrifteten Zettel (mit chinesischen Schriftzeichen), den er nach außen reicht.

[3] siehe z.B. http://www.med-ai.com/models/eliza.html.de

Es stellen sich nun die Fragen: Versteht die Person im Raum chinesisch? Versteht das Gesamtsystem chinesisch? John Searles Argument ist, dass kein Teil des Systems etwas vom Inhalt versteht, und daher weder die Person noch das System chinesisch versteht. Das Gegenargument (entsprechend dem Behaviorismus) ist, dass das Gesamtsystem etwas versteht, da das Verständnis beobachtbar ist.

1.3.4 Das Prothesenexperiment

Dieses Gedankenexperiment geht davon aus, dass die Verbindungen von Neuronen im Gehirn voll verstanden sind und dass man funktional gleiche Bauteile (elektronische Neuronen) herstellen und im Gehirn einsetzen kann. Das Experiment besteht nun darin, einzelne Neuronen durch elektronische Neuronen zu ersetzen. Die Fragen dabei sind: Wird sich die Funktionalität des Gehirns ändern? Ab welcher Anzahl von elektronischen Neuronen wird sich das Prothesen-Gehirn in einen Computer verwandeln, der nichts versteht. Die Folgerung aus diesem Experiment ist, dass entweder das Gehirn nur aus Neuronen besteht (d.h. das Prothesengehirn ändert nichts, die starke KI-Hypothese gilt daher), oder das es etwas gibt, das noch unbekannt ist (Bewusstsein, Geist, etc.).

1.3.5 Symbolverarbeitungshypothese vs. Konnektionismus

Ein *physikalisches Symbolsystem* basiert auf Symbolen, denen eine Bedeutung in der Realität zugeordnet werden kann. Aus einer eingegeben Symbolstruktur (z.B. ein String aus Symbolen o.ä.) werden sukzessive weitere Symbolstrukturen erzeugt und evtl. auch ausgegeben.

Die *Symbolverarbeitungshypothese* (von A. Newell and H. Simon) geht davon aus, dass ein physikalisches Symbolsystem so konstruiert werden kann, dass es intelligentes Verhalten zeigt (d.h. den Turingtest besteht). Überlegungen zu Quantencomputern und deren parallele, prinzipielle Fähigkeiten könnten hier zu neuen Ansätzen und Unterscheidungen führen. Im Moment gibt es aber noch keine verwendbaren Computer, die auf diesem Ansatz basieren.

M. Ginsberg charakterisiert die künstliche Intelligenz durch das Ziel der Konstruktion eines physikalischen Symbolsystems, das zuverlässig den Turingtest besteht.

Die *Hypothese des Konnektionismus* geht davon aus, dass man subsymbolische, verteilte, parallele Verarbeitung benötigt, um eine Maschine zu konstruieren, die intelligentes Verhalten zeigt (den Turingtest besteht).

Die technische Hypothese dazu ist, dass man künstliche neuronale Netze benötigt. Als Gegenargument dazu wäre zu nennen, dass man künstliche neuronale Netze auch (als Software) programmieren und auf normaler Hardware simulieren kann.

1.4 KI-Paradigmen

Zwei wesentliche Paradigmen der künstlichen Intelligenz sind zum einen das physikalische Symbolsystem (erstellt durch explizites Programmieren und verwenden von Logiken, Schlussregeln und Inferenzverfahren). Die Stärken dieses Paradigmas bestehen darin, dass man gute Ergebnisse in den Bereichen Ziehen von Schlüssen, strategische Spiele, Planen, Konfiguration, Logik, ... erzielen kann.

Ein anderes Paradigma konzentriert sich auf Lernverfahren und verwendet insbesondere künstliche neuronale Netze. Dessen Stärken liegen vorallem in der Erkennung von Bildern, der Musterverarbeitung, der Verarbeitung verrauschter Daten, dem maschinelles Lernen, und adaptiven Systeme (Musterlernen).

Trotz allem benötigt man für ein komplexes KI-System im Allgemeinen alle Paradigmen.

1.4.1 Analyse und Programmieren von Teilaspekten

Die folgende Arbeitshypothese steht hinter der Untersuchung von Teilaspekten der Künstlichen Intelligenz:

Untersuche und programmiere Teilaspekte an kleinen Beispielen. Wenn das klar verstanden ist, dann untersuche große Beispiele und kombiniere die Lösungen der Teilaspekte.

Das ergibt zwei Problematiken:

Kombination der Lösungen von Teilaspekten. Die Lösungen sind oft kombinierbar, aber es gibt genügend viele Beispiele, die inkompatibel sind, so dass die Kombination der Methoden oft nicht möglich ist bzw. nur unter Verzicht auf gute Eigenschaften der Teillösungen.

realistische Beispiele (scale up) Wenn die Software-Lösung für kleine Beispiele (Mikrowelten) funktioniert, heißt das noch nicht, dass diese auch für realistische Beispiele (Makrowelten) sinnvoll funktioniert bzw. durchführbar ist.

1.4.2 Wissensrepräsentation und Schlussfolgern

Wissensrepräsentationshypothese: (Smith, 1982) "Die Verarbeitung von Wissen läßt sich trennen in: Repräsentation von Wissen, wobei dieses Wissen eine Entsprechung in der realen Welt hat; und in einen Inferenzmechanismus, der Schlüsse daraus zieht."

Dieses Paradigma ist die Basis für alle Programme in Software/Hardware, deren innere Struktur als Modellierung von Fakten, Wissen, Beziehungen und als Operationen, Simulationen verstanden werden kann.

Ein Repräsentations- und Inferenz-System (representation and reasoning system) besteht, wenn man es abstrakt beschreibt, aus folgenden Komponenten:

1. Eine *formale Sprache*: Zur Beschreibung der gültigen Symbole, der gültigen syntaktischen Formen einer Wissensbasis, der syntaktisch korrekten Anfragen usw.
   Im allgemeinen kann man annehmen, dass eine Wissensbasis eine Multimenge von gültigen Sätzen der formalen Sprache ist.

2. Eine *Semantik*: Das ist ein Formalismus, der den Sätzen der formalen Sprache eine Bedeutung zuweist.
   Im allgemeinen ist die Semantik modular, d.h. den kleinsten Einheiten wird eine Bedeutung zugeordnet, und darauf aufbauend den Sätzen, wobei die Bedeutung der Sätze auf der Bedeutung von Teilsätzen aufbaut.

3. Eine *Inferenz-Prozedur* (operationale Semantik) die angibt, welche weiteren Schlüsse man aus einer Wissensbasis ziehen kann. I.a. sind diese Schlüsse wieder Sätze der formalen Sprache.
   Diese Inferenzen müssen korrekt bzgl. der Semantik sein.

Dieses System kann auf Korrektheit geprüft werden, indem man einen theoretischen Abgleich zwischen Semantik und operationaler Semantik durchführt.

Die *Implementierung* des Repräsentations- und Inferenz-Systems besteht aus:

1. Parser für die formale Sprache

2. Implementierung der Inferenzprozedur.

Interessanterweise trifft diese Beschreibung auch auf andere Formalismen zu, wie Programmiersprachen und Logische Kalküle.

1.5 Bemerkungen zur Geschichte zur Geschichte der KI

Wir geben einen groben Abriss über die geschichtliche Entwicklung des Gebiets der künstlichen Intelligenz.

1950: A. Turing: Imititationsspiel

1956 J. McCarthy Dartmouth Konferenz: Start des Gebietes “artificial intelligence“ und Formulierung der Ziele.

1957- 1962 “allgemeiner Problemlöser“
Entwicklung von LISP (higher-order, Parser und Interpreter zur Laufzeit)
Dameprogramm von A. Samuels (adaptierend “lernend“)
Newell & Simon: GPS (General Problem Solver)

1963-1967 spezialisierte Systeme

- semantische Verarbeitung (Benutzung von Spezialwissen über das Gebiet)

- Problem des Common Sense
- Resolutionsprinzip im automatischen Beweisen

1968- 1972 Modellierung von Mikrowelten

- MACSYMA mathematische Formel-Manipulation
- DENDRAL Expertensystem zur Bestimmung von organischen Verbindungen mittels Massenspektroskopie
- SHRDLU, ELIZA: erste Versuche zur Verarbeitung natürlicher Sprache

1972-77 Wissensverarbeitung als Technologie

- medizinisches Expertensystem: MYCIN
- D. Lenats AM (automated mathematics)
- KI-Sprachentwicklungen: PLANNER, KRL PROLOG KL-ONE

1977- Entwicklung von Werkzeugen Trend zur Erarbeitung von Grundlagen;

- Expertensystem-Schalen (E-MYCIN)
- Fifth generation project in Japan (beendet)
- Computerlinguistik
- künstliche neuronale Netze
- logisches Programmieren
- 1985 Doug Lenats CYC

aktuelle Forschungsrichtungen

- Technologie zum textuellen Sprachverstehen und zur Ermöglichung von Mensch-Computer-Dialogen. (Wahlster: Verbmobil-Projekt)
- Robotik-Ansatz: (Embodied artificial intelligence, R. Pfeifer). sinngemäß: Eine gemeinsame Untersuchung und Entwicklung von Sensorik, Motorik, d.h. physikalische Gesetzmäßigkeiten des Roboters und der Steuerung mittels Computer (neuronalen-Netzen) ist notwendig: „intelligentes Insekt“
- Automatische Deduktion
- Logik
- Robotik
- künstliche neuronale Netze
- …

1.5.1 Schlussfolgerungen aus den bisherigen Erfahrungen:

- Das Fernziel scheint mit den aktuellen Methoden, Techniken, Hardware und Software nicht erreichbar
- Motivationen und Visionen der KI sind heute in der Informatik verbreitet.
- Die *direkte Programmierung von Systemen* (ohne sehr gute Lernalgorithmen) hat Grenzen in der Komplexität des einzugebenden Modells (Regeln, Fakten, ...); Auch wird es zunehmend schwerer, die Systeme und die Wissensbasis zu warten und konsistent zu ändern.
  Das Projekt CYC[4] von Doug Lenat zur Erzeugung eines vernünftigen Programms mit Alltagswissen durch Eingabe einer Enzyklopädie hat nicht zum erhofften durchschlagenden Erfolg geführt.
- Manche Gebiete wie (computerunterstützte) „Verarbeitung natürlicher (schriftlicher oder gesprochener) Sprache“, Robotik, Automatische Deduktion, . . . haben sich zu eigenständigen Forschungsgebieten entwickelt.
- Forschungsziele sind aktuell eher spezialisierte Aspekte bzw. anwendbare Verfahren:
  Logik und Inferenzen, Mensch-Maschine-Kommunikation, Lernverfahren (adaptive Software), Repräsentationsmechanismen; eingebettete KI, nicht-Standard-Logiken und Schlussfolgerungssysteme, ...

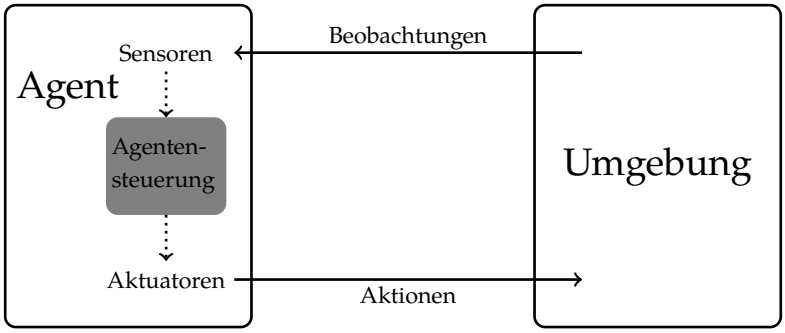
1.6 Intelligente Agenten

Unter dem Oberbegriff *Agenten* kann man alle KI-Systeme einordnen.

Wir stellen einige einfache Überlegungen zu Agenten und deren Interaktionsmöglichkeiten mit ihrer Umgebung an.

Ein Agent hat

- *Sensoren* zum Beobachten seiner Umgebung und
- *Aktuatoren* (Aktoren; Effektoren) um die Umgebung zu manipulieren.



[4] http://www.cyc.com/

Er macht *Beobachtungen* (Messungen, percept). Die (zeitliche) Folge seiner Beobachtungen nennt man *Beobachtungssequenz* (percept sequence). Er kann mittels seiner Aktuatoren Aktionen ausführen, die die Umgebung, evtl. auch seine Position, beeinflussen. Wenn man das Verhalten des Agenten allgemein beschreiben will, dann kann man das als

$$\textit{Agentenfunktion}\colon \{\text{Beobachtungsfolgen}\} \to \{\text{Aktionen}\}.$$

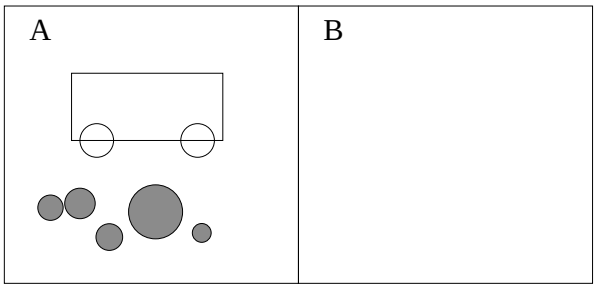
beschreiben.

Diese Agentenfunktion kann man durch das *Agentenprogramm* definieren. Prinzipiell kann man diese Funktion durch eine vollständige Tabelle beschreiben, oder durch einen Algorithmus.

Beispiel 1.6.1 (Staubsaugerwelt (Russell & Norvig, 2010))**.** *Das ist nur eine Modellwelt, die man variieren kann, um die Problematik und die Effekte zu illustrieren, und anhand derer man sich leichter verdeutlichen kann, welche Konzepte eine Rolle spielen.*

Der einfachste Fall ist wie folgt modelliert:

- *Es gibt nur zwei Orte an denen sich der Agent (Staubsauger) aufhalten kann: Quadrate A und B (bzw. Räume A, B)*
- *Jedes Quadrat kann Dreck enthalten oder nicht. Der Agent kann nur sehen, ob das Quadrat in dem er sich aufhält, dreckig ist: Sauber/ Dreckig, D.h. seine Beobachtung ergibt einen Booleschen Wert.*
- *Die mögliche Aktionen des Agenten sind: `InsAndereQuadrat`, `Saugen` und `NichtsTun`.*



Einige implizite Annahmen sind in diesem Modell versteckt:

- *Der Agent ist nur an einem Ort.*
- *Aktionen kann er nur zur aktuellen Zeit und am aktuellen Ort vornehmen.*
- *Die Umgebung kann sich unabhängig vom Agenten verändern: Hier nur der Zustand „Dreck oder kein Dreck im Quadrat A oder B“.*
- *Normalerweise nimmt man auch an, dass der Dreck nicht von selbst verschwindet. Jedoch kann er nicht seinen eigenen Ort beobachten, oder die Anzahl der möglichen Quadrate (Räume).*

- *Der obige Agent kann nicht beobachten, wo er ist, und seine Aktionen sind immer ausführbar. Er kann nur eine Aktion gleichzeitig ausführen.*

Die Aufgabe oder Fragestellung ist nun: Wann ist der Agent (das zugehörige Programm) gut / vernünftig bzw. intelligent ? Dazu braucht man eine Vorgabe: ein (externes) *Performanzmaß*, d.h. eine *Leistungsbewertung* des Agenten. Z.B:

- Alles soll immer maximal sauber sein. Konsequenz wäre, dass der Staubsauger immer hin und her fährt und saugt.
- Alles soll möglichst sauber sein, und der Stromverbrauch soll möglichst klein sein. Hier kann man diskrete Zeit annehmen, pro Zeiteinheit jeweils eine Anzahl Punkte pro sauberes Quadrat vergeben und jeden Wechsel des Quadrats mit einem bestimmten Abzug versehen. Die mittlere Punktzahl pro Zeit wäre dann das passende Maß.
- Alles soll möglichst sauber sein, und der Staubsauger soll möglichst wenig stören. Der Agent kann nicht beobachten, ob er stört.

Der optimale agierende Agent ist der *intelligente* Agent.

Man sieht, dass abhängig vom Performanzmaß jeweils andere Agenten als optimal zählen. Im allgemeinen ist das Performanzmaß nicht vom Agenten selbst zu berechnen, da er nicht ausreichend viele Beobachtungen machen kann. Normalerweise muss man Kompromisse zwischen den verschiedenen Kriterien machen, da nicht alle Kriterien gleichzeitig optimal zu erfüllen sind, und da auch nicht alle Kriterien zu berechnen bzw. effizient zu berechnen sind.

1.6.1 Gesichtspunkte, die die Güte des Agenten bestimmen

- Das Performanzmaß
- Das Vorwissen über die Umgebung
- Die möglichen Aktionen
- Die aktuelle Beobachtungsfolge

Definition 1.6.2. *Ein* vernünftiger (intelligenter, rationaler) Agent *ist derjenige, der stets die optimale Aktion bzgl des Performanzmaßes wählt, aufgrund seiner Beobachtungsfolge und seines Vorwissens über die Umgebung.*

Allerdings muss man beachten, dass man den vernünftigen Agenten nicht immer implementieren kann, da der Agent oft das Performanzmaß nicht berechnen kann. Zum Vorwissen kann hier z.B. die stochastische Verteilung des Verschmutzung über die Zeit gelten, d.h. wie groß ist die Wahrscheinlichkeit, dass das Quadrat A in der nächsten Zeiteinheit wieder dreckig wird.

1.6.2 Lernen

Da normalerweise das Vorwissen über die Umgebung nicht ausreicht, oder sich die Umgebung und deren Langzeitverhalten ändern kann, ist es für einen guten Agenten notwendig

- mittels der Sensoren Wissen über die Umgebung zu sammeln;
- lernfähig zu sein, bzw. sich adaptiv zu verhalten, aufgrund der Beobachtungssequenz.

Z.B. kann eine Erkundung der Umgebung notwendig sein, wenn der Staubsauger in einer neuen Umgebung eingesetzt wird, und das Wissen dazu nicht vorgegeben ist.

Ein Agent wird als *autonom* bezeichnet, wenn der Agent eher aus seinen Beobachtungen lernt und nicht auf vorprogrammierte Aktionen angewiesen ist.

1.6.3 Verschiedene Varianten von Umgebungen

Umgebungen können anhand verschiedener Eigenschaften *klassifiziert* werden. Die Eigenschaften sind:

- *Vollständig beobachtbar vs. teilweise beobachtbar*.
  Der Staubsauger kann z.B. nur sein eigenes Quadrat beobachten.
- *Deterministisch vs. Stochastisch*.
  Der Dreck erscheint zufällig in den Quadraten.
- *Episodisch vs. sequentiell*.
  Episodisch: Es gibt feste Zeitabschnitte, in denen beobachtet agiert wird, und die alle unabhängig voneinander sind.
  Sequentiell. Es gibt Spätfolgen der Aktionen.
- *Statisch vs. Dynamisch*
  Dynamisch: die Umgebung kann sich während der Nachdenkzeit des Agenten verändern. Wir bezeichnen die Umgebung als semi-dynamisch, wenn sich während der Nachdenkzeit zwar die Umgebung nicht verändert, jedoch das Performanzmaß. Ein Beispiel ist Schachspielen mit Uhr.
- *Diskret vs. Stetig*.
- *Ein Agent oder Multiagenten*.
  Bei Multiagenten kann man unterscheiden zwischen Gegnern / Wettbewerber / Kooperierenden Agenten. Es gibt auch entsprechende Kombinationen.

Beispiel 1.6.3. *Eine Beispieltabelle aus (Russell & Norvig, 2010)*

Arbeits-umgebung	**Beobacht-bar**	**Deter-min-istisch**	**Episo-disch**	**Statisch**	**Diskret**	**Agenten**
Kreuzworträtsel	*vollst.*	*det.*	*seq.*	*statisch*	*diskret*	*1*
Schach mit Uhr	*vollst.*	*det.*	*seq.*	*semi*	*diskret*	*n*
Poker	*teilw.*	*stoch.*	*seq.*	*statisch*	*diskret*	*n*
Backgammon	*vollst.*	*stoch.*	*seq.*	*statisch*	*diskret*	*n*
Taxifahren	*teilw.*	*stoch.*	*seq.*	*dyn.*	*stetig*	*n*
Medizinische Diagnose	*teilw.*	*stoch.*	*seq.*	*dyn.*	*stetig*	*1*
Bildanalyse	*vollst.*	*det.*	*episod.*	*semi*	*stetig*	*1*
Interaktiver Englischlehrer	*teilw.*	*stoch.*	*seq.*	*dyn.*	*diskret*	*n*

1.6.4 Struktur des Agenten

Ein Agent besteht aus:

- Physikalischer Aufbau inkl. Sensoren und Aktuatoren; man spricht von der *Architektur des Agenten*.
- dem Programm des Agenten

Wir beschreiben in allgemeiner Form einige Möglichkeiten:

- Tabellengesteuerter Agent: (endlicher Automat) mit einer Tabelle von Einträgen:
  Beobachtungsfolge1 $\mapsto$ Aktion1
  Beobachtungsfolge2 $\mapsto$ Aktion2
  ... ...

  Diese Beschreibung ist zu allgemein und eignet sich nicht als Programmieridee; die Tabelle ist einfach zu groß. Leicht abgeändert und optimiert, nähert man sich „normalen“ Programmen:

- Agent mit Zustand
  implementiert eine Funktion der Form (Zustand, Beobachtung) $\mapsto$ Aktion, Zustand'

  Wobei der Zustand auch die gespeicherte Folge (Historie) der Beobachtungen sein kann.

- Einfacher Reflex-Agent:
  Tabellengesteuert (endlicher Automat) mit einer Tabelle von Einträgen:
  Beobachtung1 $\mapsto$ Aktion1
  Beobachtung2 $\mapsto$ Aktion2

- Modellbasierte Strukturierung:
  Der Zustand wird benutzt, um die Umgebung zu modellieren und deren Zustand zu speichern. Z.B. beim Staubsauger: inneres Modell des Lageplan des Stockwerks; welche Räume wurden vom Staubsauger gerade gesäubert, usw.

  – Zweckbasierte Strukturierung (goalbased, zielbasiert): Die Entscheidung, welche Aktion als nächste ausgeführt wird, hängt davon ab, ob damit ein vorgegebenes Ziel erreicht wird.

    Dr Agent kann erkennen, ob er das Ziel erreicht hat.

  – Nutzenbasierte Strukturierung (utility-based, nutzenbasiert): Die Entscheidung, welche Aktion als nächste ausgeführt wird, hängt davon ab, ob damit eine vorgegebene (interne) Nutzenfunktion (internes Gütemaß) verbessert wird. Da das Performanzmaß vom Agenten selbst meist nicht berechnet werden kann, da ihm die Informationen fehlen, gibt es eine *Bewertungsfunktion (utility function)*, die der Agent berechnen und als Bewertung verwenden kann.

- Lernende Agenten (s.o.): Das Vorsehen aller möglichen Aktionen auf alle Varianten der Beobachtung ist, auch stark optimiert, sehr aufwändig, manchmal unmöglich, da man leicht Fälle beim Programmieren übersieht, und oft die optimale Aktion nicht so einfach aus der jeweiligen Beobachtungsfolge ausrechnen kann. Die Speicherung der ganzen Beobachtungsfolge kann zudem sehr viel Speicher benötigen.

  Lernverfahren bieten hier einen Ausweg und erhöhen die Autonomie des Agenten.

  Da man eine Meta-Ebene ins Programm einführt, muss man unterscheiden zwischen

  – Lernmodul

  – Ausführungmodul

  Das Ausführungsmodul ist das bisher betrachtete Programm, das Aktionen berechnet, das Lernmodul verwendet die Beobachtungsfolge, um das Ausführungsmodul bzw. dessen Parameter (im Zustand gespeichert) zu verändern.

  Zum Lernen benötigt man eine Rückmeldung bzw. Bewertung (critic) der aktuellen Performanz, um dann das Ausführungsmodul gezielt zu verbessern. Das kann eine interne Bewertungsfunktion sein, aber auch ein von Außen gesteuertes Training: Es gibt verschiedene Lernmethoden dazu: online, offline, die sich weiter untergliedern. Offline-Lernen kann man als Trainingsphase sehen, wobei anhand vorgegebener oder zufällig erzeugter „Probleme“ die optimale Aktion trainiert werden kann. Das Lernverfahren kann z.B. durch Vorgabe der richtigen Aktion durch einen Lehrer, oder mittels Belohnung / Bestrafung für gute / schlechte Aktionsauswahl das *Training* durchführen.

2
Suchverfahren

2.1 Algorithmische Suche

Oft hat man in einer vorgegebenen Problemlösungs-Situation die Aufgabe, einen bestimmten Zweck zu erreichen, einen (Aktions-)Plan für einen Agenten zu entwickeln, einen Weg zu finden o.ä. Hierzu benötigt man zunächst eine Repräsentation des aktuellen Zustandes sowie der Möglichkeiten, die man hat, um sein Ziel zu erreichen. Im Allgemeinen gibt es kein effizientes Verfahren, um den Plan zu berechnen, denn nicht alle Probleme haben so einfache Lösungen wie z.B. die Aufgabe zu zwei gegebenen Zahlen deren Produkt zu bestimmen. Man benötigt daher *Suchverfahren*. In diesem Kapitel werden wir einige Suchverfahren und deren Anwendung und Eigenschaften erörtern.

Beispiele für solche Suchprobleme sind:

- Spiele: Suche nach dem besten Zug
- Logik: Suche nach einer Herleitung einer Aussage
- Agenten: Suche nach der optimalen nächsten Aktion
- Planen: Suche nach einer Folge von Aktionen eines intelligenten Agenten.
- Optimierung: Suche eines Maximums einer mehrstelligen Funktion auf den reellen Zahlen

Wir beschränken uns auf die Fälle, in denen die Aufgabe eindeutig und exakt formulierbar ist.

Die exakte Repräsentation der Zustände, der Übergänge, der Regeln ist Teil der Programmierung und Optimierung der Suche. Meist hat man:

- Anfangssituationen
- Kalkülregeln, die es erlauben aus einer Situation die nächste zu berechnen (Spielregeln), die sogenannte *Nachfolgerfunktion*
- Ein Test auf Zielsituation

Beispiel 2.1.1.

Schach: *Gegeben ist als Anfangssituation eine Stellung im Spiel, die Nachfolgerfunktion entspricht den möglichen Zügen. Eine Zielsituation ist erreicht, wenn man gewonnen hat. Gesucht ist nun ein Zug, der zum Gewinn führt.*

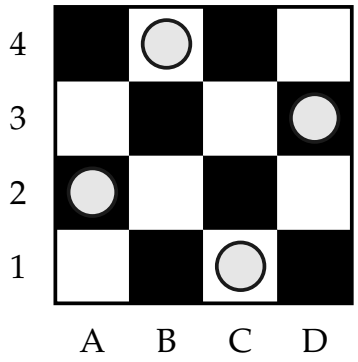
Deduktion: *Gegeben ist als Anfangssituation eine zu beweisende Aussage A und einer Menge von Axiomen und evtl. schon bewiesenen Sätzen und Lemmas. Gesucht ist ein Beweis für A.*

Planen: *Gegeben eine formale Beschreibung des interessierenden Bereichs; z.B. Fahrplan, Anfangsort, Zeit, Zielort der zu planenden Reise. Gesucht ist ein Plan, der die Reise ermöglicht.*

2.1.1 Beispiel: das n-Damen Problem

Beim n-Damen Problem[1] ist ein $n \times n$-Schachfeld gegeben und gesucht ist eine Lösung dafür n Damen auf dem Schachbrett so zu verteilen, dass keine Dame eine andere bedroht (Damen bedrohen sich, wenn sie auf einer horizontalen, vertikalen oder diagonalen Linie stehen).

Eine mögliche Lösung für das 4-Damen Problem ist:



Ein möglicher Ablauf der Suche, wenn $n = 4$ ist, ist der folgende:

1. Dame A-1
2. Dame A-2; Bedrohung
3. Dame A-2 nach B-2 Bedrohung (Diagonale)
4. Dame B-2 nach C-2
5. Dame A-3; Bedrohung
6. Dame A-3 nach B-3; Bedrohung

usw

9. Dame auf Reihe 3 zurücknehmen, Dame C-2 auf D-2
10. Dame A-3
11. Dame A-3 nach B-3

usw.

[1] Wir betrachten das n-Damen Problem hier nur als Demonstrationsbeispiel für die Suche, denn es es gibt bessere, linear berechenbare systematische Lösungen.

2.1.2 Beispiel: „Missionare und Kannibalen“

3 Missionare und 3 Kannibalen sind auf einer Seite eines Flusses. Es gibt ein Boot, in dem höchstens zwei Personen rudern können.
Bedingung: Auf keiner Uferseite dürfen jemals mehr Kannibalen als Missionare sein.
Gesucht: ein Plan zur Überquerung des Flusses, der diese Bedingung einhält.

Kodierungsmöglichkeiten des Problems:
Wir betrachten aufeinanderfolgende Zustände, die aus den Anzahlen der Missionare und der Kannibalen pro Ufer sowie der Position des Bootes bestehen:

1. Am Anfang:	3M, 3K, L, 0M, 0K
Ziel ist die Situation:	0M, 0K, R, 3M, 3K

mögliche zweite Situationen:

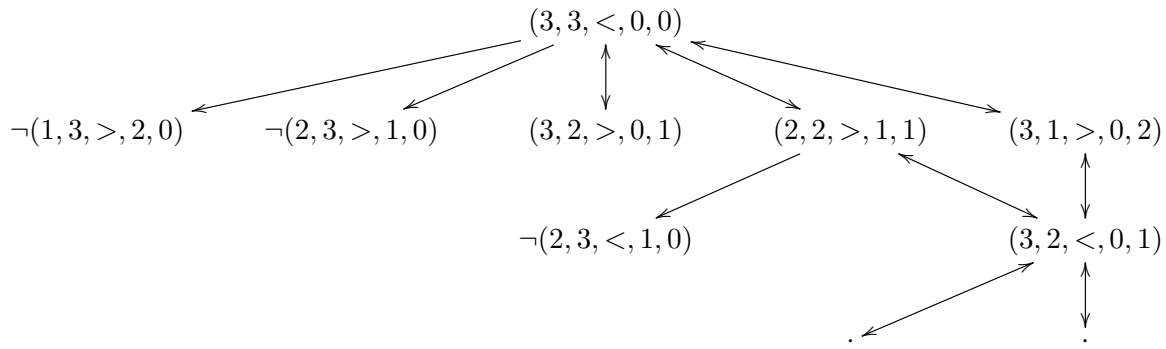
2.1	2M, 3K, R, 1M, 0K
2.2	1M, 3K, R, 2M, 0K
2.3	2M, 2K, R, 1M, 1K
2.4	3M, 2K, R, 0M, 1K
2.5	3M, 1K, R, 0M, 2K

Man sieht: 2.1 und 2.2 sind verboten, die anderen erlaubt.

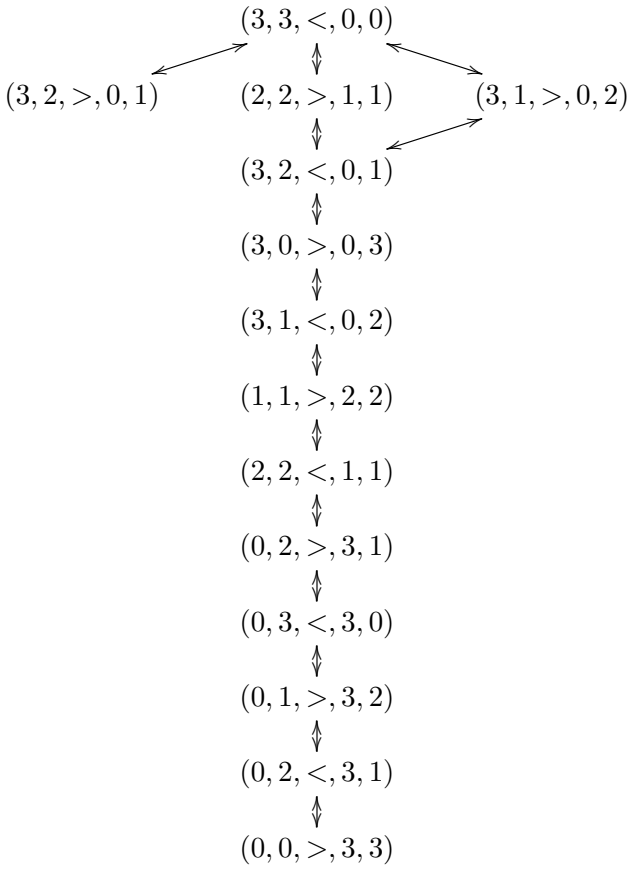
Die naive Suche würde Zyklen enthalten, die dem Hin-und-Her-Rudern entsprechen, ohne dass sich etwas ändert. Es gibt offensichtliche Verbesserungen der naiven Suche:

- Suche nicht in einem Baum aller Möglichkeiten, sondern in einem gerichteten Graphen.
- schon versuchte Situationen werden nicht noch einmal verfolgt.

Ein Diagramm des Zusammenhangs der ersten Zustände. Verbotene Zustände sind mit ¬ markiert.



Der Suchgraph ohne ungültige Situationen ist:



Allerdings ist das Problem doch nicht ganz eindeutig formuliert: Es gibt folgende Probleme der Repräsentation:

1. Die Repräsentation kann Eindeutigkeit herbeiführen, die in der Problemstellung nicht enthalten war. Zum Beispiel ist das Ein-und Aussteigen aus dem Boot nicht mitberücksichtigt: Was passiert, wenn ein Kannibale mit dem Boot am Ufer ankommt, und dort befinden sich ein Missionar und ein Kannibale, und der Missionar will im Boot zurückfahren. Ist das erlaubt oder nicht? Repräsentiert man andere Zustände, z.B. nur die Zustände während des Bootfahrens, kann es erlaubt sein.

2. Die Repräsentation bestimmt die Struktur des Suchraumes mit. Z.B. Wenn man die Missionare und Kannibalen mit Namen benennt, und als Zustand die Menge der Personen auf der Startseite des Flusses speichert und zusätzlich B genau dann in die Menge einfügt, wenn das Boot auf der Startseite des Flusses ist. Dann ist die Anfangssituation: $\{M_1, M_2, M_3, K_1, K_2, K_3, B\}$
   Jetzt gibt es 21 Folgesituationen:
   1. $\{M_2, M_3, K_1, K_2, K_3\}$
   2. $\{M_1, M_3, K_1, K_2, K_3\}$

3. $\{M_1, M_2, K_1, K_2, K_3\}$
4. $\{M_1, M_2, M_3, K_2, K_3\}$
5. $\{M_1, M_2, M_3, K_1, K_3\}$
6. $\{M_1, M_2, M_3, K_1, K_2\}$
7. $\{M_3, K_1, K_2, K_3\}$
8. $\{M_2, K_1, K_2, K_3\}$
9. $\{M_2, M_3, K_2, K_3\}$
10. $\{M_2, M_3, K_1, K_3\}$
11. $\{M_2, M_3, K_1, K_2\}$
12. $\{M_1, K_1, K_2, K_3\}$
13. $\{M_1, M_3, K_2, K_3\}$
14. $\{M_1, M_3, K_1, K_3\}$
15. $\{M_1, M_3, K_1, K_2\}$
16. $\{M_1, M_2, K_2, K_3\}$
17. $\{M_1, M_2, K_1, K_3\}$
18. $\{M_1, M_2, K_1, K_2\}$
19. $\{M_1, M_2, M_3, K_3\}$
20. $\{M_1, M_2, M_3, K_2\}$
21. $\{M_1, M_2, M_3, K_1\}$

Die Anzahl der erlaubten Folgesituationen ist 12.

Offensichtlich ist in dieser Repräsentation die Suche sehr viel schwieriger.

3. Die Repräsentation hat auch eine Wahl bei der Festlegung der Knoten und der Nachfolgerfunktion. Z.B. kann man bei n-Dame auch als Nachfolger *jede* Situation nehmen, bei der eine Dame mehr auf dem Schachbrett steht

Das Finden einer geeigneten Repräsentation und die Organisation der Suche erfordert Nachdenken und ist Teil der Optimierung der Suche.

2.1.3 Suchraum, Suchgraph

Wenn man vom Problem und der Repräsentation abstrahiert, kann man die Suche nach einer Lösung als Suche in einem gerichteten Graphen (*Suchgraph, Suchraum*) betrachten. Hier ist zu beachten, dass der gerichtete Graph nicht explizit gegeben ist, er kann z.B. auch unendlich groß sein. Der Graph ist daher *implizit* durch Erzeugungsregeln gegeben. Die Suche und partielle Erzeugung des Graphen sind somit verflochten. Der Suchgraph besteht aus:

- *Knoten* Situation, Zustände
- *Kanten* in Form einer Nachfolger-Funktion N, die für einen Knoten dessen Nachfolgesituationen berechnet.
- *Anfangssituation*
- *Zielsituationen*: Eigenschaft eines Knotens, die geprüft werden kann.

Diesen Graphen (zusammen mit seiner Erzeugung) nennt man auch *Suchraum*.
Wir definieren einige wichtige Begriffe.

Definition 2.1.2. *Die* Verzweigungrate des Knotens K *(branching factor) ist die Anzahl der direkten Nachfolger von K, also die Mächtigkeit der Menge $N(K)$.*

Die mittlere Verzweigungsrate des Suchraumes *ist entsprechend die durchschnittliche Verzweigungrate aller Knoten.*

Die Größe des Suchraumes ab Knoten K in Tiefe d *ist die Anzahl der Knoten, die von K aus in d Schritten erreichbar sind, d.h. die Mächtigkeit von $\overline{N}^d(K)$, wobei*

$$\overline{N}^1(M) = \bigcup\{N(L) \mid L \in M\} \text{ und } \overline{N}^i(K) = \overline{N}(\overline{N}^{i-1}(K)).$$

Eine Suchstrategie *ist eine Vorgehensweise zur Durchmusterung des Suchraumes. Eine Suchstrategie ist* vollständig, *wenn sie in endlich vielen Schritten (Zeit) einen Zielknoten findet, falls dieser existiert.*

Bemerkung 2.1.3. *Im Allgemeinen hat man eine mittlere Verzweigungsrate $c > 1$. Damit ist der Aufwand der Suche exponentiell in der Tiefe des Suchraums. Das nennt man auch* kombinatorische Explosion.

Die meisten Suchprobleme sind NP-vollständig bzw. NP-hart. In diesem Fall benötigen alle bekannten Algorithmen im schlechtesten Fall (mindestens) exponentiellen Zeitaufwand in der Größe des Problems.

2.1.4 Prozeduren für nicht-informierte Suche (Blind search)

Wir betrachten zunächst die nicht-informierte Suche, wobei nicht-informiert bedeutet, dass die Suche nur den Graphen und die Nachfolgerfunktion verwenden darf (kann), aber keine anderen Informationen über die Knoten, Kanten usw. verwenden kann.

Die *Parameter* sind

- Menge der initialen Knoten
- Menge der Zielknoten, bzw. eindeutige Festlegung der Eigenschaften der Zielknoten
- Nachfolgerfunktion N

Algorithmus **Nicht-informierte Suche**

Datenstrukturen: L sei eine Menge von Knoten, markiert mit dem dorthin führenden Weg.

Eingabe: Setze $L :=$ Menge der initialen Knoten mit leerem Weg

Algorithmus:

1. Wenn L leer ist, dann breche ab.
2. Wähle einen beliebigen Knoten K aus L.
3. Wenn K ein Zielknoten ist, dann gebe aus: Zielknoten und Weg dorthin (d.h. Weg im Graphen dorthin)
4. Wenn K kein Zielknoten, dann nehme Menge $N(K)$ der direkten Nachfolger von K und verändere L folgendermaßen:
   $L := (L \cup N(K)) \setminus \{K\}$ (Wege entsprechend anpassen)
   Mache weiter mit Schritt 1

Wir betrachten im folgenden Varianten der blinden Suche, die insbesondere das Wählen des Knotens K aus L eindeutig durchführen.

2.1.4.1 Varianten der blinden Suche: Breitensuche und Tiefensuche

Die Tiefensuche verwendet anstelle der Menge der L eine Liste von Knoten und wählt stets den ersten Knoten der Liste als nächsten zu betrachtenden Knoten. Außerdem werden neue Nachfolger stets *vorne* in die Liste eingefügt, was zur Charakteristik führt, dass zunächst in der Tiefe gesucht wird.

Algorithmus **Tiefensuche**

Datenstrukturen: L sei eine Liste (Stack) von Knoten, markiert mit dem dorthin führenden Weg.

Eingabe: Füge die initialen Knoten in die Liste L ein.

Algorithmus:

1. Wenn L die leere Liste ist, dann breche ab.
2. Wähle ersten Knoten K aus L, sei R die Restliste.
3. Wenn K ein Zielknoten ist, dann gebe aus: Zielknoten und Weg dorthin (d.h. Weg im Graphen dorthin)
4. Wenn K kein Zielknoten, dann sei $N(K)$ die (geordnete) Liste der direkten Nachfolger von K, mit dem Weg dorthin markiert
   $L := N(K)$ ++ R. (wobei ++ Listen zusammenhängt)
   Mache weiter mit 1.

Eine einfache, rekursive Implementierung in Haskell ist:

```
dfs ::(a -> Bool)    -- Zieltest (goal)
    -> (a -> [a])     -- Nachfolgerfunktion (succ)
    -> [a]            -- Startknoten
    -> Maybe (a, [a]) -- Ergebnis: Just (Zielknoten,Pfad) oder Nothing

dfs goal succ stack =
  -- Einfuegen der Anfangspfade, dann iterieren mit go
  go [(k,[k]) | k <- stack]
 where
  go [] = Nothing -- Alles abgesucht, nichts gefunden
  go ((k,p):r)
   | goal k    = Just (k,p) -- Ziel gefunden
   | otherwise = go ([(k',k':p) | k' <- succ k] ++ r)
```

Beachte, dass die explizite Speicherung der Pfade in Haskell nicht viel Effizienz kostet (da diese lazy ausgewertet werden). In imperativen Sprachen kann man durch geschickte Speicherung der Liste den aktuellen Suchpfad in der Liste L speichern: Jeder Eintrag ist ein Knoten mit Zeiger auf den nächsten Nachbarknoten.

Bemerkung 2.1.4 (Eigenschaften der Tiefensuche)**.** *Die* Komplexität *(worst-case) des Verfahrens ist bei fester Verzweigungsrate $c > 1$:*

Platz: *linear in der Tiefe. (wenn die Verzweigungsrate eine obere Schranke hat)*

Zeit: *entspricht der Anzahl der besuchten Knoten (als Baum gesehen), d.h. Aufwand ist exponentiell in der Tiefe.*

Beachte, dass Tiefensuche nicht vollständig ist, wenn der Suchgraph unendlich groß ist, denn die Suche kann am Ziel vorbeilaufen, und für immer im unendlichen langen Pfad laufen.

2.1.4.2 Pragmatische Verbesserungsmöglichkeiten der Tiefensuche

Einige offensichtliche, einfache Verbesserungsmöglichkeiten der Tiefensuche sind:

Tiefensuche mit Tiefenbeschränkung k

Wenn die vorgegebene Tiefenschranke k überschritten wird, werden keine Nachfolger dieser Knoten mehr erzeugt. Die Tiefensuche findet in diesem Fall jeden Zielknoten, der höchstens Tiefe k hat.

Eine Haskell-Implementierung dazu wäre (anstelle der Tiefe wird die noch verfügbare Suchtiefe mit den Knoten gespeichert):

```
dfsBisTiefe goal succ stack maxdepth =
  -- wir speichern die Tiefe mit in den Knoten auf dem Stack:
  go   [(k,maxdepth,[k]) | k <- stack]
 where
  go [] = Nothing -- Alles abgesucht, nichts gefunden
  go ((k,i,p):r)
   | goal k = Just (k,p) -- Ziel gefunden
   | i > 0  = go ([(k',i -1, k':p) | k' <- succ k] ++ r)
   | otherwise =  go r -- Tiefenschranke erreicht
```

Tiefensuche mit Sharing

Nutze aus, dass der gerichtete Graph manchmal kein Baum ist. Schon untersuchte Knoten werden nicht nochmal untersucht und redundant weiterverfolgt. Implementierung durch Speichern der bereits besuchten Knoten z.B. in einer Hash-Tabelle. Eine Implementierung in Haskell dazu ist:

```
dfsSharing goal succ stack =
  -- Einfuegen der Anfangspfade, dann iterieren mit go,
  -- letzes Argument ist die Merkliste
  go [(k,[k]) | k <- stack] []
 where
  go [] mem = Nothing -- Alles abgesucht, nichts gefunden
  go ((k,p):r) mem
   | goal k    = Just (k,p) -- Ziel gefunden
   | k `elem` mem = go r mem -- Knoten schon besucht
   | otherwise    = go ([(k',k':p) | k' <- succ k] ++ r) (k:mem)
```

Platzbedarf Anzahl der besuchten Knoten (wegen Speicher für schon untersuchte Knoten)

Zeit: $O(n * \log(n))$ mit $n =$ Anzahl der untersuchten Knoten (n ist exponentiell in der Tiefe bei Verzweigungrate $c > 1$).

Tiefensuche mit Sharing, aber beschränktem Speicher

Diese Variante funktioniert wie Tiefensuche mit Sharing, aber es werden maximal nur B Knoten gespeichert.
Der Vorteil gegenüber Tiefensuche ist Ausnutzen von gemeinsamen Knoten; besser als Tiefensuche mit Sharing, da die Suche weiterläuft, wenn die Speicherkapazität nicht für alle besuchten Knoten ausreicht.

2.1.4.3 Effekt des Zurücksetzen: (Backtracking)

Die Tiefensuche führt Backtracking durch, wenn nicht erfolgreiche Pfade erkannt werden. Die Arbeitsweise der Tiefensuche ist: Wenn Knoten K keine Nachfolger hat (oder eine Tiefenbeschränkung) überschritten wurde, dann mache weiter mit dem nächsten Bruder von K. Dies wird auch als chronologisches Zurücksetzen – chronological backtracking – bezeichnet.

Abwandlungen dieser Verfahren sind *Dynamic Backtracking* und *Dependency-directed Backtracking*. Diese Varianten schneiden Teile des Suchraumes ab und können verwendet werden, wenn mehr über die Struktur des Suchproblems bekannt ist. Insbesondere dann, wenn sichergestellt ist, dass man trotz der Abkürzungen der Suche auf jeden Fall noch einen Zielknoten finden kann.

Es gibt auch Suchprobleme, bei denen kein Backtracking erforderlich ist (greedy Verfahren ist möglich). Dies kann z.B. der Fall sein, wenn jeder Knoten noch einen Zielknoten als Nachfolger hat. Die Tiefensuche reicht dann aus, wenn die Tiefe der Zielknoten in alle Richtungen unterhalb jedes Knotens beschränkt ist. Anderenfalls reicht Tiefensuche nicht aus (ist nicht vollständig), da diese Suche sich immer den Ast aussuchen kann, in dem der Zielknoten weiter weg ist.

Ein vollständiges Verfahren ist die *Breitensuche*:

Algorithmus **Breitensuche**

Datenstrukturen: L sei eine Menge von Knoten, markiert mit dem dorthin führenden Weg.

Eingabe: Füge die initialen Knoten in die Menge$L$ ein.

Algorithmus:

1. Wenn L leer ist, dann breche ab.
2. Wenn L einen Zielknoten K enthält, dann gebe aus: K und Weg dorthin.
3. Sonst sei $N(L)$ Menge aller direkten Nachfolger der Knoten von L, mit einem Weg dorthin markiert.
   Mache weiter mit Schritt 1 und $L := N(L)$.

Eine Implementierung in Haskell ist:

```
bfs goal succ start =
  go [(k,[k]) | k <- start] -- Pfade erzeugen
 where
  go [] = Nothing -- nichts gefunden
  go rs =
   case filter (goal . fst) rs of -- ein Zielknoten enthalten?
        -- Nein, mache weiter mit allen Nachfolgern
    [] -> go [(k',k':p)  | (k,p) <- rs, k' <- succ k]
        -- Ja, dann stoppe:
    (r:rs) -> Just r
```

Die Komplexität der Breitensuche bei fester Verzweigungsrate $c > 1$ ist:

Platz: $\sim$ Anzahl der Knoten in Tiefe d, d.h. c^d, und das ist exponentiell in der Tiefe d.

Zeit: (Anzahl der Knoten in Tiefe d) + Aufwand für Mengenbildung: $n + n * log(n)$, wenn $n = \#Knoten$. Bei obiger Annahme einer fester Verzweigungsrate ist das $c^d(1 + d * log(c))$.

Die Breitensuche ist vollständig! D.h. wenn es einen (erreichbaren) Zielknoten gibt, wird sie auch in endlicher Zeit einen finden.

Fazit der Komplexität; Tiefen- und Breitensuche

$d =$ Tiefe und $c =$ mittlere Verzweigungsrate.

$\approx$	Tiefensuche	Breitensuche
Zeit	$O(c^d)$	$O(c^d(1 + d \log c))$
Platz	$O(c * d)$	$O(c^d)$
Vollständig	nein	ja

2.1.4.4 Iteratives Vertiefen (iterative deepening)

Dieses Verfahren ist ein Kompromiss zwischen Tiefen- und Breitensuche, wobei man die Tiefensuche mit Tiefenbeschränkung anwendet. Hierbei wird die Tiefenschranke iterativ erhöht. Wenn kein Zielknoten gefunden wurde, wird die ganze Tiefensuche jeweils mit größerer Tiefenschranke nochmals durchgeführt.

Eine Implementierung in Haskell ist:

```
idfs goal succ stack =
 let -- alle Ergebnisse mit sukzessiver Erh"ohung der Tiefenschranke
     alleSuchen = [dfsBisTiefe goal succ stack i | i <- [1..]]
 in
   case filter isJust alleSuchen of
     [] -> Nothing -- Trotzdem nichts gefunden
     (r:rs) -> r   -- sonst erstes Ergebnisse
```

Man beachte, dass man die Tiefe, in welcher der erste Zielknoten auftritt, nicht vorhersagen kann.

Der Vorteil des iterativen Vertiefens ist der viel geringere Platzbedarf gegenüber der Breitensuche bei Erhaltung der Vollständigkeit des Verfahrens. Man nimmt eine leichte Verschlechterung des Zeitbedarfs in Kauf, da Knoten mehrfach untersucht werden.

Komplexitätsbetrachtung: bei fester Verzweigungsrate $c > 1$.

Platzbedarf: linear in der Tiefe.

Zeitbedarf: Wir zählen nur die im Mittel besuchten Knoten, wenn der erste Zielknoten in Tiefe k ist, wobei wir bei der Rechnung die Näherung $\sum_{i=1}^{n} a^i \approx \frac{a^{n+1}}{a-1}$ für $a > 1$ verwenden.

Tiefensuche mit Tiefenbeschränkung k:

$$0.5 * (\sum_{i=1}^{k} c^i) \approx 0.5 * (\frac{c^{k+1}}{c-1})$$

Iteratives Vertiefen bis Tiefe k:

$$\begin{aligned}
&\sum_{i=1}^{k-1} \frac{c^{i+1}}{c-1} + 0.5 * (\frac{c^{k+1}}{c-1}) = \frac{\sum_{i=1}^{k-1} c^{i+1}}{c-1} + 0.5 * (\frac{c^{k+1}}{c-1}) \\
= \quad & \frac{(\sum_{i=1}^{k} c^i) - c^k}{c-1} + 0.5 * (\frac{c^{k+1}}{c-1}) \approx \frac{1}{c-1}((\frac{c^{k+1}}{c-1}) - c^1) + 0.5 * (\frac{c^{k+1}}{c-1}) \\
= \quad & (\frac{c^{k+1}}{(c-1)^2}) - \frac{c}{c-1} + 0.5 * (\frac{c^{k+1}}{c-1}) \approx (\frac{c^{k+1}}{(c-1)^2}) + 0.5 * (\frac{c^{k+1}}{c-1})
\end{aligned}$$

Der Faktor des Mehraufwandes für iteratives Vertiefen im Vergleich zur Tiefensuche (mit der richtigen Tiefenbeschränkung) ergibt sich zu:

$$\frac{\frac{c^{k+1}}{(c-1)^2} + 0.5 * (\frac{c^{k+1}}{c-1})}{0.5 * (\frac{c^{k+1}}{c-1})} = \frac{\frac{c^{k+1}}{(c-1)^2}}{0.5 * (\frac{c^{k+1}}{c-1})} + 1 = \frac{2}{c-1} + 1$$

Ein Tabelle der ca.-Werte des Faktors $d = \frac{2}{c-1} + 1$ ist

c	2	3	4	5	...	10
d	3	2	$1,67$	$1,5$	...	$1,22$

D.h. der eigentliche Aufwand ergibt sich am Horizont der Suche. Der Vergleich ist zudem leicht unfair für iteratives Vertiefen, da die Tiefensuche ja den Wert von k nicht kennt.

Das Verfahren des iteratives Vertiefen wird z.B. im Automatischen Beweisen verwendet (siehe z.B. M. Stickel: A PROLOG-technology theorem prover.) Die Entscheidungsalternativen, die man beim Optimieren abwägen muss, sind:

- Speichern
- Neu Berechnen.

Stickels Argument: in exponentiellen Suchräumen greift die Intuition nicht immer: Neuberechnen kann besser sein als Speichern und Wiederverwenden.

Bemerkung 2.1.5. *Beim dynamischen Programmieren ist in einigen Fällen der Effekt gerade umgekehrt. Man spendiert Platz für bereits berechnete Ergebnisse und hat dadurch eine enorme Ersparnis beim Zeitbedarf.*

2.1.4.5 Iteratives Vertiefen (iterative deepening) mit Speicherung

Die Idee hierbei ist, den freien Speicherplatz beim iterativen Vertiefen sinnvoll zu nutzen. Hierbei kann man zwei verschiedene Verfahren verwenden:

1. Bei jedem Restart der Tiefensuche wird der zugehörige Speicher initialisiert. Während der Tiefensuchphase speichert man Knoten, die bereits expandiert waren und vermeidet damit die wiederholte Expansion. Hier kann man soviele Knoten speichern wie möglich. Man spart sich redundante Berechnungen, aber der Nachteil ist, dass bei jeder Knotenexpansion nachgeschaut werden muss, ob die Knoten schon besucht wurden.

2. Man kann Berechnungen, die in einer Tiefensuchphase gemacht wurden, in die nächste Iteration der Tiefensuche retten und somit redundante Berechnungen vermeiden. Wenn alle Knoten der letzten Berechnung gespeichert werden können, kann man von dort aus sofort weiter machen (d.h. man hat dann Breitensuche). Da man das i.a. nicht kann, ist folgendes sinnvoll:
   Man speichert den linken Teil des Suchraum, um den Anfang der nächsten Iteration schneller zu machen, und Knoten, die schon als Fehlschläge bekannt sind, d.h. solche, die garantiert keinen Zielknoten als Nachfolger haben.

2.1.5 Rückwärtssuche

Die Idee der Rückwärtssuche ist:

> man geht von einem (bekannten) Zielknoten aus und versucht den Anfangsknoten zu erreichen.

Voraussetzungen dafür sind:

- Man kann Zielknoten explizit angeben (nicht nur eine Eigenschaft, die der Zielknoten erfüllen muss).
- man kann von jedem Knoten die direkten Vorgänger berechnen.

Rückwärtssuche ist besser als Vorwärtssuche, falls die Verzweigungsrate in Rückwärtsrichtung kleiner ist die Verzweigungsrate in Vorwärtsrichtung.

Man kann normale Suche und Rückwärtssuche kombinieren zur *Bidirektionalen Suche*, die von beiden Seiten sucht.
Dies erfordert Speicherung der aktuellen Knoten und eine Suche nach gemeinsamen Knoten der Vorwärts- und Rückwärtssuche

2.2 Informierte Suche, Heuristische Suche

Die Suche nennt man „informiert“, wenn man zusätzlich eine Bewertung von allen Knoten des Suchraumes angeben kann, d.h. eine *Schätzfunktion*, die man interpretieren kann als Ähnlichkeit zu einem Zielknoten, oder als Schätzung des Abstands zum Zielknoten. Der Zielknoten sollte ein Maximum bzw. Minimum der Bewertungsfunktion sein.

Eine Heuristik („*Daumenregel*“) ist eine Methode, die oft ihren Zweck erreicht, aber nicht mit Sicherheit. Man spricht von heuristischer Suche, wenn die Schätzfunktion in vielen praktisch brauchbaren Fällen die richtige Richtung zu einem Ziel angibt, aber möglicherweise manchmal versagt.

Das Suchproblem kann jetzt ersetzt werden durch:

> Minimierung (bzw. Maximierung) einer Knotenbewertung (einer Funktion) auf einem gerichteten Graphen (der aber erst erzeugt werden muss)

Variante: Maximierung in einer Menge oder in einem n-dimensionaler Raum.

Beispiel: 8-Puzzle

Wir betrachten als Beispiel das sogenannte 8-Puzzle. Dabei sind 8 Plättchen mit den Zahlen 1 bis beschriftet und in einem 3x3 Raster angeordnet, d.h. es gibt ein leeres Feld. Mögliche Züge sind: Plättchen können auf das freie Feld verschoben werden, wenn sie direkter Nachbar des freien Feldes sind. Ziel des Puzzles ist es, die Zahlen der Reihe nach anzuordnen.

8		**1**
6	**5**	**4**
7	**2**	**3**

Wir repräsentieren das 8-Puzzle als als 3×3-Matrix, wobei $a_{ij} = n$, wenn n die Nummer des Plättchens ist, das an Position (i, j) ist. B bezeichne das leere Feld.

Anfang: beliebige Permutation der Matrix, z.B.

$$\begin{pmatrix} 8 & B & 1 \\ 6 & 5 & 4 \\ 7 & 2 & 3 \end{pmatrix}$$

Ziel:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & B \end{pmatrix}$$

Bewertungsfunktionen: zwei Beispiele sind:

1. $f_1()$ Anzahl der Plättchen an der falschen Stelle
2. $f_2()$ Anzahl der Züge (ohne Behinderungen zu beachten), die man braucht, um Endzustand zu erreichen.

$$S_1 = \begin{pmatrix} 2 & 3 & 1 \\ 8 & 7 & 6 \\ B & 5 & 4 \end{pmatrix} \quad \begin{array}{l} f_1(S_1) = 7 \text{ (nur Plättchen 6 ist richtig)} \\ f_2(S_1) = 2+1+1+3+1+0+2+2 = 12 \\ \text{(Die Summe ist in der Reihenfolge } 1, \ldots, n) \end{array}$$

$$S_2 = \begin{pmatrix} 2 & 3 & 1 \\ B & 7 & 6 \\ 8 & 5 & 4 \end{pmatrix} \quad \begin{array}{l} f_1(S_2) = 7 \\ f_2(S_2) = 2+1+1+3+1+0+2+1 = 11 \end{array}$$

Man sieht, dass die zweite Bewertungsfunktion genauer angibt als die erste, wie gut die erreichte Situation ist.

Aber: es ist bekannt, dass es Ausgangssituationen gibt, von denen aus man das Ziel nicht erreichen kann (Permutation ist ungerade).

In den Unterlagen ist ein Haskell-Programm, das mit Best-First und verschiedenen Bewertungsfunktionen, nach nicht allzu langer Zeit eine Zugfolge für 3×3 findet. Wenn man dieses Puzzle für größere n per Hand und per Suche probiert und vergleicht, so erkennt man, dass der Rechner sich in lokalen Optimierungen verliert, während man von Hand eine Strategie findet, die die Zielsituation herstellt, falls es überhaupt möglich ist. Diese Strategie ist in etwa so:

- Verschiebe zunächst die 1 so, dass diese an ihrem Platz ist.
- Verschiebe zunächst die 2 so, dass diese an ihrem Platz ist.
- .....
- bei den letzten beiden Ziffern der ersten Zeile: $n-1, n$ erlaube eine Suche durch Verschieben innerhalb eines 3×2-Kästchens.
- Benutze diese Strategie bis zur vorletzten Zeile.
- Bei der vorletzten Zeile, fange von links an, jeweils zwei Plättchen korrekt zu schieben, bis das letzte 2×3 Rechteck bleibt.
- Innerhalb dieses Rechtecks probiere, bis die Plättchen an ihrem Platz sind.

Man sieht, dass einfache Bewertungsfunktionen diese Strategie nicht erzeugen. Kennt man die Strategie, kann man eine (komplizierte) Bewertungsfunktion angeben, die dieses Verhalten erzwingt. Beim n-Puzzle ist es vermutlich schwieriger diese Bewertungsfunktion zu finden, als das Problem direkt zu programmieren, wenn man eine Strategie erzwingen will. Hier braucht man eigentlich einen guten Planungsalgorithmus mit Zwischen- und Unterzielen und Prioritäten.

2.2.1 Bergsteigerprozedur (Hill-climbing)

Dies ist auch als Gradientenaufstieg (-abstieg)[2] bekannt, wenn man eine n-dimensionale Funktion in die reellen Zahlen maximieren will (siehe auch Verfahren im Bereich Operations Research).

Parameter der Bergsteigerprozedur sind

- Menge der initialen Knoten
- Nachfolgerfunktion (Nachbarschaftsrelation)

[2] Gradient: Richtung der Vergrößerung einer Funktion; kann berechnet werden durch Differenzieren

- Bewertungsfunktion der Knoten, wobei wir annehmen, dass Zielknoten maximale Werte haben (Minimierung erfolgt analog)

- Zieltest

Algorithmus **Bergsteigen**

Datenstrukturen: L : Liste von Knoten, markiert mit Weg dorthin
h sei die Bewertungsfunktion der Knoten
Eingabe: L sei die Liste der initialen Knoten, absteigend sortiert entsprechend h

Algorithmus:

1. Sei K das erste Element von L und R die Restliste
2. Wenn K ein Zielknoten, dann stoppe und geben K markiert mit dem Weg zurück
3. Sortiere die Liste $NF(K)$ absteigend entsprechend h und entferne schon besuchte Knoten aus dem Ergebnis. Sei dies die Liste L'.
4. Setze $L := L'$++R und gehe zu 1.

Bergsteigen verhält sich analog zur Tiefensuche, wobei jedoch stets als nächster Knoten der Nachfolger expandiert wird, der heuristisch den besten Wert besitzt. Bei *lokalen Maxima* kann die Bergsteigerprozedur eine zeitlang in diesen verharren, bevor Backtracking durchgeführt wird. Beachte, dass das Vermeiden von Zyklen unabdingbar ist, da ansonsten die Bergsteigerprozedur in lokalen Maxima hängen bleibt. Der Erfolg der Bergsteigerprozedur hängt stark von der Güte der Bewertungsfunktion ab. Ein anderes Problem ergibt sich bei *Plateaus*, d.h. wenn mehrere Nachfolger die gleiche Bewertung haben, in diesem Fall ist nicht eindeutig, welcher Weg gewählt werden soll.

Durch die Speicherung der besuchten Knoten ist der Platzbedarf linear in der Anzahl der besuchten Knoten, also exponentiell in der Tiefe.

In Haskell könnte man die Bergsteigerprozedur implementieren durch folgenden Code:

```
hillclimbing cmp heuristic goal successor start =
 let -- sortiere die Startknoten
   list = map (\k -> (k,[k])) (sortByHeuristic start)
 in go list []
  where
   go ((k,path):r) mem
    | goal k    = Just (k,path) -- Zielknoten erreicht
    | otherwise =
       let -- Berechne die Nachfolger (nur neue Knoten)
           nf = (successor k) \\ mem
           -- Sortiere die Nachfolger entsprechend der Heuristik
           l' = map (\k -> (k,k:path)) (sortByHeuristic nf)
       in go  (l' ++ r) (k:mem)
   sortByHeuristic = sortBy (\a b  -> cmp (heuristic a) (heuristic b))
```

Dabei ist das erste Argument `cmp` die Vergleichsfunktion (z.B. `compare`), die es daher flexibel ermöglicht zu minimieren oder zu maximieren. Das zweite Argument ist die Heuristik als Funktion von Zuständen nach Werten, das dritte Argument ist der Zieltest, das vierte die Nachfolgerfunktion und `start` ist eine Liste von initialen Zuständen.

2.2.2 Der Beste-zuerst (Best-first) Suche

Wir betrachten als nächste die Best-first-Suche. Diese wählt immer als nächsten zu expandierenden Knoten, denjenigen Knoten aus, der den *besten Wert* bzgl. der Heuristik hat. Der Algorithmus ist:

Algorithmus **Best-First Search**

Datenstrukturen:
Sei L Liste von Knoten, markiert mit dem Weg dorthin.
h sei die Bewertungsfunktion der Knoten

Eingabe: L Liste der initialen Knoten, sortiert, so dass die besseren Knoten vorne sind.

Algorithmus:

1. Wenn L leer ist, dann breche ab
2. Sei K der erste Knoten von L und R die Restliste.
3. Wenn K ein Zielknoten ist, dann gebe K und den Weg dahin aus.
4. Sei $N(K)$ die Liste der Nachfolger von K. Entferne aus $N(K)$ die bereits im Weg besuchten Knoten mit Ergebnis $\mathcal{N}$
5. Setze $L := \mathcal{N}$ ++ R
6. Sortiere L, so dass bessere Knoten vorne sind und gehe zu 1.

Die best-first Suche entspricht einer Tiefensuche, die jedoch den nächsten zu expandierenden Knoten anhand der Heuristik sucht. Im Unterschied zum Bergsteigen, bewertet die Best-First-Suche stets alle Knoten im Stack und kann daher schneller zurücksetzen und daher lokale Maxima schneller wieder verlassen. In der angegebenen Variante ist die Suche nicht vollständig (analog wie die Tiefensuche). Der Platzbedarf ist durch die Speicherung der besuchten Knoten exponentiell in der Tiefe.

Der Vollständigkeit halber geben wir eine Implementierung in Haskell an:

```
bestFirstSearchMitSharing cmp heuristic goal successor start =
 let -- sortiere die Startknoten
     list = sortByHeuristic (map (\k -> (k,[k])) (start))
 in go list []
  where
   go ((k,path):r) mem
    | goal k    = Just (k,path) -- Zielknoten erreicht
    | otherwise =
       let -- Berechne die Nachfolger und nehme nur neue Knoten
           nf = (successor k) \\ mem
           -- aktualisiere Pfade
           l' = map (\k -> (k,k:path)) nf
           -- Sortiere alle Knoten nach der Heuristik
           l'' = sortByHeuristic (l' ++ r)
       in go l'' (k:mem)
   sortByHeuristic =
    sortBy (\(a,_) (b,_)-> cmp (heuristic a) (heuristic b))
```

2.2.3 Simuliertes Ausglühen (simulated annealing)

Die Analogie zum Ausglühen ist:
Am Anfang hat man hohe Energie und alles ist beweglich,
dann langsame Abkühlung, bis die Kristallisation eintritt (bei minimaler Energie).

Die Anwendung in einem Suchverfahren ist sinnvoll für Optimierung von n-dimensionalen Funktionen mit reellen Argumenten, weniger bei einer Suche nach einem Zielknoten in einem gerichteten Suchgraphen.
Bei Optimierung von n-dimensionalen Funktionen kann man die Argumente verändern, wobei man den Abstand der verschiedenen Argumenttupel abhängig von der Temperatur definieren kann. Bei geringerer Temperatur sinkt dieser Abstand. Der Effekt ist, dass man zunächst mit einem etwas groben Raster versucht, Anhaltspunkte für die Nähe zu einem Optimum zu finden. Hat man solche Anhaltspunkte, verfeinert man das Raster, um einem echten Maximum näher zu kommen, und um nicht über den Berg drüber zu springen bzw. diesen zu untertunneln.

Implementierung in einem Optimierungsverfahren:

1. Am Anfang große Schrittweite (passend zum Raum der Argumente). Mit diesen Werten und der Schrittweite Suche analog zu Best-First; dann allmähliche Verminderung der Schrittweite, hierbei Schrittweite und Richtung zufällig auswählen, bis sich das System stabilisiert. Die besten Knoten werden jeweils weiter expandiert.

2. Alternativ: Das System läuft getaktet. Am Anfang wird eine zufällige Menge von Knoten berücksichtigt. Bessere Knoten dürfen mit größerer Wahrscheinlichkeit im nächsten Takt Nachfolger erzeugen als schlechte Knoten. Die Wahrscheinlichkeit für schlechte Knoten wird allmählich gesenkt, ebenso die Schrittweite.

Analoge Verfahren werden auch in künstlichen neuronalen Netzen beim Lernen verwendet. Variante 2 hat Analogien zum Vorgehen bei evolutionären Algorithmen. Ein Vorteil des Verfahrens liegt darin, dass lokale Maxima können verlassen werden. Ein Probleme des Verfahrens ist, dass es einige Parameter zu justieren sind, die stark vom Problem abhängen können:

- Schnelligkeit der „Abkühlung“
- Abhängigkeit der Wahrscheinlichkeit von der „Temperatur“ (bzw. Wert der Schätzfunktion)

2.3 A^*-Algorithmus

Die vom A^*-Algorithmus zu lösenden Suchprobleme bestehen aus den Eingaben:

- ein (gerichteter) Graph (wobei die Nachfolgerbeziehung i.a. algorithmisch gegeben ist), d.h. es gibt eine Nachfolgerfunktion NF, die zu jedem Knoten N die Nachfolger NF(N) berechnet. Wir nehmen an, dass der Graph schlicht ist, d.h. es gibt höchstens eine Kante zwischen zwei Knoten.

- eine (i.a. reellwertige) Kostenfunktion c für Kanten, sodass $c(N_1, N_2) \in \mathbb{R}$ gerade die Kosten der Kante von Knoten N_1 zu Knoten N_2 festlegt.
  Für einen Weg $N_1 \to N_2 \to \ldots \to N_k$ bezeichne $c_W(N_1 N_2 \ldots N_k)$ gerade die Summe der Kosten $\sum_{i=1}^{k-1} c(N_i, N_{i+1})$.

- ein Startknoten S

- eine Schätzfunktion $h(\cdot)$, die die Kosten von einem bereits erreichten Knoten N bis zum nächsten Zielknoten abschätzt.

- einen Test auf Zielknoten Z. Die Zielknoten sind meist algorithmisch definiert, d.h. man hat nur einen Test, ob ein gegebener Knoten ein Zielknoten ist.

Ziel ist das Auffinden eines optimalen (d.h. mit minimalen Kosten) Weges vom Startknoten zu einem der Zielknoten,

Beispiel 2.3.1. *Ein typisches Beispiel ist die Suche nach einem kürzesten Weg von A nach B in einem Stadtplan. Hier sind die Knoten die Kreuzungen und die Kanten die Straßenabschnitte zwischen den Kreuzungen; die Kosten entsprechen der Weglänge.*

Der A^*-Algorithmus stützt sich auf eine Kombination der bereits verbrauchten Kosten $g(N)$ vom Start bis zu einem aktuellen Knoten N und der noch geschätzten Kosten $h(N)$ bis zu einem Zielknoten Z, d.h. man benutzt die Funktion $f(N) = g(N) + h(N)$. Die Funktion $h(\cdot)$ sollte leicht zu berechnen sein, und muss nicht exakt sein.

Es gibt zwei Varianten des A^*-Algorithmus (abhängig von den Eigenschaften von h).

- Baum-Such-Verfahren: Kein Update des minimalen Weges bis zu einem Knoten während des Ablaufs.

- Graph-Such-Verfahren: Update des minimalen Weges bis zu einem Knoten während des Ablaufs

Im Folgenden ist der A^*-Algorithmus als Graph-Such-Verfahren angegeben.

Algorithmus **A^*-Algorithmus**

Datenstrukturen:

- Mengen von Knoten: `Open` und `Closed`
- Wert $g(N)$ für jeden Knoten (markiert mit Pfad vom Start zu N)
- Heuristik h, Zieltest Z und Kantenkostenfunktion c

Eingabe:

- `Open` $:= \{S\}$, wenn S der Startknoten ist
- $g(S) := 0$, ansonsten ist g nicht initialisiert
- `Closed` $:= \emptyset$

Algorithmus:

repeat
  Wähle N aus `Open` mit minimalem $f(N) = g(N) + h(N)$
  if $Z(N)$ **then**
    break; // Schleife beenden
  else
    Berechne Liste der Nachfolger $\mathcal{N} := NF(N)$
    Schiebe Knoten N von `Open` nach `Closed`
    for $N' \in \mathcal{N}$ **do**
      if $N' \in$ `Open` $\cup$ `Closed` und $g(N) + c(N, N') > g(N')$ **then**
        skip // Knoten nicht verändern
      else
        $g(N') := g(N) + c(N, N')$; // neuer Minimalwert für g(N')
        Füge N' in `Open` ein und (falls vorhanden) lösche N' aus `Closed`;
      end-if
    end-for
  end-if
until `Open` $= \emptyset$
if `Open` $= \emptyset$ **then** Fehler, kein Zielknoten gefunden
else N ist der Zielknoten mit $g(N)$ als minimalen Kosten
end-if

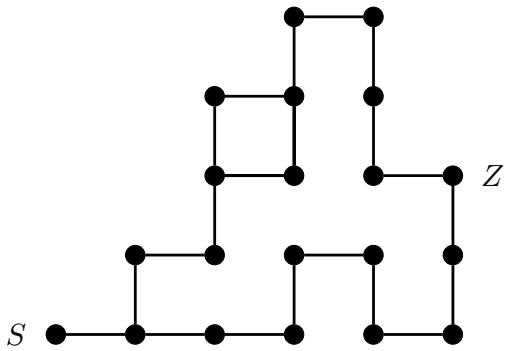
Bei der Variante als *Baumsuche* gibt es keine `Closed`-Menge. Daher kann der gleiche Knoten mehrfach in der Menge `Open` mit verschiedenen Wegen stehen. Bei Graphsuche hat man immer den aktuell bekannten minimalen Weg, und somit den Knoten nur einmal in `Open`.

Beachte, dass der (bzw. ein) Zielknoten in der `Open`-Menge sein kann, dieser aber erst als Ziel erkannt wird, wenn er als zu expandierender Knoten ausgewählt wird. Ebenso ist zu beachten, dass ein Knoten in der `Closed`-Menge sein kann, und wieder in die `Open`-Menge eingefügt wird, weil ein kürzerer Weg entdeckt wird.

Eine rekursive Implementierung der A^*-Suche in Haskell ist:

```
-- Eintr"age in open / closed: (Knoten, (g(Knoten), Pfad zum Knoten))
aStern heuristic goal successor open closed
  | null open = Nothing -- Kein Ziel gefunden
  | otherwise =
     let n@(node,(g_node,path_node)) = Knoten mit min. f-Wert
                 minimumBy (\(a,(b,_)) (a',(b',_))
                    -> compare  ((heuristic a) + b) ((heuristic a') + b')) open
     in
      if goal node then Just n else  -- Zielknoten expandiert
       let nf = (successor node)   -- Nachfolger
           -- aktualisiere open und closed:
           (open',closed')  = update nf (delete n open) (n:closed)
           update [] o c = (o,c)
           update ((nfnode,c_node_nfnode):xs) o c =
            let (o',c') = update xs o c -- rekursiver Aufruf
                -- m"oglicher neuer Knoten, mit neuem g-Wert und Pfad
                newnode = (nfnode,(g_node + c_node_nfnode,path_node ++ [node]))
            in  case lookup nfnode open of -- Knoten in Open?
                 Nothing -> case lookup nfnode closed of -- Knoten in Closed?
                             Nothing -> (newnode:o',c')
                             Just (curr_g,curr_path) ->
                              if curr_g > g_node + c_node_nfnode
                               then (newnode:o',delete (nfnode,(curr_g,curr_path)) c')
                               else (o',c')
                 Just (curr_g,curr_path) ->
                  if curr_g > g_node + c_node_nfnode
                   then (newnode:(delete (nfnode,(curr_g,curr_path)) o'),c')
                   else  (o',c')
       in aStern heuristic goal successor open' closed'
```

Beispiel 2.3.2. *Im Beispiel kann man sich den Fortschritt des A^*-Algorithmus leicht klarmachen, wenn man eine Kante als 1 zählt, weiß wo das Ziel ist, und als Schätzfunktion des Abstands die Rechteck-Metrik $|y_2 - y_1| + |x_2 - x_1|$ des Abstands vom aktuellen Knoten zum Ziel nimmt.*



Rechnet man das Beispiel durch, dann sieht man, dass der A^*-Algorithmus sowohl den oberen als auch den unteren Weg versucht.

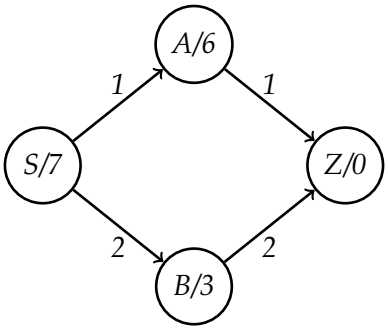
2.3.1 Eigenschaften des A^*-Algorithmus

Im folgenden einige Bezeichnungen, die wir zur Klärung der Eigenschaften benötigen.

$g^*(N, N')$	=	Kosten des optimalen Weges von N nach N'
$g^*(N)$	=	Kosten des optimalen Weges vom Start bis zu N
$c^*(N)$	=	Kosten des optimalen Weges von N bis zum nächsten Zielknoten Z.
$f^*(N)$	=	$g^*(N) + c^*(N)$
		(Kosten des optimalen Weges durch N bis zu einem Zielknoten Z)

Der A^*-Algorithmus hat gute Eigenschaften, wenn die Schätzfunktion h die Kosten unterschätzt. D.h., wenn für jeden Knoten N: $h(N) \leq c^*(N)$ gilt.

Beispiel 2.3.3. *Wir betrachten das folgende Beispiel, mit einer* überschätzenden *Schätzfunktion:*



Die Notation dabei ist $N/h(N)$, d.h. in jedem Knoten ist die heuristische Bewertung angegeben. Führt man den A^*-Algorithmus für dieses Beispiel mit Startknoten S und Zielknoten Z aus, erhält man den Ablauf:

Am Anfang*:*

`Open` : $\{S\}$

`Closed` : $\{\emptyset\}$

Nach Expansion von S*:*

`Open` : $\{A, B\}$ *mit* $g(A) = 1$ *und* $g(B) = 2$

`Closed` : $\{S\}$

Nächster Schritt*:*

Da $f(A) = g(A) + h(a) = 1 + 6 = 7$ *und* $f(B) = g(B) + h(B) = 2 + 3 = 5$*, wird* B *als nächster Knoten expandiert und man erhält:*

`Open` : $\{Z, A\}$ *mit* $g(Z) = 4$

`Closed` : $\{B, S\}$

Nächster Schritt*:*

Da $f(A) = g(A) + h(A) = 1 + 6 = 7$ *und* $f(Z) = g(Z) + h(Z) = 4 + 0 = 0$*, wird der Zielknoten* Z *expandiert und der A^*-Algorithmus gibt den Weg* $S \to B \to Z$ *mit Kosten* 4 *als optimalen Weg aus, obwohl ein kürzerer Weg* $S \to A \to Z$ *existiert.*

Beispiel 2.3.4. *Die Länge jedes Weges von A nach B in einem Stadtplan ist stets länger als die direkte Entfernung (Luftlinie)* $\sqrt{(b_2 - a_2)^2 + (b_1 - a_1)^2}$*. Diese einfach zu berechnende Schätzfunktion unterschätzt die echten Kosten.*

Wenn alle Straßen rechtwinklig aufeinander zulaufen, und es keine Abkürzungen gibt, kann man auch die Rechtecknorm als Schätzung nehmen: $|b_2 - a_2| + |b_1 - a_1|$, denn die echten Wege können dann nicht kürzer als die Rechtecknorm sein (auch Manhattan-Abstand genannt).

Definition 2.3.5. *Voraussetzungen für den A^*-Algorithmus:*

1. *es gibt nur endlich viele Knoten N mit $g^*(N) + h(N) \leq d$, wobei $d = inf\{$Kosten aller Wege von S zu einem Zielknoten $Z\}$.*

2. *Für jeden Knoten N gilt: $h(N) \leq c^*(N)$, d.h. die Schätzfunktion ist unterschätzend.*

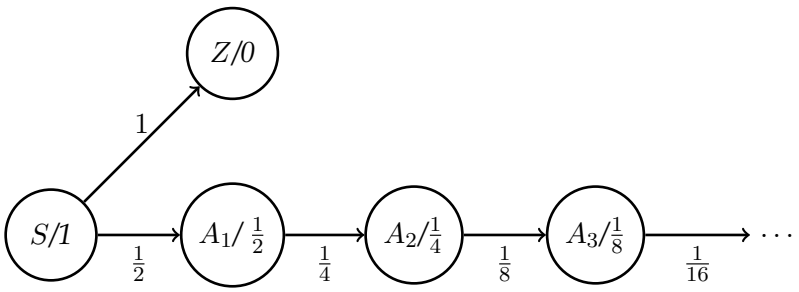
3. *Für jeden Knoten N ist die Anzahl der Nachfolger endlich.*

4. *Alle Kanten kosten etwas: $c(N, N') > 0$ für alle N, N'.*

5. *Der Graph ist schlicht*[3]*.*

Die Voraussetzung (1) der Endlichkeit der Knoten und (4) sind z.B. erfüllt, wenn es eine untere Schranke $\delta > 0$ für die Kosten einer Kante gibt.

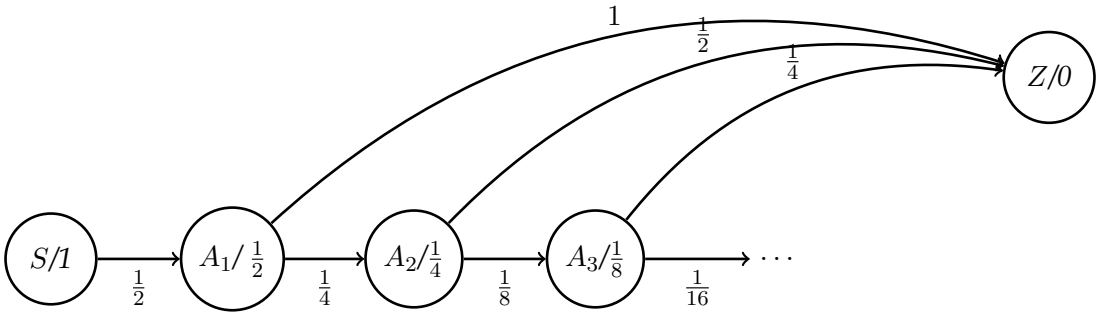
Beispiel 2.3.6. *Betrachte den folgenden Suchgraph, der Bedingung 1 aus Definition 2.3.5 verletzt:*



Dabei gehen wir davon aus, dass der Pfad $A_1 \to A_2 \to A_3 \to \ldots$ entsprechend unendlich weiter verläuft, mit $c(A_i, A_{i+1}) = \frac{1}{2^{i+1}}$ und $h(A_i) = \frac{1}{2^i}$. In diesem Fall gibt es nur den direkten Pfad $S \to Z$ als mögliche und optimale Lösung, insbesondere ist $d = 1$. Der A^*-Algorithmus startet mit S und fügt die Knoten A_1 und Z in die `Open`*-Menge ein. Anschließend wird stets für $i = 1, \ldots$ A_i expandiert und A_{i+1} in die* `Open`*-Menge eingefügt. Da Z nie expandiert wird, terminiert der A^*-Algorithmus nicht.*

Beispiel 2.3.7. *Das folgende Beispiel ist ähnlich zum vorhergehenden, es verletzt ebenfalls Bedingung 1 aus Definition 2.3.5. Es zeigt jedoch, dass es zum Infimum d nicht notwendigerweise auch einen endlichen Weg im Suchgraphen geben muss:*

[3] schlicht = zwischen zwei Knoten gibt es höchstens eine Kante



Der Pfad $A_1 \to A_2 \to A_3 \to \ldots$ laufe entsprechend unendlich weiter, wobei $c(A_i, Z) = \frac{1}{2^{i-1}}$, $c(A_i, A_{i+1}) = \frac{1}{2^{i+1}}$ und $h(A_i) = \frac{1}{2^i}$. In diesem Fall gibt es unendlich viele Wege von S nach Z, der Form $S \to A_1 \to \ldots A_k \to Z$ für $k = 1, 2, \ldots$. Die Kosten der Pfade sind $1\frac{1}{2}, 1\frac{1}{4}, 1\frac{1}{8}, \ldots$. Das Beispiel zeigt, daher das $d = 1$ als Infimum definiert werden muss, und dass es keinen Weg von S zu Z mit Kosten 1 gibt, man aber für jedes $\epsilon > 0$ einen Weg findet, dessen Kosten kleiner $1 + \epsilon$ sind. Der A-Algorithmus würde für obigen Graphen, nur die A_i-Knoten expandieren, aber nie den Zielknoten Z expandieren.*

Wir zeigen nun zunächst, dass die Bedingungen aus Definition 2.3.5 ausreichen, um zu folgern, dass zum Infimum d auch stets ein endlicher Weg mit Kosten d existiert.

Für einen Knoten N sei $infWeg(N) := inf\{$Kosten aller Wege von N zu einem Zielknoten$\}$

Satz 2.3.8. *Es existiere ein Weg vom Start S bis zu einem Zielknoten. Sei $d = infWeg(S) = inf\{$Kosten aller Wege von S zu einem Zielknoten $Z\}$. Die Voraussetzungen in Definition 2.3.5 seien erfüllt.*
Dann existiert ein optimaler Weg von S zum Ziel mit Kosten d.

Beweis. Wir zeigen mit Induktion, dass wir für jedes $n \geq 1$ einen der folgenden beiden Wege konstruieren können:

1. $S = K_1 \to \ldots \to K_i$ mit $i \leq n$, $c_W(K_1, \ldots, K_i) = d$ und K_i ist ein Zielknoten
2. $S = K_1 \to \ldots \to K_n$, wobei $c_W(K_1, \ldots K_j) + infWeg(K_j) = d$ für alle $j \in \{1, \ldots, n\}$

Basisfall $n = 1$. Dann erfüllt der „Weg“ S die Voraussetzung, da $infWeg(S) = d$ angenommen wurde. Für den Induktionsschritt, sei die Bedingung für $n-1$ erfüllt. Wir betrachten zwei Fälle:

- Die Induktionsannahme erfüllt Bedingung 1, d.h. es gibt einen Weg $K_1 \to \ldots K_i$ (für ein $i \leq n-1$) mit $c_W(K_1, \ldots, K_i) = d$ und K_i ist Zielknoten. Dann erfüllt der gleiche Weg Bedingung 1 für n.
- Die Induktionsannahme erfüllt Bedingung 2. Sei $K_1 \to \ldots \to K_{n-1}$ der entsprechende Weg. Insbesondere gilt $c_W(K_1, \ldots, K_{n-1}) + infWeg(K_{n-1}) = d$.

Wenn K_{n-1} ein Zielknoten ist, ist Bedingung 1 erfüllt und wir sind fertig. Anderenfalls betrachte alle Nachfolger $N_1, \ldots, N_m$ von K_j. Da es nur endliche viele Nachfolger gibt, gibt es mindestens einen Nachfolger N_h, so dass $c_W(K_1, \ldots, K_{n-1}) + c(K_j, N_h) + infWeg(N_h) = d$. Mit $K_n := N_h$ erfüllt der Weg $K_1 \to \ldots \to K_n$ Bedingung 2.

Zum Beweis der eigentlichen Aussage, müssen wir jetzt nur noch zeigen, dass es ein n gibt, dass Bedingung 1 erfüllt, d.h. es gibt keinen unendlichen langen Weg der Bedingung 2 erfüllt.

Betrachte die konstruierte Folge $K_1 \to K_2 \to \ldots$ gem. Bedingung 2. Ein Knoten kann nicht mehrfach in der Folge $K_0, K_1, \ldots$ vorkommen, da man sonst das Infimum d echt verkleinern könnte (den Zyklus immer wieder gehen). Weiterhin muss für jeden Knoten K_j gelten, dass $h(K_j) \leq infWeg(K_j)$ ist, da h unterschätzend ist und $infWeg(K_j)$ das entsprechende Infimum der existierenden Wege zum Ziel ist. Gäbe es unendlich viele Knoten K_j, dann wäre daher Bedingung 1 aus Definition 2.3.5 verletzt. Somit folgt, dass es keine unendliche Folge gibt und man daher stets einen endlichen optimalen Weg findet. □

Lemma 2.3.9. *Es existiere ein optimaler Weg $S = K_0 \to \ldots \to K_n = Z$ vom Startknoten S bis zu einem Zielknoten Z . Die Voraussetzungen in Definition 2.3.5 seien erfüllt. Dann ist während der Ausführung des A^*-Algorithmus stets ein Knoten K_i in* `Open`*, markiert mit $g(K_i) = g^*(K_i)$, d.h. mit einem optimalen Weg von S zu K_i.*

Beweis. Induktion über die Iterationen des A^*-Algorithmus. Für den Basisfall, genügt es zu beachten, dass $S \in$ `Open` und $g(S) = 0$ gilt, und im Anschluss K_1 in `Open` eingefügt wird. Da $S \to K_1$ und K_1 auf einem optimalen Weg liegt, muss K_1 mit $g(K_1) = c(S, K_1) = g^*(S, K_1)$ markiert werden (sonst gäbe es einen kürzeren Weg von S zu Z).

Induktionsschritt: Als Induktionshypothese verwenden wir, dass `Open` mindestens einen der Knoten K_i enthält, der mit $g(K_i) = g^*(K_i)$ markiert ist. Sei j maximal, so dass K_j diese Bedingungen erfüllt. Wir betrachten unterschiedliche Fälle:

- Ein Knoten $\neq K_j$ wird expandiert. Dann verbleibt K_j in `Open` und der Wert $g(K_j)$ wird nicht verändert, da er bereits optimal ist.

- Der Knoten K_j wird expandiert. Dann wird K_{j+1} in `Open` eingefügt und erhält den Wert $g(K_{j+1}) = g(K_j) + c(K_j, K_{j+1}) = g^*(K_j) + c(K_j, K_{j+1})$ Dieser Wert muss optimal sein, da ansonsten der Weg $S \to K_1 \to \ldots \to K_n$ nicht optimal wäre.

□

Lemma 2.3.10. *Unter der Annahme, dass die Bedingungen aus 2.3.5 erfüllt sind, gilt: Wenn A^* einen Zielknoten expandiert, dann ist dieser optimal.*

Beweis. Nehme an $S = K_0 \to K_1 \dots K_n = Z$ ist ein optimaler Weg. Angenommen, ein Zielknoten Z' wird expandiert. Dann ist Z' der Knoten aus `Open` mit dem minimalen $f(Z') = g(Z') + h(Z') = g(Z')$. Betrachte den Knoten K_j mit maximalem j aus dem optimalen Weg, der noch in `Open` ist und mit $g(K_j) = g^*(K_j)$ markiert ist (gem. Lemma 2.3.9). Wenn $Z = Z'$ und genau auf dem Weg erreicht wurde, dann ist genau der optimale Weg $S \to K_1 \to \dots \to Z$ gefunden worden. Wenn $K_j \neq Z'$, dann gilt $f(K_j) = g(K_j) + h(K_j) \leq d$ da h die Kosten bis zum Ziel unterschätzt. Da Z' expandiert wurde, gilt $f(Z') = g(Z') + h(Z') = g(Z') \leq g(K_j) + h(K_j) \leq d$, und daher ist der gefundene Weg optimal. □

Lemma 2.3.11. *Unter der Annahme, dass die Bedingungen aus 2.3.5 erfüllt sind, und ein Weg vom Start zum Zielknoten existiert gilt: Der A*-Algorithmus expandiert einen Zielknoten nach endlich vielen Schritten.*

Beweis. Seien d die Kosten des optimalen Wegs $S = K_0 \to K_1 \dots \to K_n = Z$. Aufgrund von Lemma 2.3.9 genügt es zu zeigen, dass jeder der Knoten K_i nach endlich vielen Schritten expandiert wird, oder ein anderer Zielknoten wird expandiert. Da stets ein K_i in `Open` mit Wert $g(K_i) = g^*(K_i)$ markiert ist und $g^*(K_i) + h(K_i) \leq d$ gelten muss (die Heuristik ist unterschätzend), werden nur Knoten L expandiert für die zum Zeitpunkt ihrer Expansion gilt: $g(L) + h(L) \leq g^*(K_i) + h(K_i) \leq d$. Da $g^*(L) + h(L) \leq g(L) + h(L)$, gilt auch $g^*(L) + h(L) \leq d$ für all diese Knoten L. Gemäß Bedingung 1 aus Definition 2.3.5 kann es daher nur endlich viele solcher Knoten L geben, die vor K_i expandiert werden.

Jetzt ist noch zu zeigen, dass kein Knoten unendlich oft expandiert wird:
Die Anzahl der Wege zwischen den Knoten L mit $g^*(L) + h(L) \leq d$ ist endlich, also auch die möglichen Werte $g^*(L)$. Damit kann jeder Knoten nur endlich oft wird nach `Open` geschoben werden, also auch nur endlich oft expandiert werden.

□

Aus den Lemmas 2.3.10 und 2.3.11 und Satz 2.3.8 folgt direkt:

Theorem 2.3.12. *Es existiere ein Weg vom Start bis zu einem Zielknoten. Die Voraussetzungen in Definition 2.3.5 seien erfüllt. Dann findet der A*-Algorithmus einen optimalen Weg zu einem Zielknoten.*

Beachte, dass der A^*-Algorithmus auch einen anderen optimalen Weg finden kann, der von $K_1, \dots, K_n$ verschieden ist.

Beispiel 2.3.13. *Für das Stadtplan-Beispiel können wir folgern, dass der A*-Algorithmus im Falle des Abstandsmaßes $h(.)$, das die Luftlinienentfernung zum Ziel misst, einen optimalen Weg finden wird. Hat der Stadtplan nur Straßen in zwei zueinander senkrechten Richtungen, dann kann man auch die Rechtecknorm nehmen.*

Der Aufwand des A^*-Algorithmus ist i.a. exponentiell in der Länge des optimalen Weges (unter der Annahme einer konstanten Verzweigungsrate).

Bemerkung 2.3.14. *Wenn man den endlichen Graphen als gegeben annimmt, mit Bewertungen der Kanten, dann kann man mit dynamischem Programmieren in polynomieller Zeit (in der Größe des Graphen) alle kostenoptimalen Wege berechnen, indem man analog zur Berechnung der transitiven Hülle eine Matrix von Zwischenergebnissen iteriert (Floyd-Algorithmus für kürzeste Wege, Dijkstra-Algorithmus).*

Der Dijkstra-Algorithmus entspricht dem A^*-Algorithmus, wenn man die Schätzfunktion $h(N) = 0$ setzt für alle Knoten N. Dieser hat als Aufgabe, einen Weg mit minimalen Kosten zwischen zwei Knoten eines gegebenen endlichen Graphen zu finden.
In diesem Fall hat man stets eine Front (`Open`) der Knoten mit minimalen Wegen ab dem Startknoten.

2.3.2 Spezialisierungen des A^*-Algorithmus:

Beispiel 2.3.15. *Weitere Beispiele für Graphen und Kostenfunktionen sind:*

Wenn $h(N) = 0$, dann ist der A^*-Algorithmus dasselbe wie die sogenannte Gleiche-Kostensuche (branch-and-bound)
Wenn $c(N_1, N_2) = k$ (k Konstante, z.B. 1), und $h(N) = 0$, dann entspricht der A^*-Algorithmus der Breitensuche.

Die Variante A^{*o} des A^*-Algorithmus, die alle optimalen Weg finden soll, hat folgende Ergänzung im Algorithmus:

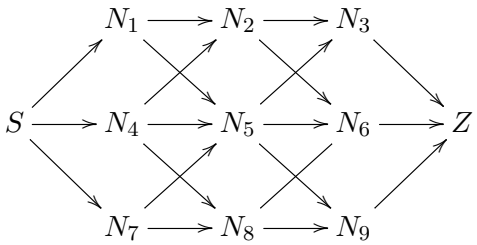
- sobald ein Weg zum Ziel gefunden wurde, mit einem Wert d, werden nur noch Knoten in `Open` akzeptiert (d.h. die anderen kommen nach `Closed`), bei denen $g(N) + h(N) \leq d$.
- Man markiert den besuchten Knoten mit allen letzten Knoten der besten Wege; (die dann alle gleichen Wert haben müssen.)
- Knoten kommen auch wieder nach `Open`, wenn ein zusätzlicher Weg dorthin gefunden wurde mit gleichen Werten; Bei Update durch einen besseren Knoten: nur diesen letzten Knoten speichern.

Der Algorithmus stoppt erst, wenn keine Knoten mehr in `Open` sind.

Die optimalen Wege sind dann (kompakt und implizit) im gerichteten Graphen gespeichert.

Beispiel 2.3.16. *Beispiel zur Anzahl aller optimalen Wege.*

Es gibt den Startknoten S, den Zielknoten Z und mehrere Knoten N_i. Alle Kanten haben gleiche Kosten 1.



Man sieht dass es expontiell viele optimalen Wege von S nach Z gibt, wen man das Beispiel zu einer Beispielserie verallgemeinert.

Theorem 2.3.17. *Es gelten die Voraussetzung von Satz 2.3.12. Dann findet der Algorithmus A^{*o} alle optimalen Wege von S nach Z.*

Beweis. Zunächst findet der A^*-Algorithmus einen optimalen Weg zu einem Ziel. Daraus ergibt sich eine obere Schranke d für die Kosten eines optimalen Weges. Aus den Voraussetzungen ergibt sich, dass der A^{*o}-Algorithmus nur endlich viele Knoten und Wege inspiziert. □

Definition 2.3.18. *Wenn man zwei Schätzfunktionen h_1 und h_2 hat mit:*

1. *h_1 und h_2 unterschätzen den Aufwand zum Ziel*
2. *für alle Knoten N gilt: $h_1(N) \leq h_2(N) \leq c^*(N)$*

dann nennt man h_2 besser informiert *als h_1.*

Hieraus alleine kann man noch nicht folgern, dass der A^*-Algorithmus zu h_2 sich besser verhält als zu h_1. (Übungsaufgabe)

Notwendig ist: Die Abweichung bei Sortierung der Knoten mittels f muss klein sein. D.h. optimal wäre $f(k) \leq f(k') \Leftrightarrow f^*(k) \leq f^*(k')$.

Der wichtigere Begriff ist:

Definition 2.3.19. *Eine Schätzfunktion $h(.)$ ist* monoton, *gdw.*
$h(N) \leq c(N, N') + h(N')$ für alle Knoten N und deren Nachfolger N' und wenn $h(Z) = 0$ ist für Zielknoten Z.

Offenbar gilt die Monotonie-Ungleichung auch für die optimale Weglänge von N nach N', wenn N' kein direkter Nachfolger von N ist. Dies kann man so sehen:
$h(N) \leq c(N, N_1) + h(N_1) \leq c(N, N_1) + c(N_1, N_2) + h(N_2)$
Iteriert man das, erhält man: $h(N) \leq c_W(N, N') + h(N')$
Und damit auch $h(N) \leq g^*(N, N') + h(N')$
Für den Weg zum Ziel Z gilt damit $h(N) \leq g^*(N, Z) + h(Z) = g^*(N, Z)$, da $h(Z) = 0$ ist.

Lemma 2.3.20. *Eine monotone Schätzfunktion $h(.)$ ist unterschätzend.*

Die Monotonie der Schätzfunktion entspricht der Dreiecksungleichung in Geometrien und metrischen Räumen.
Die Monotonieeigenschaft beschreibt das Verhalten der Funktion f beim Expandieren: Wenn die Schätzfunktion monoton ist, dann wächst der Wert von f monoton, wenn man einen Weg weiter expandiert.

Satz 2.3.21. *Ist die Schätzfunktion h monoton, so expandiert der A^*-Algorithmus jeden untersuchten Knoten beim ersten mal bereits mit dem optimalen Wert. D.h. $g(N) = g^*(N)$ für alle expandierten Knoten.*

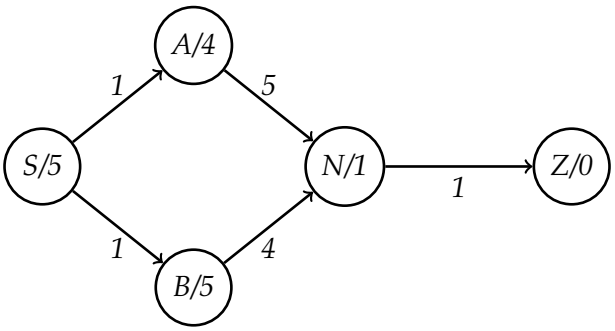
Beweis. Sei $S = K_1 \to \ldots \to K_n$ ein optimaler Weg von S nach K_n. Wir nehmen an die Aussage sei falsch und es gelte $g(K_n) > g^*(K_n)$ und K_n wird expandiert. Aus Lemma 2.3.9 folgt: Es gibt einen Knoten $K_i \neq K_n$ und $(i < n)$, der in `Open` ist und für den gilt $g(K_i) = g^*(K_i)$.

$$\begin{aligned} f(K_i) &= g^*(K_i) + h(K_i) \\ &\leq g^*(K_i) + g^*(K_i, K_N) + h(K_N) \quad \text{folgt aus Monotonie} \\ &= g^*(K_N) + h(K_N) \\ &< g(K_N) + h(K_N) = f(K_N). \end{aligned}$$

Da $f(K_i) < f(K_N)$ müsste aber K_i anstelle von K_n expandiert werden. D.h. wir haben einen Widerspruch gefunden. □

Diese Aussage erlaubt es bei monotonen Schätzfunktionen, den A^*-Algorithmus zu vereinfachen: man braucht kein Update der optimalen Weglänge in `Closed` durchzuführen. D.h. die Baumsuchvariante des A^*-Algorithmus reicht aus.

Beispiel 2.3.22. *Die Aussage bedeutet nicht, dass nicht schon Knoten des optimalen Weges vorher in `Open` sind. Diese könnten auf einem nichtoptimalen Weg erreicht werden, werden dann aber nicht expandiert. Das folgende Beispiel zeigt dies.*



Der A^*-Algorithmus expandiert zunächst S, dann A. Bei der Expansion von A wird N in `Open` *eingefügt (mit dem bis dahin gefundenen Weg $S \to A \to N$), aber der optimale Weg von S zu*

N ist noch nicht entdeckt. Im nächsten Schritt wird B expandiert, was zur Entdeckung des Weges $S \to B \to N$ führt. Erst jetzt wird der Knoten N expandiert.

Theorem 2.3.23. *Ist die Schätzfunktion h monoton, so findet der A^*-Algorithmus auch in der Baum-Variante (ohne update) den optimalen Weg zum Ziel.*

Bemerkung 2.3.24. *Der Dijkstra-Algorithmus zum Finden optimaler Wege in einem Graphen entspricht (wenn man von der Speicherung des optimalen Weges absieht) folgender Variante:*
Man wählt $h() = 0$ und nimmt den A^*-Algorithmus.
Dann ist $h()$ monoton und es gelten die Aussagen für monotone Schätzfunktionen.

Es gelten folgende weitere Tatsachen für monotone Schätzfunktionen:

Satz 2.3.25. *Unter den Voraussetzungen von Aussage 2.3.21 (d.h. Monotonie von h) gilt:*

1. *Wenn N später als M expandiert wurde, dann gilt $f(N) \geq f(M)$.*
2. *Wenn N expandiert wurde, dann gilt $g^*(N) + h(N) \leq d$ wobei d der optimale Wert ist.*
3. *Jeder Knoten mit $g^*(N) + h(N) \leq d$ wird von A^{*o} expandiert.*

Theorem 2.3.26. *Wenn eine monotone Schätzfunktion gegeben ist, die Schätzfunktion in konstanter Zeit berechnet werden kann, dann läuft der A^*-Algorithmus in Zeit $O(|D|)$, wenn $D = \{N \mid g^*(N) + h(N) \leq d\}$.*
Nimmt man die Verzweigungsrate c als konstant an, d als Wert des optimalen Weges und δ als der kleinste Abstand zweier Knoten, dann ist die Komplexität $O(c^{\frac{d}{\delta}})$.

In einem rechtwinkligen Stadtplan, in dem die Abstände zwischen zwei Kreuzungen alle gleich sind, gilt, dass der A^*-Algorithmus in quadratischer Zeit abhängig von der Weglänge des optimalen Weges einen Weg findet. Das liegt daran, dass es nur quadratisch viele Knoten in allen Richtungen gibt. Im allgemeinen kann die Suche exponentiell dauern (z.B. mehrdimensionale Räume)

Beispiel 2.3.27. *Die Suche eines kürzesten Weges im Stadtplan mit dem Luftlinienabstand als Schätzfunktion ist monoton: Mit $ll()$ bezeichnen wir den Luftlinienabstand. Es gilt wegen der Dreiecksungleichung: $ll(N, Z) < ll(N, N') + ll(N', Z)$. Da der echte Abstand stets größer als Luftlinie ist, gilt: $ll(N, N') \leq c(N, N')$, und mit der Bezeichnung $h(\cdot) := ll(\cdot)$ gilt $h(N) \leq c(N, N') + h(N')$.*

Dasselbe gilt für das Beispiel des gitterförmigen Stadtplans, bei dem man die Rechtecknorm nimmt, da auch diese die Dreiecksungleichung erfüllt.

Man sieht auch, dass in einem quadratischen Stadtplan die Rechtecknorm besser informiert ist als die Luftlinienentfernung.

Beispiel 2.3.28. *Wir betrachten die komplexere Situation, dass ein (ebenes) Verkehrswegenetz gegeben ist, z.B. mit Bahn und Flugzeug. Wir vereinfachen, und nehmen an dass es zwei Arten von Kanten gibt: schnelle Kanten, die mit dem Flugzeug zurückgelegt werden, und mittelschnelle,*

auf denen eine Bahn führt. Geschwindigkeit der Flugzeugs ist v_F, die der Bahn v_B. Gesucht ist bei Eingabe von zwei Knoten $A.B$ die schnellste Verbindung.
Gibt es dazu eine gute monotone Schätzfunktion h?.

- *Eine erste einfache Schätzfunktion ist $h_1(A,B) = d(A,B)/v_F$. D.h. nehme die Luftlinie direkt mit dem Flugzeug. Behauptung: h_1 ist monoton. Klar ist, dass diese unterschätzend ist.*
  Sie ist auch monoton: Der Weg von N über N_1 nach Z ist wg der Dreiecksungleichung länger die Luftlinie von N nach Z, also $ll(N,Z) \leq ll(N,N') + ll(N',Z)$ und da $ll(N,N') \leq c(N,N')$ gilt auch $ll(N,Z) \leq c(N,N') + ll(N',Z)$. Rechnet man die Zeit, dann gilt auch $ll(N,Z)/v_F \leq c(N,N')/v_F + ll(N',Z)/v_F \leq c(N,N')/v_B + ll(N',Z)/v_F$.

- *Eine weitere Schätzfunktion $h_2(N,Z)$ ist folgende: Man sucht die nächsten Flughäfen N_F zu N und Z_F zu Z. Dann nimmt man an, dass die Flughäfen auf der Luftlinie liegen:*
  1. *Wenn $ll(N,N_F)+ll(Z,Z_F) \geq ll(N,Z)$, dann $h_2(N,Z) = ll(N,Z)(v_B)$. D.h. Bahnfahren ist schneller.*
  2. *Wenn $ll(N,N_F) + ll(Z,Z_F) < ll(N,Z)$, dann $h_2(N,Z) = (ll(N,N_F) + ll(Z,Z_F))/v_B + (ll(N,Z) - (ll(N,N_F) + ll(Z,Z_F)))/v_F$.*

  Behauptung: diese Schätzfunktion ist monoton. Sei N' der nächste Knoten. Dazu sei N'_F der nächste Flughafen. Wenn N selbst schon der FLughafen ist, dann ist die Rechnung einfach, also nehmen wir an, dass N kein Flughafen ist.
  Es gilt, dass $ll(N,N_F) \leq ll(N,N') + ll(N',N'_F)$ da N_F der nächste Flughafen zu N ist. Bei gleichem Abstand N,N' ist $h_2(N',Z)$ minimal, wenn N' auf der Luftlinie N,Z liegt. Dann ist auf jeden Fall die Flugstrecke die in $h_2(N,Z)$ eingeht, länger als die die in $h_2(N',Z)$ eingeht. Es reicht somit aus die Rechnung so zu machen, als sei N' und N'_F auf der Luftlinie N,Z, und erhält nach kurzer Rechnung die Abschätzung $h_2(N,Z) \leq c(N,N') + h_2(N',Z)$, also die Monotonie.

Das Fazit ist, dass die Suche nach dem optimalen (d.h. schnellsten) Weg mit dieser Schätzfunktion einfach ist.

Satz 2.3.29. *es gelten die Voraussetzungen in Satz 2.3.12. Seien d die Kosten des optimalen Weges. Seien h_1, h_2 zwei monotone Schätzfunktionen, so dass h_2 besser informiert sei als h_1.*

Sei A_1 der A^{*o}-Algorithmus zu h_1 und A_2 sei der A^{*o}-Algorithmus zu h_2.
Dann gilt: Alle Knoten N mit $g^*(N) + h_2(N) \leq d$ die von A_2 expandiert werden, werden auch von A_1 expandiert.

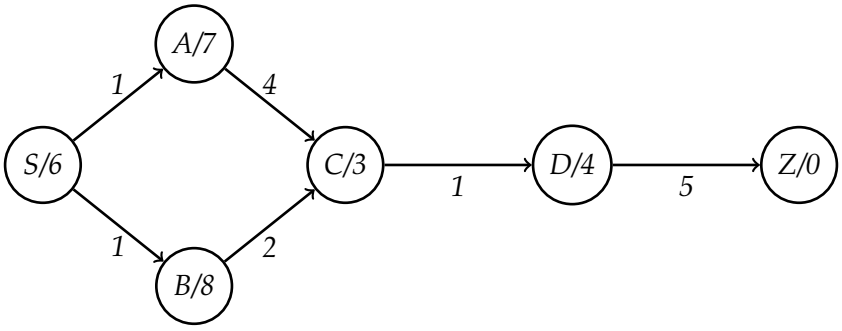
Beweis. Der A^{*o}-Algorithmus stoppt erst, wenn er keine Knoten mehr findet, die unter der Grenze sind. Ein Knoten N wird von A_1 expandiert, wenn es einen Weg von S nach N gibt, so dass für alle Knoten M auf dem Weg die Ungleichung $g^*(M) + h_2(M) \leq d$ gilt. Da wegen $h_1(M) \leq h_2(M)$ dann $g^*(M) + h_1(M) \leq d$ gilt, wird der Knoten auch von A_1 expandiert

□

D.h. es lohnt sich, bessere monotone Schätzfunktionen zu konstruieren. Allerdings sollte man diese nicht wieder mit einer aufwändigen Suche bestimmen, da dann der Vorteil wieder verlorengeht.

Was macht man, wenn die Schätzfunktion nicht monoton ist? Im Fall einer unterschätzenden Funktion h gibt es eine Methode der Korrektur während des Ablaufs des Algorithmus.
Angenommen, N hat Nachfolger N' und es gilt $h(N) > c(N, N') + h(N')$. Nach Voraussetzung ist der optimale Wert eines Weges von N zum Zielknoten größer als $h(N)$. Wenn es einen Weg von N' zu einem Ziel gäbe der besser als $h(N) - c(N, N')$ ist, dann hätte man einen Widerspruch zur Unterschätzung. Damit kann man $h'(N') := h(N) - c(N, N')$ definieren. Danach gilt $h(N) = c(N, N') + h'(N')$.

Beispiel 2.3.30. *Es kann vorkommen, dass ein Knoten im normalen A*-Algorithmus von* `Open` *nach* `Closed` *geschoben wird, aber danach nochmal nach* `Open` *bewegt wird, da man doch einen besseren Weg gefunden hat. Betrachte den Suchgraph:*



Die Expansionsreihenfolge ist:

expandiert	*Nachfolger*	`Open`	`Closed`
S	*A,B*	*S*	
A	*C*	*A,B*	*S*
C	*D*	*B,C*	*S,A*
B	*C*	*B,D*	*S,A,C*
C	*D*	*C,D*	*S,A,B*
...			

Der Knoten C wird daher von `Closed` *erneut nach* `Open` *geschoben. Diese Schätzfunktion kann nicht monoton sein. Es gilt* $8 = h(B) > c(B, C) + h(C) = 5$.

Was dieses Beispiel auch noch zeigt, ist dass der Reparaturmechanismus, der dynamisch h zu einer monotonen Schätzfunktion verbessert, das Verhalten des Algorithmus nicht so verbessert, dass Aussage 2.3.21 gilt, denn die Aussage gilt nicht für dieses Beispiel unter der Annahme der nachträglichen Korrektur von h.

2.3.3 IDA*-Algorithmus und SMA*-Algorithmus

Da der A*-Algorithmus sich alle jemals besuchten Knoten merkt, wird er oft an die Speichergrenze stoßen und aufgeben. Um das zu umgehen, nimmt man eine Kombination des abgeschwächten A^*-Algorithmus und des iterativen Vertiefens. Die Rolle der Grenze spielt in diesem Fall der maximale Wert d eines Weges, der schon erzeugt wurde.

IDA* mit Grenze d:

- Ist analog zu A^*.
- es gibt keine `Open/Closed`-Listen, nur einen Stack mit Knoten und Wegekosten.
- der Wert $g(N)$ wird bei gerichteten Graphen nicht per Update verbessert.
- Der Knoten N wird nicht expandiert, wenn $f(N) > d$.
- das Minimum der Werte $f(N)$ mit $f(N) > d$ wird das d in der nächsten Iteration.

Dieser Algorithmus benötigt nur linearen Platz in der Länge des optimalen Weges, da er nur einen Stack (Rekursions-Stack des aktuellen Weges) verwalten muss.

Da dieser Algorithmus in einem gerichteten Graphen möglicherweise eine exponentielle Anzahl von Wegen absuchen muss, obwohl das beim A^*-Algorithmus durch Update zu vermeiden ist, gibt es eine pragmatische Variante, den SMA*-Algorithmus, der wie der A^*-Algorithmus arbeitet, aber im Fall, dass der Speicher nicht reicht, bereits gespeicherte Knoten löscht. Der Algorithmus nutzt hierzu die verfügbare Information und löscht die Knoten, der am ehesten nicht zu einem optimalen Weg beiträgt. Das sind Knoten, die hohe f-Werte haben. Es gibt weitere Optimierungen hierzu (siehe Russel-Norvig).

2.4 Suche in Spielbäumen

Ziel dieses Kapitels ist die Untersuchung von algorithmischen Methoden zum intelligenten Beherrschen von strategischen *Zweipersonenspielen*, wie z.B. Schach, Dame, Mühle, Go, Tictactoe, bei dem der Zufall keine Rolle spielt (siehe auch 2 Artikel aus (Wegener, 1996)).

Spiele mit mehr als zwei Personen sind auch interessant, aber erfordern andere Methoden und werden hier nicht behandelt.

Zum Schachspiel existierten bereits vor Benutzung von Computern Schachtheorien, Untersuchungen zu Gewinnstrategien in bestimmten Spielsituationen (Eröffnungen, Endspiel, usw) und zum Bewerten von Spielsituationen. Mittlerweile gibt es zahlreiche Varianten von recht passabel spielenden Schachprogrammen, und auch einige sehr spielstarke Programme.

Es gibt zwei verschiedene Ansätze, ein Programm zum Schachspielen zu konzipieren:

1. Man baut auf den bekannten Schachtheorien und Begriffen auf, wie Verteidigung, Angriff, Bedrohung, Decken, Mittelfeld, ..., und versucht die menschliche Methode zu programmieren.

2. Man versucht mit reiner Absuche aller Möglichkeiten unter Ausnutzung der Geschwindigkeit eines Computers, einen guten nächsten Zug zu finden.

Betrachtet man heutige, erfolgreiche Schachprogramme, dann ist die wesentliche Methode das Absuchen der Varianten, wobei auf einer (einfachen) statischen Bewertung von Spielsituationen aufgebaut wird. Ergänzend dazu wird eine Eröffnungsbibliothek verwendet und Endspielvarianten werden getrennt programmiert. Die Programmierung der meisten Schachtheorien scheitert daran, dass sie nicht auf Computer übertragbar sind, da die Begriffe unscharf definiert sind.

Wir betrachten zunächst sogenannte Spielbäume, die ein Zweipersonenspiel repräsentieren:

Definition 2.4.1 (Spielbaum)**.** *Es gibt zwei Spieler, die gegeneinander spielen. Diese werden aus bald ersichtlichen Gründen* Maximierer *und* Minimierer *genannt. Wir nehmen an, es gibt eine* Darstellung *einer* Spielsituation *(z.B. Schachbrett mit Aufstellung der Figuren), und die Angabe* welcher der beiden Spieler am Zug ist. *Der Spielbaum dazu ist wie folgt definiert:*

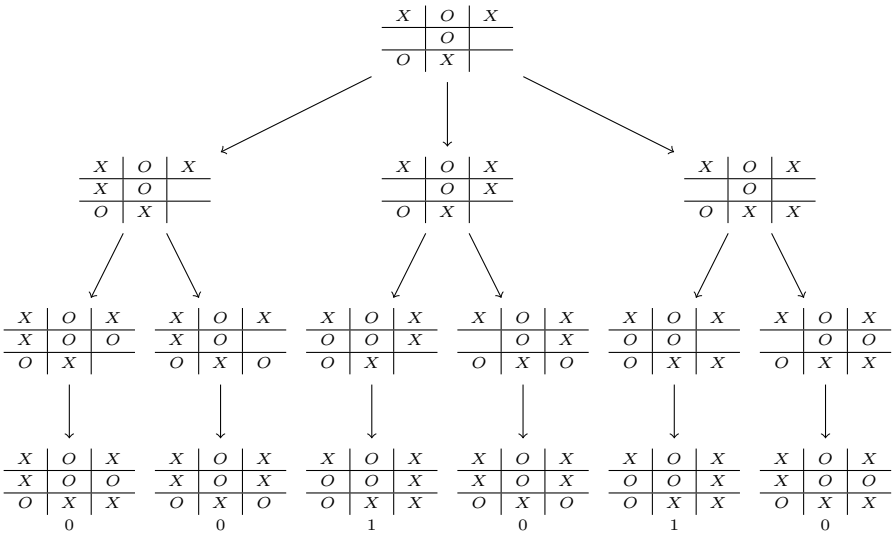
- *Die Wurzel ist mit der aktuellen Spielsituation markiert.*
- *Für jeden Knoten sind dessen Kinder genau die Spielsituationen die durch den nächsten Zug des aktuellen Spielers möglich sind. Pro Ebene wechselt der Spieler*
- *Blätter sind Knoten, die keine Nachfolger mehr haben, d.h. Endsituation des Spiels*
- *Blätter werden mit einem* Gewinnwert *bewertet. Normalerweise ist die eine ganze Zahl. Bei Schach z.B.: 1, wenn der Maximierer gewonnen hat, -1, wenn der Minimierer gewonnen hat, 0 bei Remis. Bei anderen Spielen kann dies auch eine Punktzahl o.ä. sein.*

Als Beispiel betrachten wir TicTacToe: In diesem Spiel wird eine Anfangs leere (3x3)-Matrix abwechselnd durch die beiden Spieler mit X und O gefüllt. Ein Spieler gewinnt, wenn er drei seiner Symbole in einer Reihe hat (vertikal, horizontal oder Diagonal). Wir nehmen an, dass Maximierer X als Symbol benutzt und Minimierer O.

Betrachte z.B. den Spielbaum zur Situation

X	O	X
	O	
O	X	

wobei der Maximierer am Zug ist.



Der Spielbaum zeigt schon, dass der Maximierer keinesfalls verlieren wird, aber kann er sicher gewinnen? Und wenn ja, welchen Zug sollte er als nächsten durchführen?

Das Ziel der Suche besteht darin, den optimalen nächsten Zug zu finden, d.h. das Finden einer Gewinnstrategie. Dabei soll zu jeder Spielsituation der optimale Zug gefunden werden. Als Sprechweise kann man verwenden, dass man einen einzelnen Zug Halbzug nennt; ein Zug ist dann eine Folge von zwei Halbzügen.

Für die betrachteten Spiele ist die Menge der Spielsituationen endlich, also kann man im Prinzip auch herausfinden, welche Spielsituationen bei optimaler Spielweise zum Verlust, Gewinn bzw. Remis führen.

Wenn man Spiele hat, die zyklische Zugfolgen erlauben, bzw. wie im Schach bei dreifach sich wiederholender Stellung (bei gleichem Spieler am Zug) die Stellung als Remis werten, dann ist die folgende Minimax-Prozedur noch etwas zu erweitern.

Die Suche im Spielbaum der Abspielvarianten mit einer Bewertungsfunktion wurde schon von den Pionieren der Informatik vorgeschlagen (von Shannon, Turing; siehe auch die Übersicht in Russel und Norvig). Dies nennt man die **Minimax-Methode**.

Algorithmus **Minimax-Suche**

Datenstrukturen: Nachfolgerfunktion, Wert(Zustand) für Endsituationen, Datenstruktur für Spielzustand, Spieler

Funktion Minimax(Zustand,Spieler):
$NF :=$ alle Nachfolgezustände zu (Zustand,Spieler);
`if` $NF = \emptyset$ `then return` Wert(Zustand) `else`
  `if` Spieler == Maximierer `then`
    `return` $max\{\text{Minmax}(Z, \overline{\text{Spieler}}) \mid Z \in NF\}$
  `else` // Spieler ist Minimierer
    `return` $min\{\text{Minmax}(Z, \overline{\text{Spieler}}) \mid Z \in NF\}$
  `endif`
`endif`
Wobei $\overline{\text{Spieler}}$ der jeweils anderen Spieler bedeuten soll.

In einer echten Implementierung, sollte man zusätzlich den gegangenen Weg speichern, damit der nächste optimale Zug ebenfalls als Ausgabe zur Verfügung steht.

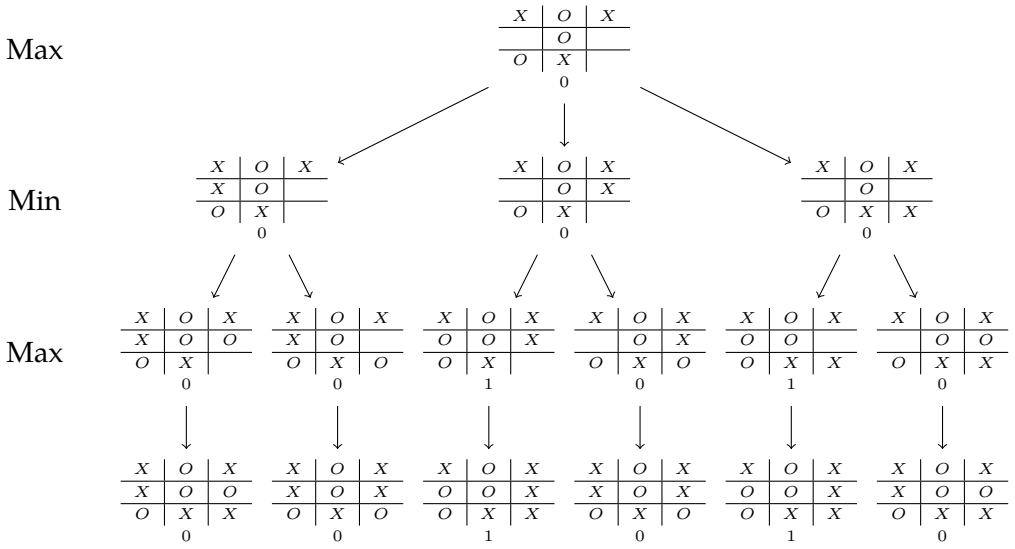
Im Spielbaum entspricht die Minimax-Suche gerade dem Verfahren:

- Bewerte alle Blätter

- Berechne den Wert der anderen Knoten Stufenweise von unten nach oben, wobei über die Werte der Kinder maximiert bzw. minimiert wird, je nachdem welcher Spieler am Zug ist.

Beachte, dass die Implementierung des Minimax-Algorithmus jedoch als *Tiefensuche* erfolgen kann, und daher die ebenenweise Bewertung (Minimierung / Maximierung) nach und nach vornimmt.

Wir betrachten erneut das TicTacToe-Beispiel und markieren alle Knoten entsprechend:



D.h. der Minimax-Algorithmus wir 0 als Ergebnis liefern, d.h. der Maximierer wird nicht gewinnen, wenn sich beide Spieler optimal verhalten, da der Minimierer stets eine Möglichkeit hat, das Ergebnis auf ein Unentschieden zu lenken.

Eine Implementierung des Minimax-Verfahrens in Haskell (mit Merken der Pfade) ist:

```
minmax endZustand wert nachfolger spieler zustand  =
  go (zustand,[]) spieler
  where
   go (zustand,p) spieler
     | endZustand zustand = (wert zustand,reverse $ zustand:p)
     | otherwise =
        let l = nachfolger spieler zustand
        in case spieler of
             Maximierer ->  maximumBy
                              (\(a,_) (b,_) -> compare a b)
                              [go (z,zustand:p) Minimierer | z <- l]
             Minimierer ->  minimumBy
                              (\(a,_) (b,_) -> compare a b)
                              [go (z,zustand:p) Maximierer | z <- l]
```

Das *praktische Problem* ist jedoch, dass ein vollständiges Ausprobieren aller Züge und die Berechnung des Minimax-wertes i.A. nicht machbar ist. Selbst bei TicTacToe gibt es am Anfang schon: 9! = 362880 mögliche Zugfolgen, was auf einem Computer gerade noch machbar ist. Wir werden später im Verlauf des Kapitels noch die Alpha-Beta-Suche kennenlernen, die es ermöglicht etwas besser zu suchen, indem einzelne Teilbäume nicht betrachtet werden müssen. Allerdings kann führt auch diese Methode nicht dazu, dass man bei anspruchsvollen Spielen (wie z.B. Schach etc.) bis zur Endsituation suchen kann.

Daher wendet man eine Abwandlung der Minimax-Suche (bzw. später auch Alpha-Beta-Suche) an: Man sucht nur bis zu einer festen Tiefe k, d.h. man schneidet den Spiel-

baum ab einer bestimmten Tiefe ab. Nun bewertet man die Blattknoten (die i.A. keine Endsituationen sind!) mithilfe einer *heuristische Bewertungsfunktion* (Heuristik). Anschließend verfährt man wie bei der Minimax-Methode und berechnet die Minima bzw. Maxima der Kinder, um den aktuellen Wert des Knotens zu ermitteln. D.h. man minimiert über die Situationen nach Zug des Gegners und maximiert über die eigenen Zugmöglichkeiten. Die Zugmöglichkeit, mit dem optimalen Wert ist dann der berechnete Zug.

Eine solche heuristische Bewertung von Spielsituationen muss algorithmisch berechenbar sein, und deren Güte bestimmt entsprechend auch die Güte des Minimax-Verfahrens.

Beispiele für die Bewertungsfunktion sind:

Schach: Materialvorteil, evtl. Stellungsvorteil, Gewinnsituation, ...

Mühle: Material, #Mühlen, Bewegungsfreiheit, …

Tictactoe: am einfachsten: Gewinn = 1, Verlust = -1, Remis = 0
nur eindeutige Spielsituationen werden direkt bewertet.
Rein theoretisch reicht diese Bewertung bei allen diesen Spielen aus, wenn man bis zum Ende suchen könnte.

Beachte, dass in Spielen ohne Zufall gilt: Die Berechnung des besten Zuges ist unabhängig von den exakten Werten der Bewertungsfunktion; es kommt nur auf die erzeugte *Ordnung* zwischen den Spielsituationen an. Z.B. ist bei TicTacToe egal, ob wir mit Sieg, Niederlage, Unentschieden mit 1, -1, 0 bewerten oder 100, 10, 50. Allerdings ändert sich die Suche, wenn wir die Ordnung ändern und bspw. Sieg, Niederlage, Unentschieden mit 1,-1,1 bewerten (Minimax unterscheidet dann nicht zwischen Sieg und Unentschieden).

Beispiel 2.4.2. *Heuristische Bewertung für Tictactoe. Eine komplizierte Bewertung ist:*

*(#einfach X-besetzte Zeilen/Spalten/Diag) * 1*
\+ *(# doppelt X-besetzte Zeilen/Spalten/Diag) * 5*
\+ *(20, falls Gewinnsituation)*
\- *(#einfach O-besetzte Zeilen/Spalten/Diag) * 1*
\- *(# doppelt O-besetzte Zeilen/Spalten/Diag) * 5*
\- *(20, falls Verlustsituation)*

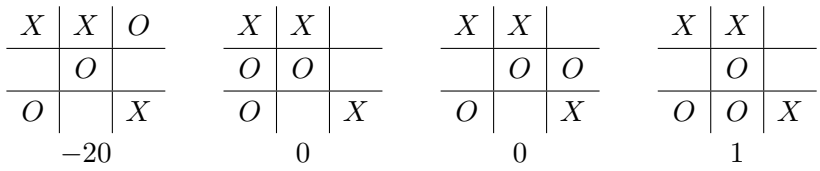
Die aktuelle Spielsituation sei

X		
	O	
O		X

. Die nächsten Spielsituationen zusammen mit den Bewertungen, wenn X-Spieler dran ist, sind:

X	X		X		X	X			X			X		
	O			O		X	O			O	X		O	
O		X	O		X	O		X	O		X	O	X	X
0			8			−3			1			−3		

Die Weiterführung der ersten Situation nach möglichen Zügen von O-Spieler ergibt:



Über diese ist zu minimieren, so dass sich hier ein Wert von -20 *ergibt.*
In der ersten Ebene, nach der ersten Zugmöglichkeit von X, ist zu maximieren. Der erste Zug trägt -20 *dazu bei.*

Auf der Webseite zu dieser Veranstaltung ist ein Haskell-Programm, das zur einfachen Bewertung $+1, 0, -1$ *eine Berechnung des nächsten Gewinnzuges bzw. falls es keinen solchen gibt, den Zug ausrechnet, der zum Remis führt, oder aufgibt. Dieses Programm findet in annehmbarer Zeit jeweils den besten Zug, und erkennt auch, dass es keinen garantierten Gewinn für den X-Spieler oder O-Spieler am Anfang gibt. Z.B.*

```
*Main> nzug 'X'  "XBBBBOBBB"
"Siegzug Feld: 3"
*Main> nzug 'X'  "XBBBBBOBB"
"Siegzug Feld: 2"
*Main> nzug 'X'  "XBBBBBBOB"
"Siegzug Feld: 3"
*Main> nzug 'X'  "XBBBBBBBO"
```

Man könnte die Bewertung verbessern, wenn man beachtet, wer am Zug ist, allerdings ergibt sich der gleiche Effekt, wenn man einen Zug weiter sucht und bewertet.

Praktisch erwünschte Eigenschaften der Bewertungsfunktion sind:

- hoher Wert $\equiv$ hohe Wahrscheinlichkeit zu gewinnen
- schnell zu berechnen

Dies verdeckt manchmal den Weg zum Sieg. z.B. im Schach: Ein Damenopfer, das zum Gewinn führt, würde zwischenzeitlich die Bewertung bzgl. Material verringern.

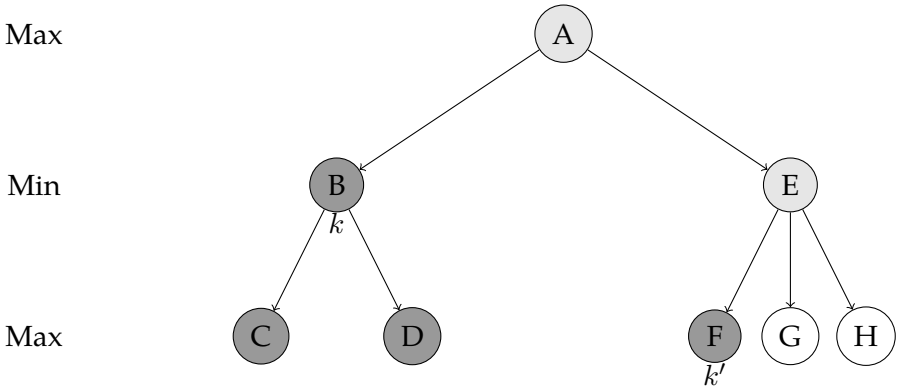
Beachte, dass der folgende Einwand nicht ganz aus der Luft gegriffen ist: Wenn man ein sehr gute Bewertungsfunktion hat: Warum sollte man überhaupt mit Minimax in die Tiefe schauen, und nicht direkt die Nachfolger der Wurzel bewerten, und den am besten bewerteten Zug übernehmen? Das Gegenargument ist, dass i.A. die Bewertung besser wird, je weiter man sich dem Ziel nähert. Betrachtet man z.B. Schach, dann kann man aus der Bewertung des Anfangszustand und dessen direkte Nachfolger vermutlich nicht viel schließen.

Eine Tiefenbeschränkte Min-Max-Suche braucht $O(c^m)$ Zeit, wenn c die mittlere Verzweigungrate, m die Tiefenschranke ist und die Bewertungsfunktion konstante Laufzeit hat

2.4.1 Alpha-Beta Suche

Betrachtet man den Ablauf der Minimax-Suche bei Verwendung der (i.A. tiefenbeschränkten) Tiefensuche, so stellt man fest, dass man manche Teilbäume eigentlich nicht betrachten muss.

Betrachte den folgenden Spielbaum



Dann ergibt sich der Wert für die Wurzel aus

$$
\begin{aligned}
& \text{MinMax}(A, Max) \\
= \; & \texttt{max}\{\text{MinMax}(B, Min), \text{MinMax}(E, Min)\} \\
= \; & \texttt{max}\{\texttt{min}\{\text{MinMax}(C, Max), \text{MinMax}(D, Max)\}, \\
& \quad \texttt{min}\{\text{MinMax}(F, Max), \text{MinMax}(G, Max), \text{MinMax}(H, Max)\}\}
\end{aligned}
$$

Nehmen wir an, die Tiefensuche, hat bereits die Werte für die Knoten B und F als k und k' berechnet (die Schattierungen der Knoten sollen dies gerade verdeutlichen: ganz dunkle Knoten sind fertig bearbeitet durch die Tiefensuche, hellgraue Knoten noch in Bearbeitung, und weiße Knoten noch unbesucht).

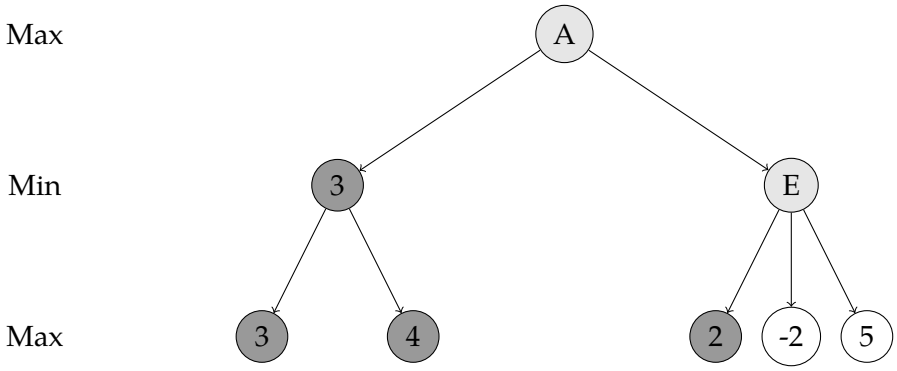
Dann ergibt sich:

$$\text{MinMax}(A, Max) = \texttt{max}\{k, min\{k', \text{MinMax}(G, Max), \text{MinMax}(H, Max)\}\}$$

Wenn $k' \leq k$, dann „weiß“ der Minimierer, dass die komplette Berechnung von $\texttt{min}\{k', \text{MinMax}(G, Max), \text{MinMax}(H, Max)\}$ (also dem Wert des Knotens E), dem Maximierer bei der Berechnung des Wertes für A irrelevant sein wird, da der Maximierer von E sicher einen Wert kleiner gleich k' (und kleiner gleich k) geliefert bekommt, und daher der Wert k für die Knoten B und E der bessere (maximale) Wert ist.

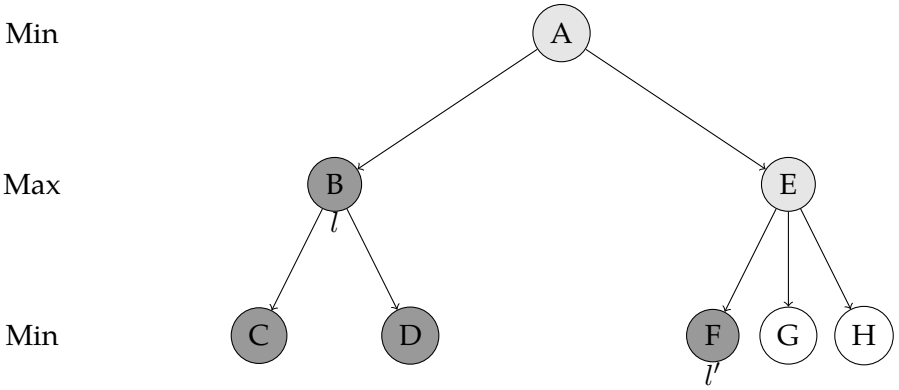
Aus diesem Grund ist es nicht mehr nötig, die Werte von G und H zu berechnen. D.h. diese Äste (G und H können größe Unterbäume besitzen!) braucht nicht betrachtet zu werden.

Um es nochmals zu verdeutlichen, betrachte den gleichen Baum mit konkreten Werten:



Obwohl das Minimum für die Berechnung des Wertes von E eigentlich -2 ergibt, kann die Suche beim Entdecken des Werts 2 abgebrochen werden, da die Bewertung für A (Maximieren), in jedem Fall die 3 aus dem linken Teilbaum bevorzugen wird (da $3 > 2$).

Schließlich gibt es den symmetrischen Fall, wenn an einem Knoten minimiert statt maximiert wird. Betrachte den Baum:



Der Baum unterscheidet sich vom vorherigen lediglich in der Minimierungs- und Maximierungsreihenfolge. Die Berechnung des Werts von A ist:

$$\begin{aligned} & \text{MinMax}(A, Min) \\ = \; & \texttt{min}\{\text{MinMax}(B, Max), \text{MinMax}(E, Max)\} \\ = \; & \texttt{min}\{\text{MinMax}(B, Max), \\ & \quad \texttt{max}\{\text{MinMax}(F, Min), \text{MinMax}(G, Min), \text{MinMax}(H, Min)\}\} \end{aligned}$$

Wenn $\text{MinMax}(B, Max) = l$ und $\text{MinMax}(F, Min) = l'$ schon berechnet sind, ergibt das $\texttt{min}\{l, \texttt{max}\{\text{l'}, \text{MinMax}(G, Min), \text{MinMax}(H, Min)\}\}$.

Wenn $l' \geq l$, dann brauchen die Werte von G und H nicht berechnet werden, da der Minimierer an A sowieso niedrigeren Wert k wählen wird.

Die Alpha-Beta-Suche verwendet genau diese Ideen und führt bei der Suche zwei Schranken mit, die das Suchfenster festlegen (und den Werten k und l in den obigen Beispielen entsprechen). Als untere Begrenzung des Suchfensters wird der Wert der Variablen α verwendet, als obere Begrenzung der aktuelle Wert der Variablen β. Das ergibt das Fenster $[\alpha, \beta]$. Der Algorithmus startet mit $\alpha = -\infty$ und $\beta = \infty$ und passt während der

Suche die Wert von α und β an. Sobald an einem Knoten $\alpha \geq \beta$ gilt, wird der entsprechende Teilbaum nicht mehr untersucht.

Beachte, dass die Alpha-Beta-Suche eine Tiefensuche mit Tiefenschranke durchführt (um die Terminierung sicherzustellen).

Algorithmus $\alpha - \beta$**-Suche**

Eingaben: Nachfolgerfunktion, Bewertungsfunktion, Tiefenschranke, Anfangsspielsituation und Spieler

Aufruf: AlphaBeta(Zustand,Spieler,$-\infty$,$+\infty$)

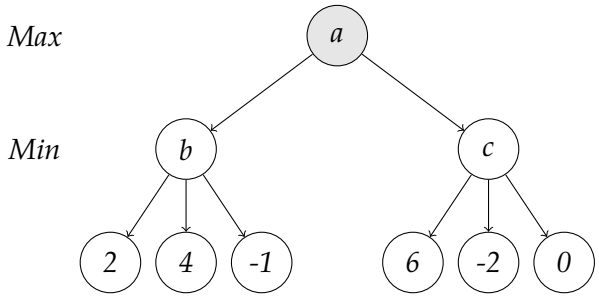
Funktion: AlphaBeta(Zustand,Spieler,α,β)

1. Wenn Tiefenschranke erreicht, bewerte die Situation, `return` Wert
2. Sei NF die Folge der Nachfolger von (Zustand,Spieler)
3. Wenn Spieler = Minimierer:
   $\beta_l := \infty$;
   `for-each` $L \in NF$
   $\beta_l :=$ `min`$\{\beta_l,$ AlphaBeta(L,Maximierer,α,β)$\}$
   `if` $\beta_l \leq \alpha$ `then return` β_l `endif` // verlasse Schleife
   $\beta :=$ `min`$\{\beta, \beta_l\}$
   `end-for`
   `return` β_l
4. Wenn Spieler = Maximierer
   $\alpha_l := -\infty$;
   `for-each` $L \in NF$
   $\alpha_l :=$ `max`$\{\alpha_l,$ AlphaBeta(L,Minimierer,α,β)$\}$
   `if` $\alpha_l \geq \beta$ `then return` α_l `endif` // verlasse Schleife
   $\alpha :=$ `max`$\{\alpha, \alpha_l\}$
   `end-for`
   `return` α_l

Beachte, dass der Algorithmus lokale α_l und β_l Werte verwendet, und dass die rekursiven Aufrufe auch entsprechend die aktuelle Tiefe anpassen müssen (was der Einfachheit halber in der Beschreibung weggelassen wurde).

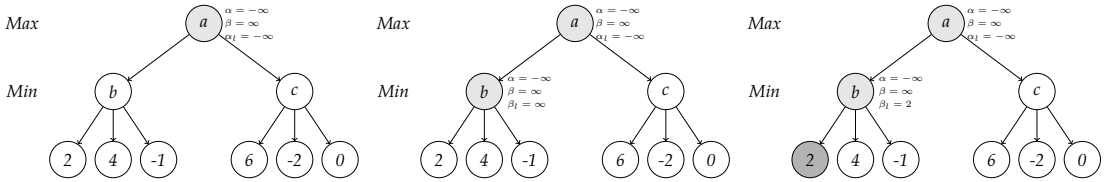
Wir betrachten ein Beispiel:

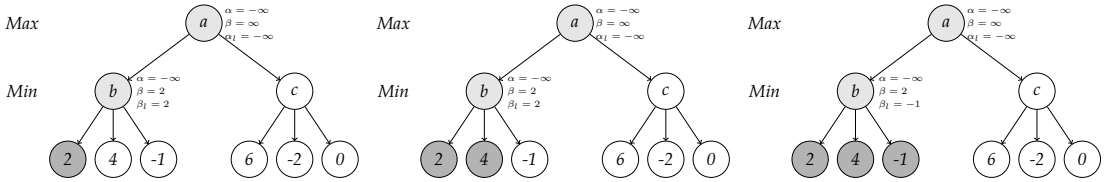
Beispiel 2.4.3. *Betrachte den folgenden Spielbaum, wobei der Maximierer an der Wurzel am Zug*

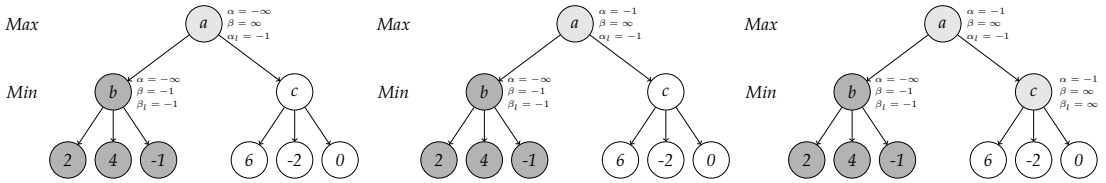


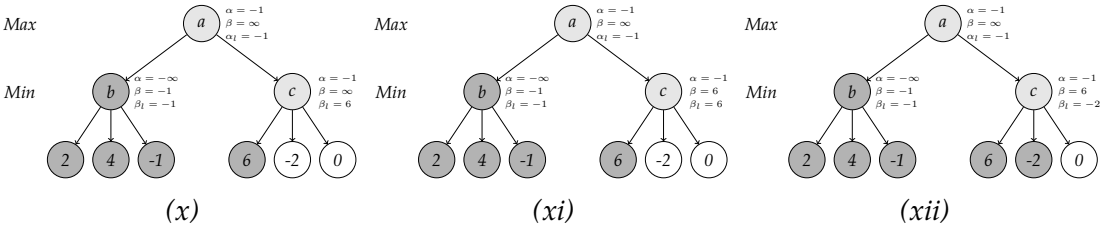
ist

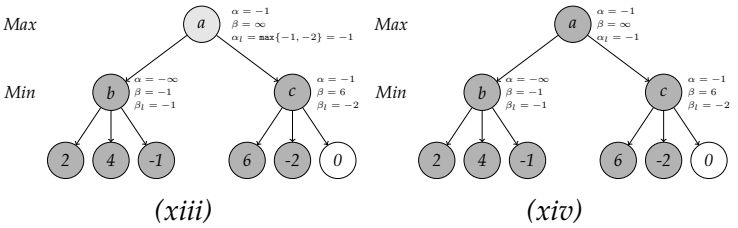
Die folgenden Abbildungen zeigen die Schrittweise Abarbeitung durch die Alpha-Beta-Suche:











Am Knoten c kann die Suche abgebrochen worden, nachdem die -2 als Kindwert entdeckt wurde. Daher kann braucht der mit 0 markierte Knoten (der auch ein großer Baum sein könnte) nicht untersucht werden. Der zugehörige Schritt ist (xii), da dort $\alpha \geq \beta_l$ (β_l ist der lokal beste gefundene β-Wert) festgestellt wird. Anschließend wird der β_l-Wert (-2) direkt nach oben gegeben, und zur

Aktualisierung des α_l-Wertes von Knoten (a) verwendet (Schritt (xiii)). Da dieser schlechter ist, als der Bereits erhaltene liefert die AlphaBeta-Suche den Wert -1 insgesamt als Ergebnis zurück.

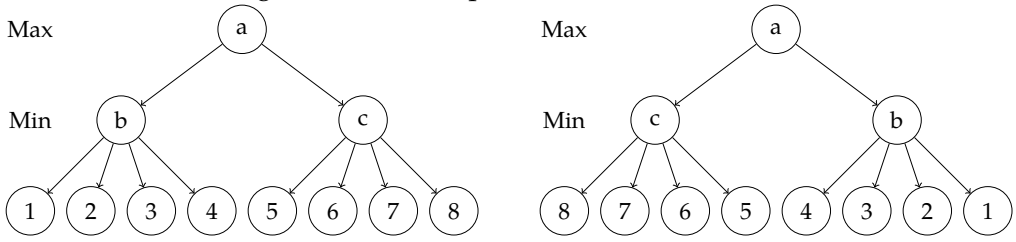
Die Alpha-Beta-Prozedur kann bei geeignetem Suchfenster auf jeder Ebene die Suche abbrechen, es muss nicht jeder innere Konten untersucht werden!.

Eine Implementierung der Alpha-Beta-Suche in Haskell (ohne Merken des Pfades) ist:

```
alphaBeta wert nachfolger maxtiefe neginfty infty spieler zustand  =
  ab 0 neginfty infty spieler zustand
 where
  ab tiefe alpha beta spieler zustand
   | null (nachfolger spieler zustand) || maxtiefe <= tiefe = wert zustand
   | otherwise =
      let l = nachfolger spieler zustand
      in
       case spieler of
         Maximierer -> maximize tiefe alpha beta l
         Minimierer -> minimize tiefe alpha beta l
  maximize tiefe alpha beta xs = it_maximize alpha neginfty xs
   where
      it_maximize alpha alpha_l []     = alpha_l
      it_maximize alpha alpha_l (x:xs) =
       let alpha_l' = max alpha_l (ab (tiefe+1) alpha beta Minimierer x)
       in  if alpha_l' >= beta then alpha_l' else
             it_maximize (max alpha alpha_l') alpha_l' xs
  minimize tiefe alpha beta xs = it_minimize beta infty xs
   where
      it_minimize beta beta_l []     = beta_l
      it_minimize beta beta_l (x:xs) =
        let beta_l' = min beta_l (ab (tiefe+1) alpha beta Maximierer x)
        in if beta_l' <= alpha then beta_l' else
             it_minimize (min beta beta_l') beta_l' xs
```

Beachte, dass die Güte der Alpha-Beta-Suche (gegenüber dem Min-Max-Verfahren) davon abhängt in welcher Reihenfolge die Nachfolgerknoten generiert und daher abgesucht werden.

Betrachte z.B. die folgenden beiden Spielbäume:



Beide Bäume stellen das selbe Spiel dar, lediglich die Reihenfolge der Nachfolgergenerierung ist verschieden. Im linken Baum, kann die Alpha-Beta-Suche keine Pfade abschneiden: Der für b ermittelte Wert 1 wird von allen Kindern von c übertroffen, daher

muss die Suche sämtliche Kinder von c explorieren (stets in der Hoffnung einen kleineren Wert zu finden). Im rechten Baum hingegen wird für c der Wert 5 ermittelt, da das erste Kind von b bereits den niedrigeren Wert 4 hat, wird die Alpha-Beta-Suche die restlichen Kinder von b nicht mehr explorieren.

Abhilfe kann hierbei wieder eine Heuristik schaffen, die entscheidet welche Kindknoten zuerst untersucht werden sollen (d.h. die Nachfolger werden anhand der Heuristik vorsortiert). Z.B. kann man bei Schach die verschiedenen Stellungen berücksichtigen und bspw. schlagende Züge reinem Ziehen vorziehen usw.

Schließlich stellt sich die Frage, wie groß die Wirkung der Alpha-Beta-Suche als Verbesserung der Min-Max-Suche ist. Wie bereits obiges Beispiel zur Reihenfolge der Nachfolger zeigt, ist im worst case die Alpha-Beta-Suche gar nicht besser als die Min-Max-Suche und muss alles inspizieren und benötigt daher $O(c^d)$ Zeit, wenn c die mittlere Verzweigungrate, d die Tiefenschranke ist und die Bewertungsfunktion konstante Laufzeit hat.

Für den best case kann man nachweisen, dass Alpha-Beta nur $O(c^{\frac{d}{2}})$ Knoten explorieren muss. Diese Schranke gilt auch im Mittel, wenn man eine gute Heuristik verwendet. Umgekehrt bedeutet das, dass man mit Alpha-Beta-Suche bei guter Heuristik bei gleichen Ressourcen (Zeit und Platz) *doppelt so tief* suchen kann, wie bei Verwendung der Min-Max-Suche. Damit wird auch die Spielstärke eines entsprechenden Programms gesteigert, denn tiefere Suche führt zur Verbesserung der Spielstärke.

Verwendet man zufälliges Raten beim Explorieren der Nachfolger, so kann man als Schranke für die Laufzeit $O(c^{\frac{3}{4}d})$ herleiten.

Wichtige Optimierungen der Suche, die z.B. beim Schach vorgenommen werden, sind die Speicherung bereits bewerteter Stellungen, um bei Zugvertauschungen nicht zuviele Stellungen mehrfach zu bewerten. Damit kann man die Vorsortierung von Zügen unterstützen.

Auftretende Probleme bei der Suche sind:

- Tiefere Suche kann in seltenen Fällen zur schlechteren Zugauswahl führen. Dies tritt mit immer geringerer Wahrscheinlichkeit ein, falls man eine gute Bewertungsfunktion hat
- Horizont-Effekt: (z.B. durch Schach bieten gerät ein wichtiger Zug außerhalb des Suchhorizonts)

Um den Horizont-Effekt in der Hauptvariante auszuschalten, kann man den Horizont unter gewissen Bedingungen erweitern.

Idee: falls bei Suche mit Tiefe d ein einzelner Zug k_0 besonders auffällt (d.h. gut beim Maximierer und schlecht beim Minimierer):

Wenn $v_d(k_0) > v_d(k) + S$ für alle Brüder $k \neq k_0$ und für feste, vorgegebene Schranke S, wobei v_d die Bewertung bei Suchtiefe d bezeichnet; Dann wird dieser Zug eine Tiefe weiter untersucht. Die tiefere Suche ist auch sinnvoll bei Zugzwang, d.h. wenn nur ein weiterer Zug möglich ist.

Im Falle des Zugzwangs vergrößert das den Suchraum nicht, aber erhöht die Vorausschau

Die Implementierung dieser Idee ist eine wichtige Erweiterung der Suchmethoden in Programmen zu strategischen Spielen.
Folgendes Vorgehen ist üblich: es gibt mehrere Durchgänge:

1. normale Bewertung

2. Erkennen aller besonderen Züge:

3. Alle auffallenden Züge dürfen um 1 weiter expandieren.

Effekte:

- Die Suche kann instabil werden:
  ob ein Zug auf Ebene n als interessant auffällt, kann davon abhängen, wie tief die Suche unterhalb dieses Knotens im Baum ist.

- Der Suchraum kann beliebig wachsen, falls man mehrere Durchgänge macht: jeweils ein anderer Knoten wird als wichtig erkannt.

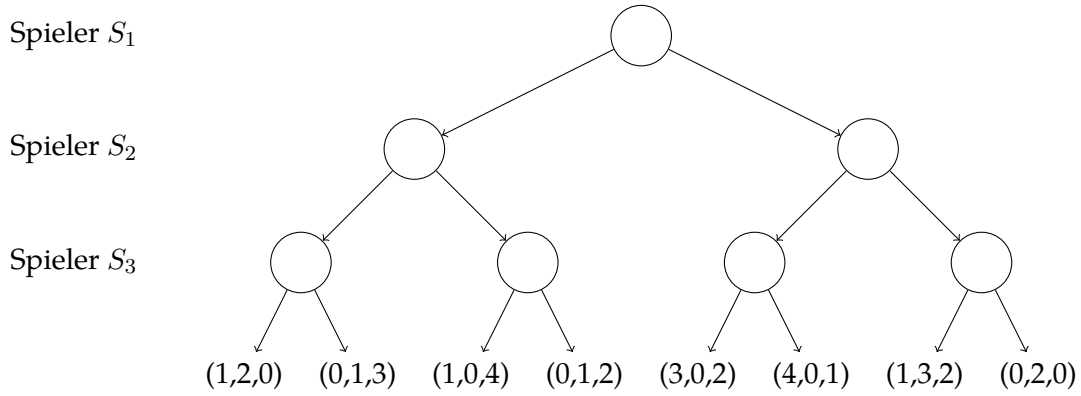
- Durch die tiefere Suche kann ein guter Zug wieder schlechter bewertet werden.

2.4.2 Mehr als 2 Spieler

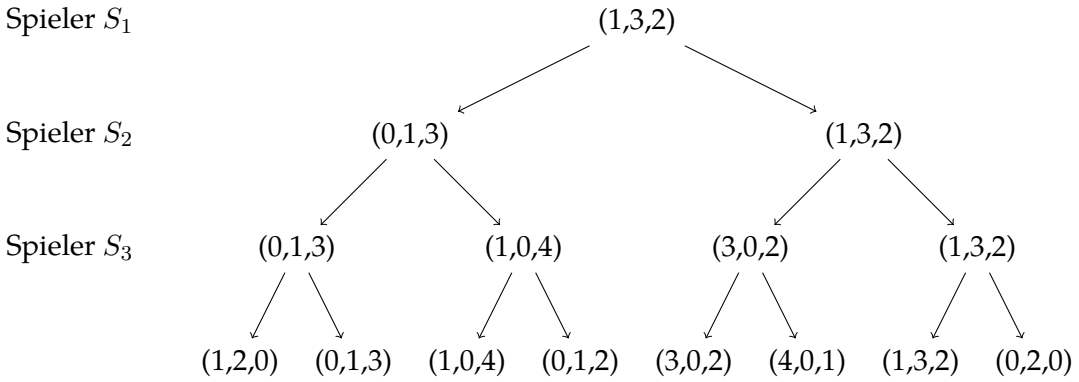
In diesem kurzen Abschnitt werden wir erläutern, warum die Min-Max-Methode (analog die Alpha-Beta-Suche) bei Spielen mit mehr als 2 Spielern nicht mehr funktioniert. Wir geben keine Lösungen für diese Spiele an, da die Verfahren komplizierte und aufwändig sind und in der Literatur nachgelesen werden können.

Wir betrachten ausschließlich Spiele mit drei Spielern (sagen wir S_1, S_2 und S_3), wobei alle Spieler nacheinander ziehen. Zur Blattbewertung macht es in diesem Fall keinen Sinn einen einzigen Wert zu verwenden, daher benutzen wir ein Drei-Tupel (Ergebnis für S_1, Ergebnis für S_2, Ergebnis für S_3). Jede Komponente stellt das Ergebnis aus Sicht des Spieler dar. Zunächst scheint das Min-Max-Verfahren einfach übertragbar: Je nachdem welcher Spiel an einem Knoten am Zug ist, wird entsprechend seiner Komponente maximiert.

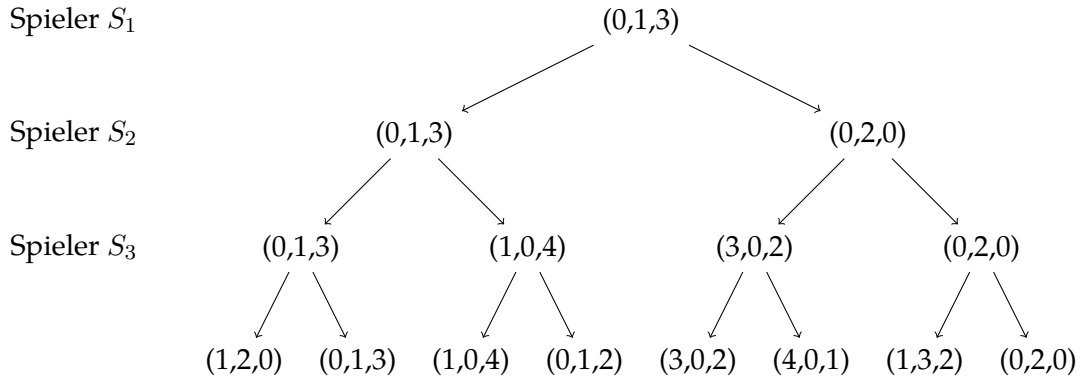
Betrachte z.B. den Spielbaum:



Bewertet man den Baum von unten nach oben, indem jeder Spieler sein Ergebnis maximiert, so erhält man:

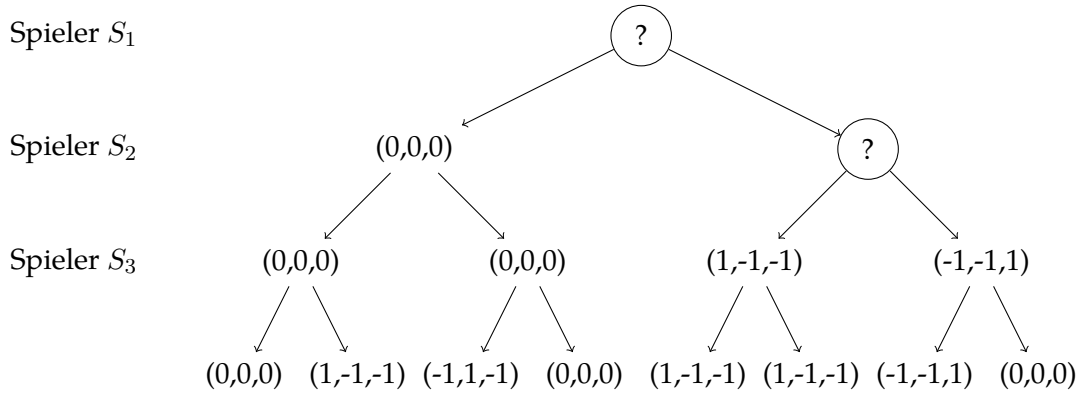


Hierbei sind wir jedoch davon ausgegangen, dass jeder Spieler nur nach der eigenen Gewinnmaximierung vorgeht. Nehmen wir an, Spieler 1 und Spieler 3 *verbünden sich*, mit dem Ziel, das Ergebnis von Spieler 2 zu minimieren, so ergibt sich eine andere Bewertung:



Noch gravierender ist der Fall, dass bei keiner Kooperation ein Spieler quasi zufällig einen Zug auswählt, da mehrere Nachfolger den gleichen Wert für ihn haben, aber diese Wahl die Bewertung der anderen Spieler beeinflussen kann.

Betrachte z.B. den Baum:



Spieler 2 hat zunächst keine Präferenz, wie er den zweiten Kindknoten bewerten soll, da die Bewertungen (1,-1,-1) und (-1,-1,1) aus seiner Sicht beide gleich schlecht sind. Allerdings nimmt er (1,-1,-1), so würde Spieler 1 an der Wurzel gewinnen, nimmt er (-1,-1,1) ist dies rational besser, da Spieler 1 an der Wurzel (0,0,0) also unentschieden nimmt.

Allgemein scheint es jedoch keine einfache Strategie zu geben, die stets zu einer rationalen Entscheidung führt.

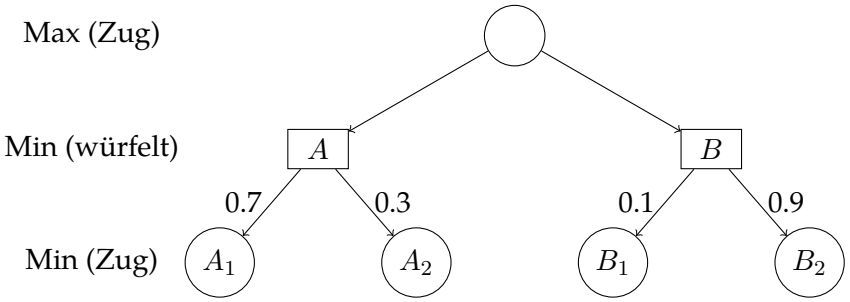
2.4.3 Spiele mit Zufallsereignissen

In diesem Abschnitt betrachten wir wieder Zwei-Spieler-Spiele, jedoch Spiele, bei denen der Zufall eine Rolle spielt. Auch in dieser Variante kann die Minimax-Suche mit leichten Anpassungen verwendet werden.

Z.B. beim Backgammon besteht ein Spielzug darin, zunächst mit zwei Würfeln zu würfeln und dann zu ziehen, wobei man bei der Zugwahl selbst entscheiden kann, welche Steine zu ziehen sind. Da der Gegner auch würfelt und erst dann eine Entscheidungsmöglichkeit für seinen Zug hat, das gleiche aber auch für weitere eigene Züge gilt, hat man keinen festen Baum der Zugmöglichkeiten. Man könnte einen Zug des Gegners vorausschauend über alle seine Würfel- und Zugmöglichkeiten minimieren, aber sinnvoller erscheint es, über die bekannte Wahrscheinlichkeit zu argumentieren und den Erwartungswert des Gegners zu minimieren und den eigenen Erwartungswert zu maximieren.

Theorem 2.4.4. *Es gilt: Wenn Wahrscheinlichkeit eine Rolle spielt, dann ist die Berechnung des Zuges mit dem besten Erwartungswert abhängig von den exakten Werten der Bewertungsfunktion; nicht nur von der relativen Ordnung auf den Situationen.*

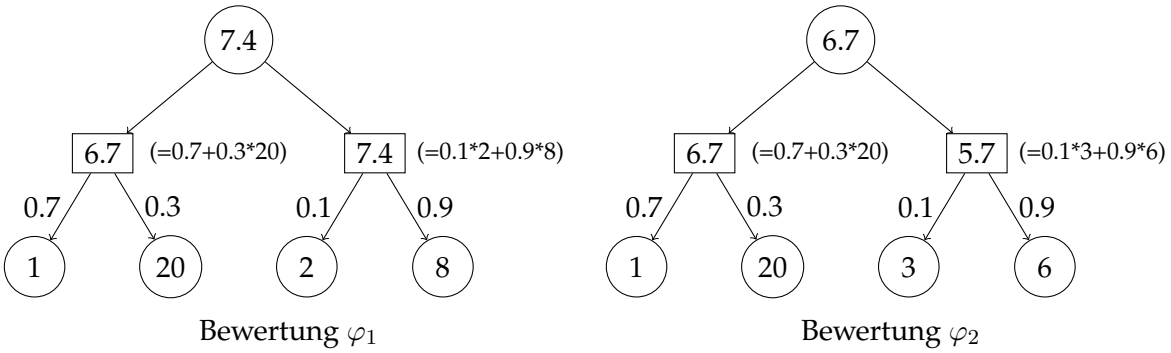
Beweis. Wir betrachten als Gegenbeispiel, den folgenden Spielbaum, wobei der Maximierer zunächst am Zug ist, und zwischen Zuständen A und B wählen kann. Im Anschluss beginnt der Zug des Minimierer, wobei dieser zunächst „würfelt“ um für Situation A zwischen A_1 (mit Wahrscheinlichkeit $0,7$) und A_2 (mit Wahrscheinlichkeit $0,3$) und für Situation B zwischen B_1 (mit Wahrscheinlichkeit $0,1$) und B_2 (mit Wahrscheinlichkeit $0,9$) zu entscheiden.



Wir betrachten zwei Bewertungen φ_1 und φ_2:

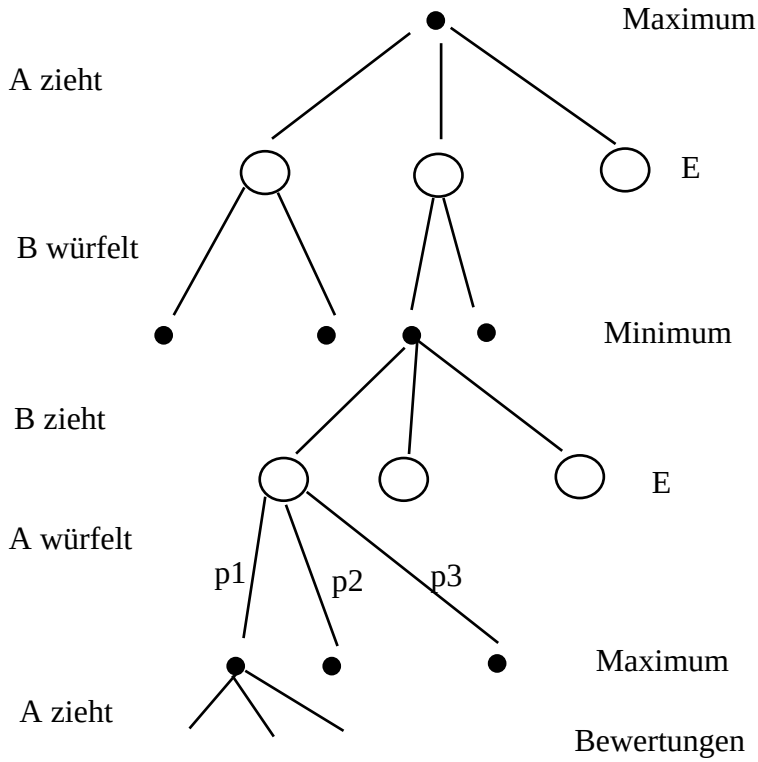
Bewertung	A_1	A_2	B_1	B_2
φ_1	1	20	2	8
φ_2	1	20	3	6

Beachte, dass die relative Ordnung Bewertung der Situationen bei φ_1 und φ_2 die gleiche ist ($\varphi_i(A_1) < \varphi_i(B_1) < \varphi_i(B_2) < \varphi_i(A_2)$ für $i = 1, 2$), aber die Erwartungswerte verschieben sich:



Damit ist einmal der rechte, einmal der linke Erwartungswert größer. D.h. einmal ist B, einmal A zu bevorzugen. □

D.h., die Bewertungsfunktion muss auch von den Werten her vorsichtig definiert werden. Beim Backgammon wäre die richtige Intuition: die Anzahl der benötigten Züge bis alle Steine das Ziel erreicht haben. Diese ist aber vermutlich schwer zu schätzen, da der Spielbaum viel zu groß ist.



Bei Vorausschau mit Tiefe 2 wird der Baum der Möglichkeiten aufgebaut. Die Blätter werden bewertet. Im Bild sind es die Situationen nach einem Zug von A. Man maximiert über die Geschwisterknoten und erhält den besten Zug. Im nächsten Schritt nach oben berechnet man den Erwartungswert für den Wert des besten Zuges. Geht man weiter nach oben, ist das gleiche zu tun für gegnerische Züge und Würfeln, wobei man minimieren muss. An der Wurzel ist das Würfelergebnis bekannt, so dass man als besten Zug denjenigen mit dem höchsten Erwartungswert ermittelt.

Hat man eine gute Bewertungsfunktion, dann kann man die Minimax-Suche verbessern durch eine Variante der Alpha-Beta-Suche. Diese Optimierung spielt allerdings in der Praxis keine so große Rolle wie in deterministischen Spielen, da der Spielbaum i.a. viel zu groß ist, um mehrere Züge voraus zu schauen.

2.5 Evolutionäre (Genetische) Algorithmen

Das Ziel bzw. die Aufgabe von evolutionären Algorithmen ist eine Optimierung von Objekten mit komplexer Beschreibung, wobei es variable Parameter gibt. Dabei werden die Objekte (Zustände) als Bitstrings kodiert, so dass die Aufgabe die Optimierung einer reellwertigen Bewertungsfunktion auf Bitfolgen fester Länge ist. Dabei wird stets eine Multimenge solcher Objekte betrachtet, d.h. es wird eine parallele Suche durchgeführt.

Genetische bzw. evolutionäre Algorithmen orientieren sich dabei an der Evolution von Lebewesen, daher werden entsprechende andere Sprechweisen benutzt, die wir gleich einführen. Die Analogie stimmt nicht ganz, denn die biologische Evolution hat kein expli-

zites Optimierungsziel. Es ist bekannt, dass die Rekombination bei der sexuellen Vermehrung eher darauf ausgerichtet ist, eine hohe genetische Diversifizierung der Individuen sicherzustellen und Gene in der Population zu verteilen, damit
die Population z.B. besser gerüstet bei einer Infektion oder bei schwankenden Umgebungsbedingungen (Hunger,Hitze, Kälte,...).

Die Eingaben eines evolutionären Algorithmus sind:

- Eine *Anfangspopulation*, d.h. eine (Multi-)Menge von Individuen (Zuständen, Objekten), die üblicherweise als Bitstring dargestellt sind. Ein Bit oder eine Teilfolge von Bits entspricht dabei einem Gen.

- Eine Bewertungsfunktion, welche Fitnessfunktion genannt wird.

Gesucht (als Ausgabe) ist ein *optimaler Zustand*

Die Anzahl der möglichen Zustände ist im Allgemeinen astronomisch, so dass eine Durchmusterung aller Zustände aussichtslos ist. Da die Evolution Lebewesen „optimiert", versucht man die Methoden der Evolution und zwar *Mutation*, *Crossover* (Rekombination) und *Selektion* analog auf diese Optimierungen zu übertragen, und mittels Simulationsläufen ein Optimum zu finden.

Die Idee ist daher

- Zustände (Individuen) werden als Bitfolgen (Chromosomen) kodiert.

- Die Bewertung entspricht der Fitness der Bitfolge (dieses Chromosomensatzes bzw. Individuums),

- Höhere Fitness bedeutet mehr Nachkommen

- Man beobachtet die Entwicklung einer Menge von Bitfolgen (*Population*). D.h. aus einer aktuellen Population wird die nächste Generation erzeugt.

- Nachkommen werden durch zufällige Prozesse unter Verwendung von Operationen analog zu Mutation und Crossover erzeugt

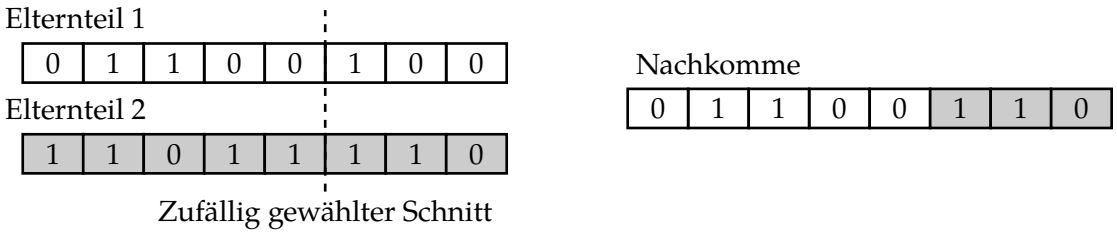
2.5.1 Genetische Operatoren

Um von einer Population der Generation n zur Population $n+1$ zu kommen, werden verschiedene Operatoren verwendet, die auf der Basis der Population zu n die Population $n+1$ ausrechnen. Wir beschreiben diese Operationen allgemein. Für spezifische Probleme können diese Operationen angepasst (oder auch verboten) sein.

Wahl der Anfangspopulation: Möglichst breit und gut verteilt.

Selektion: Unter *Selektion* versteht man die Auswahl eines Individuums (Chromosom) aus der Population zur Generation n Diese Auswahl kann zum Zwecke der Fortpflanzung oder der Mutation, des Weiterlebens usw. passieren. Diese Auswahl erfolgt für ein Individuum mit hoher Fitness mit höherer Wahrscheinlichkeit, wobei die Wahrscheinlichkeit auch vom Zweck der Selektion abhängen kann.

Rekombination Zwei Chromosomen werden ausgewählt mittels Selektion. Danach wird innerhalb der Chromosomen (zufällig) eine Stelle zum Zerschneiden ermittelt, ab dieser werden die Gene ausgetauscht. Es entstehen zwei neue Chromosomensätze. Alternativ kann man auch mehrere Abschnitte der Chromosomen austauschen. Die folgende Darstellung illustriert die Rekombination:



Mutation. mit einer gewissen Wahrscheinlichkeit werden zufällig ausgewählte Bits in den Chromosomen verändert. ($0 \to 1$ bzw. $1 \to 0$)

Aussterben Wenn die Populationsgröße überschritten wird, stirbt das schlechteste Individuum. Alternativ ein zufällig gewähltes, wobei das schlechteste mit größerer Wahrscheinlichkeit als des beste ausstirbt.

Ende der Evolution Wenn vermutlich das Optimum erreicht ist.

Ein genetischer / evolutionärer Algorithmus beginnt daher mit einer initialen Population (1. Generation) und erzeugt durch die genetischen Operationen Nachfolgegenerationen. Er stoppt, wenn ein optimales Individuum gefunden wurde, oder nach einer Zeitschranke. Im letzteren Fall wird nicht notwendigerweise ein Optimum gefunden, aber häufig (z.B. bei Verteilungsproblemen) ein ausreichend gutes Ergebnis.

Ein einfaches Grundgerüst für einen genetischen Algorithmus ist:

Algorithmus **Genetische Suche**

Eingabe: Anfangspopulation, Fitnessfunktion ϕ, K die Populationsgröße

Datenstrukturen: S, S' : Mengen von Individuen
Algorithmus:
$S :=$ Anfangspopulation;
`while` (S enthält kein Individuum mit maximalem ϕ) `do`:
 $S' := \emptyset$
 `for` $i := 1$ `to` K `do`:
  Wähle zufällig (mit ϕ gewichtet) zwei Individuen A und B aus S;
  Erzeuge Nachkommen C aus A und B durch Rekombination;
  C' := Mutation (geringe Wahrscheinlichkeit) von C;
  $S' := S' \cup \{C'\}$;
 `end for`
 $S := S'$
`end while`
Gebe Individuum mit maximaler Fitness aus

Allerdings sind weitere genetische Operationen und Anpassungen von Mutation, Rekombination und Selektion möglich. Außerdem ist es evtl. ratsam besonders gute Individuen ohne Rekombination in die nächste Generation zu übernehmen (weiterleben bzw. Klonen).

Im Allgemeinen sind die adjustierende *Parameter* vielfältig:

- Erzeugung der initialen Population und Kriterien zum Beenden
- Mutationswahrscheinlichkeit: wieviel Bits werden geflippt?, welche Bits?
- Crossover-Wahrscheinlichkeit: an welcher Stelle wird zerschnitten? Welche Individuen dürfen Nachkommen zeugen?
- Abhängigkeit der Anzahl der Nachkommen von der Fitness
- Größe der Population

Wie bereits in den vorherigen Abschnitten zur Suche sind auch folgende Aspekte hier wichtig:

- Finden einer geeigneten Repräsentation der Zustände
- Finden geeigneter genetischer Operatoren (auf der Repräsentation).

Problem: auch bei genetischen Algorithmen kann sich die Population um ein lokales Maximum versammeln. Z.B. durch Inzucht oder durch einen lokalen Bergsteigereffekt, bei dem die Mutation das lokale Optimum nicht verlassen kann.

Die Codierung der Chromosomen ist im allgemeinen eine Bitfolge, in diesem Fall spricht man von *genetischen Algorithmen.*
Die Kodierung kann aber auch dem Problem angepasst sein, so dass nicht mehr alle Kodierungen einem Problem entsprechen (ungültige Individuen). In dem Fall ist es meist auch sinnvoll, Operatoren zu verwenden, die nur gültige Nachkommen erzeugen. In diesem Fall spricht man von *evolutionären Algorithmen*

2.5.2 Ermitteln der Selektionswahrscheinlichkeit

Alternativen:

1. Proportional zur Fitness.
   Dies macht die Optimierung sehr sensibel für den genauen Wert und der Steigung der Fitness. Flacht die Fitness ab – z.B. in der Nähe zum Optimum oder auf einem Plateau– dann gibt es keine bevorzugte Auswahl mehr.

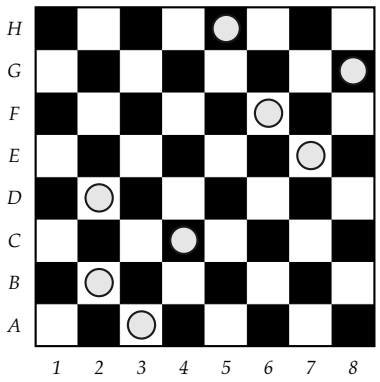
2. Proportional zur Reihenfolge innerhalb der Population.
   z.B bestes Individuum 2-3 fach wahrscheinlicher als schlechtestes Individuum. In diesem Fall ist man flexibler in der Auswahl der Fitness-Funktion. Die Auswahl hängt nicht vom genauen Wert ab.

3. Proportional zu einer normierten Fitness (z.B. Fitness $-c$), wobei man die Fitness c als das Minimum wählen kann.

Beispiel 2.5.1. *n-Dame mit evolutionären Algorithmen.*
Wir geben zwei verschiedene Kodierungen an.
Kodierung 1:

Codierung der Individuen *: Ein Individuum ist eine Folge von n Zahlen, wobei die i. Zahl die Position (Spalte) der Dame in Zeile i angibt.*

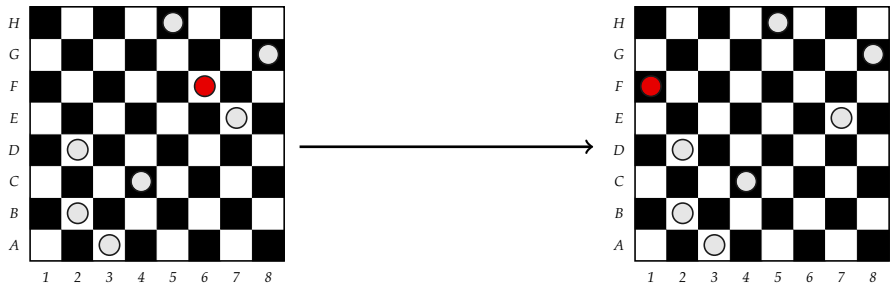
Z.B. für das 8-Damenproblem kodiert die Folge [3,2,4,2,7,6,8,5] gerade die Belegung:



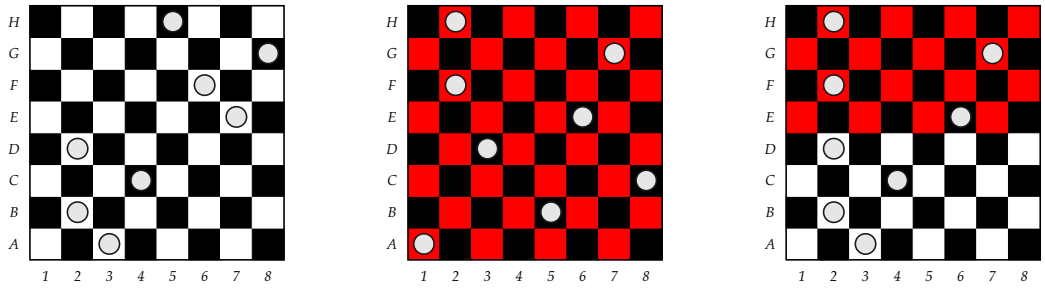
Fitness: *Anzahl der Damenpaare, die sich nicht gegenseitig bedrohen. Im schlechtesten Fall bedrohen sich alle Paare gegenseitig, und die Fitness ist daher 0. Im besten Fall (Optimum, Ziel) sind dies alle Damenpaare (keine Dame bedroht eine andere). Der Wert hierfür ist* $\binom{n}{2} = \frac{n*(n-1)}{2}$.

Operationen: *Mutation ändert einen Eintrag in der Folge auf einen beliebigen Wert zwischen 1 und* n. *Dies entspricht gerade dem Verschieben einer einzelnen Dame in der entsprechenden Zeile.*

Z.B. entspricht die Mutation des Individuums [3,2,4,2,7,6,8,5] zum [3,2,4,2,7,1,8,5] gerade der folgenden Bewegung:



Rekombination entspricht gerade dem horizontalen Teilen der zwei Schachfelder. Z.B. beim 8-Damenproblem und den Individuen [3,2,4,2,7,6,8,5] und [1,5,8,3,6,2,7,2] entsteht durch Rekombination mit Schnitt in der Mitte das Individuum [3,2,4,2,6,2,7,2]:



Da man jedoch beim Damespiel schon weiß, das Zustände mit doppelten Zahlen in der Folge nicht gültig sind kann man alternativ wie folgt kodieren:

Man lässt nur Permutationen *der Zahlen von 1 bis* n *als Individuen zu. Als Mutationsoperator verwendet man das Austauschen zweier Zahlen, Die Rekombination scheint für diese Kodierung wenig sinnvoll, da sie im Allgemeinen keinen gültigen Individuen erzeugt. Wenn man sie verwendet, sollte man verhindern, dass verbotene Individuen entstehen:*

1. *großer negativer Wert der Fitnessfunktion*

2. *sofortiges Aussterben*

3. *deterministische oder zufällige Abänderung des Individuums in ein erlaubtes*

Eine beispielhafte Veränderung einer Population für das 5-Damenproblem, wobei die Population nur aus 1-2 Elementen besteht:

Population	$\{[3,2,1,4,5]\}$
Fitness	$\varphi([3,2,1,4,5]) = 4$
Mutationen	$[3,2,1,4,5] \rightarrow [5,2,1,4,3]$; $[3,2,1,4,5] \rightarrow [3,4,1,2,5]$
Population	$\{[5,2,1,4,3],[3,4,1,2,5]\}$
Bewertung	$\varphi([5,2,1,4,3]) = 6$, $\varphi([3,4,1,2,5]) = 6$
Mutationen	$[3,4,1,2,5] \rightarrow [2,5,1,4,3]$, $[5,2,1,4,3] \rightarrow [5,2,4,1,3]$
Population	$\{[5,2,1,4,3],[3,4,1,2,5],[2,5,1,4,3],[5,2,4,1,3]\}$
Bewertung	$\varphi([2,5,1,4,3]) = 8$, $\varphi([5,2,4,1,3]) = 10$
Selektion	$\{[2,5,1,4,3],[5,2,4,1,3]\}$

Die Kodierung $[5,2,4,1,3]$ *ist eine Lösung.*

Reine Mutationsveränderungen sind analog zu Bergsteigen und Best-First.

2.5.2.1 Bemerkungen zu evolutionären Algorithmen:

Wesentliche Unterschiede zu Standardsuchverfahren

- Evolutionäre Algorithmen benutzen Codierungen der Lösungen
- Evolutionäre Algorithmen benutzen eine Suche, die parallel ist und auf einer Menge von Lösungen basiert.
- Evolutionäre Algorithmen benutzen nur die Zielfunktion zum Optimieren
- Evolutionäre Algorithmen benutzen probabilistische Übergangsregeln. Der Suchraum wird durchkämmt mit stochastischen Methoden.

Eine gute Kodierung erfordert *annähernde Stetigkeit*. D.h. es sollte einen annähernd stetigen Zusammenhang zwischen Bitfolge und Fitnessfunktion geben. D.h. optimale Lösungen sind nicht singuläre Punkte, sondern man kann sie durch Herantasten über gute, suboptimale Lösungen erreichen.

Damit Crossover zur Optimierung wesentlich beiträgt, muss es einen (vorher meist unbekannten) Zusammenhang geben zwischen Teilfolgen in der Kodierung (Gene) und deren Zusammenwirken bei der Fitnessfunktion. Es muss so etwas wie „gute Gene “geben, deren Auswirkung auf die Fitness eines Individuums sich ebenfalls als gut erweist.

Baustein-Hypothese (Goldberg, 1989) (zur Begründung des Crossover)
(Building block hypothesis)
Genetische Algorithmen verwenden einfache Chromosomenbausteine und sammeln und mischen diese um die eigene Fitness zu optimieren.

Hypothese: das Zusammenmischen vieler guter (kleiner) Bausteine ergibt das fitteste Individuum

2.5.2.2 Parameter einer Implementierung

Es gibt viele Varianten und Parameter, die man bei genetischen/evolutionären Algorithmen verändern kann.

- Welche Operatoren werden verwendet? (Mutation, Crossover, ...)
- Welche Wahrscheinlichkeiten werden in den Operatoren verwendet? Wahrscheinlichkeit einer Mutation, usw.
- Welcher Zusammenhang soll zwischen Fitnessfunktion und Selektionswahrscheinlichkeit bestehen?
- Wie groß ist die Population? Je größer, desto breiter ist die Suche angelegt, allerdings dauert es dann auch länger.
- Werden Individuen ersetzt oder dürfen sie weiterleben?
- Gibt es Kopien von Individuen in der Population?
- ...

2.5.3 Statistische Analyse von Genetischen Algorithmen

Vereinfachende Annahme: Fitness = Wahrscheinlichkeit, dass ein Individuum 1 Nachkommen in der nächsten Generation hat.

Es soll die Frage untersucht werden: „wie wächst die Häufigkeit einzelner Gene?“ von Generation zu Generation?

D.h. wie ist die „genetische Drift“?

Sei S die Menge der Individuen, die ein bestimmtes Gen G enthält.
Die „Schema-theorie“ stellt diese mittels eines Schemas dar:
z.B. durch $[****10101****]$. Die Folge $1\,0\,1\,0\,1$ in der Mitte kann man als gemeinsames Gen der Unterpopulation S ansehen. Die ebenfalls vereinfachende Annahme ist, dass das Gen vererbt wird: die Wahrscheinlichkeit der Zerstörung muss gering sein.

Sei $p()$ die Fitnessfunktion (und Wahrscheinlichkeit für Nachkommen), wobei man diese auch auf Mengen anwenden darf, und in diesem Fall die Fitness aller Individuen summiert wird. Sei V die Gesamtpopulation.

Der Erwartungswert der Anzahl der Nachkommen aus der Menge S ist, wenn man annimmt, dass $|V|$ mal unabhängig ein Nachkomme ermittelt wird.
$p(s_1) * |V| + \ldots + p(s_{|S|}) * |V| = p(S) * |V|$

Verbreitung tritt ein, wenn $p(S) * |V| > |S|$, d.h. wenn $\frac{p(S)}{|S|} > 1/|V|$ ist. D.h. wenn die mittlere Wahrscheinlichkeit in der Menge $|S|$ erhöht ist:

$$\frac{p(S)}{|S|} > 1/|V|$$

Damit kann man $\frac{p(S)}{|S|}$ als Wachstumsfaktor der Menge S ansehen. Wenn dieser Vorteil in der nächsten Generation erhalten bleibt, dann verbreitet sich dieses Gen exponentiell. Es tritt aber eine Sättigung ein, wenn sich das Gen schon ausgebreitet hat. Analog ist das Aussterben von schlechten Genen exponentiell.

Eine etwas andere Analyse ergibt sich, wenn man annimmt, dass über die Generationen die Fitness a der S-Individuen konstant ist, ebenso wie die Fitness b der Individuen in $V \setminus S$. Dann erhält man die Nachkommenswahrscheinlichkeit durch Normierung der Fitness auf 1. Wir nehmen an, dass S_t die Menge der Individuen zum Gen S in der Generation t ist.

Das ergibt

$$p_S := \frac{a}{a * |S_t| + b * (|V| - |S_t|)}$$

Damit ist der Erwartungswert für $|S_{t+1}|$:

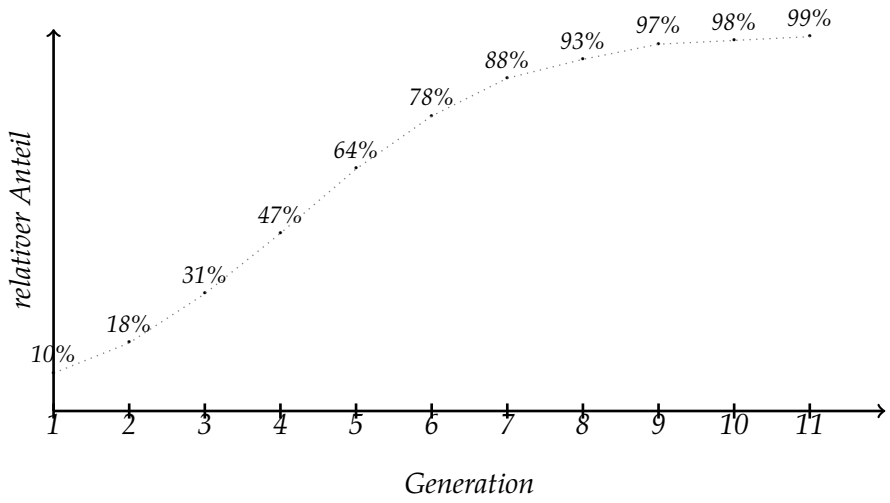
$$|S_{t+1}| = \frac{a * |S_t|}{a * |S_t| + b * (|V| - |S_t|)} * |V|$$

Wenn wir $r(S,t)$ für den relativen Anteil von S schreiben, dann erhalten wir:

$$|r(S,t+1)| = \frac{a}{a * r(S,t) + b * (1 - r(S,t))} * r(S,t).$$

Beispiel 2.5.2. *Wählt man als Wert $a = 2b$, dann erhält man als Funktion $\frac{2r}{r+1}$, folgende Werte und Kurve für die Anteile, wenn man mit $0,1$ startet*

t	1	2	3	4	5	6	7	8	9	10	11
$r(S,t)$	$0,1$	$0,18$	$0,31$	$0,47$	$0,64$	$0,78$	$0,88$	$0,93$	$0,97$	$0,98$	$0,99$



2.5.4 Anwendungen

Man kann evolutionäre Algorithmen verwenden für Optimierungen von komplexen Problemstellungen, die

- keinen einfachen Lösungs- oder Optimierungsalgorithmus haben,
- bei denen man relativ leicht sagen kann, wie gut diese Lösungen sind
- die nicht in Echt-Zeit lösbar sein müssen, d.h. man hat ausreichend (Rechen-)Zeit um zu optimieren.
- bei denen man aus einer (bzw. zwei) (Fast-)Lösungen neue Lösungen generieren kann.
- wenn man die Problemstellung leicht mit weiteren Parametern versehen will, und trotzdem noch optimieren können will.

Beispiele:

Handlungsreisenden-Problem.

Es gibt verschiedene Kodierung, und viele Operatoren, die untersucht wurden. Die einfachste ist eine Kodierung als Folge der besuchten Städte. Die Fitnessfunktion ist die Weglänge (evtl. negativ).
Mutationen, die gültige Individuen (Wege) erzeugen soll, braucht Reparaturmechanismen. Man kann einen algorithmischen Schritt dazwischen schalten, der **lokale Optimierungen** erlaubt, z.B. kann man alle benachbarten Touren prüfen, die sich von der zu optimierenden Tour nur um 2 Kanten unterscheiden.
Ein vorgeschlagener Operator ist z.B. das partielle Invertieren: Man wählt zwei Schnittpunkte in einer gegebenen Tour aus, und invertiert eine der Teiltouren mit anschließender Reparatur.

Ähnliches Bemerkungen wie für Mutation gelten für **Crossover**, das eine neue Tour aus Teiltouren anderer Touren der Population zusammensetzt.

Operations Research-Probleme: Schedulingprobleme, Tourenplanung.

SAT (Erfüllbarkeit von aussagenlogischen Formeln)
Bei SAT kann man die Interpretation als Bitstring kodieren: Man nimmt eine feste Reihenfolge der Aussagenvariablen und schreibt im Bitstring die Belegung mit 0/1 hin. Mutation und Crossover erzeugen immer gültige Individuen. Als Fitnessfunktion kann man die Anzahl der wahren Klauseln nehmen.

Optimale Verteilung der Studenten auf Proseminare (Bachelor-Seminare). Dies wird im Institut mit einem evolutionären Algorithmus optimiert (geschrieben von Herrn Björn Weber)

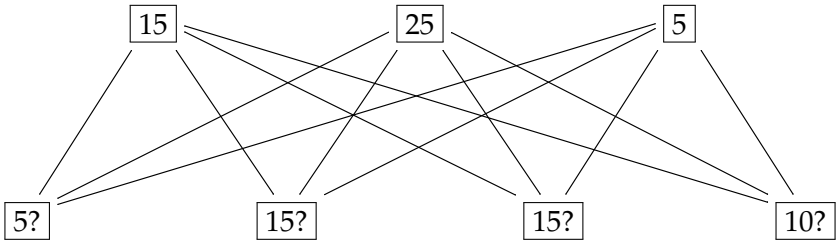
2.5.5 Transportproblem als Beispiel

Wir betrachten als Beispiel ein einfaches Transportproblem mit 3 Lagern L_1, L_2, L_3 von Waren und 4 Zielpunkten $Z_1, \ldots, Z_4$. [4]. Man kennt die Kosten für den Transport von Waren von L_i nach Z_i pro Einheit.

Folgende Daten sind gegeben:

Lagerbestand			Warenwünsche			
L_1	L_2	L_3	Z_1	Z_2	Z_3	Z_4
15	25	5	5	15	15	10

Die Wünsche sind erfüllbar. Ein Bild ist:



Die Kosten pro Einheit sind:

	Z_1	Z_2	Z_3	Z_4
L_1	10	0	20	11
L_2	12	7	9	20
L_3	0	14	16	18

[4] siehe (Michalewicz, 1992)

Eine optimale Lösung ist:

	5	15	15	10
15	0	5	0	10
25	0	10	15	0
5	5	0	0	0

Die Kosten sind:
$5 * 0 + 10 * 11 + 10 * 7 + 15 * 9 + 5 * 0 = 315$
Will man das Problem mittels evolutionärer Algorithmen lösen, so kann man verschiedene Kodierungen wählen.

1. Nehme die Lösungsmatrix als Kodierung.
2. Nehme eine Matrix der gleichen Dimension wie die Lösungsmatrix, Prioritäten 1,2,3,...als Einträge.

Als Fitnessfunktion wählt man die Kosten (bzw. negativen Kosten: Ersparnisse).

1. hat den Vorteil, dass man eine exakte Kodierung hat. Allerdings weiss man bei diesem Problem, dass man die billigsten Fuhren zuerst nehmen kann, so dass man damit evtl. Lösungen variiert, die man mit etwas Einsicht in das Problem als äquivalent ansehen kann. Als Bedingung hat man noch, dass es keine negativen Werte geben soll und dass die Spaltensummen den Wünschen entsprechen sollen, die Zeilensummen sollen nicht größer sein als der jeweilige Lagerbestand.
2. hat den Vorteil, dass alle Kodierung gültig sind und man das Wissen über das Problem nutzt. Aus den Prioritäten wird nach folgendem festgelegten Schema eine Lösung berechnet: zunächst wird die Fuhre mit der Priorität 1 so voll wie möglich gemacht. Danach unter Beachtung, dass schon geliefert wurde, die Fuhre mit Priorität 2. usw. Hierbei verliert man keine optimalen Lösungen. Zudem kann man leichter neue Kodierungen erzeugen.

   Eine Kodierung entspricht einer Permutation der 12 Matrixeinträge, die als Folge von 12 Zahlen geschrieben werden kann.

Mutation in 1. : verschiebe eine Wareneinheit auf eine andere Fuhre. Hierbei kann aber eine ungültige Lösung entstehen. Crossover: unklar.

Mutation in 2) : permutiere die Folge durch Austausch von 2 Prioritäten. Hierbei entsteht eine gültige Kodierung.

Inversion: invertiere die Reihenfolge der Prioritäten:

Crossover: Wähle zwei Permutation p_1, p_2 aus der Population aus, und wähle aus p_1 ein Unterfolge aus. Entferne aus p_2 die Prioritäten, die in p_2 vorkommen, ergibt Folge

p'_2. Dann setze die Prioritäten an den freien Stellen in p'_2 wieder ein.
Alternativ (aber ohne bezug: setze die Prioritäten an irgendwelchen Stellen wieder ein.

2.5.6 Schlußbemerkungen

Bemerkung 2.5.3. *Es gibt generische Programme, die die Berechnung der Generationen usw. bereits ausführen, und denen man nur noch die Kodierung der Problemstellung geben muss, die Kodierung der Operatoren und die Parameter.*

Es gibt auch verschiedene eingebettete implementierte Anwendungen

Bemerkung 2.5.4. *Für komplexe Probleme und deren Optimierung gilt die Heuristik: mit einem Evolutionären Algorithmus kann man immer ganz brauchbar verbessern, wenn man wenig über die Struktur und Eigenschaft der Problemklasse sagen kann.*

Aber sobald man sich mit der Problemklasse genauer beschäftigt und experimentiert, wird ein Optimierungsalgorithmus informierter sein, und sich mehr und mehr zu einem eigenständigen Verfahren entwickeln, das bessere Ergebnisse produziert als allgemeine Ansätze mit evolutionären Algorithmen.

Probleme der genetischen/evolutionären Algorithmen ist die möglicherweise sehr lange Laufzeit Man braucht mehrere Durchläufe, bis man eine gute EInstellung der Parameter gefunden hat. Auch problematisch ist, dass unklar ist, wann man mit Erfolg aufhören kann und wie nahe man einem Optimum ist.

3
Maschinelles Lernen

3.1 Einführung: Maschinelles Lernen

Da die direkte Programmierung eines intelligenten Agenten sich als nicht möglich herausgestellt hat, ist es klar, dass man zum Erreichen des Fernziels der Künstlichen Intelligenz eine Entsprechung eines Lernprozesses benötigt, d.h. man benötigt *Maschinelles Lernen*.

Es gibt viele verschiedene Ansichten darüber, was Maschinelles Lernen ist, was mit Lernen erreicht werden soll usw. Hier sind erst Anfänge in der Forschung gemacht worden. Die praktisch erfolgreichsten Methoden sind solche, die auf statistisch/stochastischen Methoden basieren und mit der Adaption von Werten (Gewichten) arbeiten:

- Adaption von Gewichten einer Bewertungsfunktion aufgrund von Rückmeldungen. Z.B. Verarbeitung natürlicher Sprachen, Strategie-Spiele mit und ohne zufällige Ereignisse: Dame, Backgammon.

- Künstliche neuronale Netze: Lernen durch gezielte Veränderung von internen Parametern. Deren praktischer Nutzen und Anwendbarkeit ist im Wesentlichen auf praktikable automatische Lernverfahren zurückzuführen.

Das Lernen von neuen Konzepten, Verfahren, logischen Zusammenhängen, usw. hat bisher nur ansatzweise Erfolg gehabt.

Für den Agenten-basierten Ansatz, soll das Lernen eine Verbesserung der Performanz des Agenten bewirken:

- Verbesserung der internen Repräsentation.

- Optimierung bzw. Beschleunigung der Erledigung von Aufgaben.

- Erweiterung des Spektrums oder der Qualität der Aufgaben, die erledigt werden können.

Beispiel 3.1.1. *Drei beispielhafte Ansätze sind:*

- *Erweiterung und Anpassung des Lexikons eines computerlinguistischen Systems durch automatische Verarbeitung von geschrieben Sätzen, wobei der Inhalt dieser Sätze gleichzeitig automatisch erfasst werden sollte.*

- *Adaption von Gewichten einer Bewertungsfunktion in einem Zweipersonenspiel, wobei man abhängig von Gewinn/Verlust Gewichte verändert. Das wurde für Dame und Backgammon mit Erfolg durchgeführt.*

- *Lernen einer Klassifikation durch Vorgabe von Trainingsbeispielen, die als positiv/negativ klassifiziert sind.*

3.1.1 Einordnung von Lernverfahren

Die Struktur eines lernenden Systems kann man wie folgt beschreiben:

Agent (ausführende Einheit, performance element). Dieser soll verbessert werden anhand von Erfahrung; d.h. etwas lernen.

Lerneinheit (learning element). Hier wird der Lernvorgang gesteuert und bewertet. Insbesondere wird hier vorgegeben, was gut und was schlecht ist. Hier kann man auch die Bewertungseinheit (critic) und den Problemgenerator einordnen.

Umwelt In der Umwelt soll agiert werden. Die Rückmeldung über den Ausgang bzw. den Effekt von Aktionen kommt aus dieser Umwelt. Das kann eine künstliche, modellhafte Umwelt oder auch die reale Umwelt sein.

Zum Teil wird Agent und Lerneinheit zusammen in einen erweiterten Agent verlagert. Prinzipiell sollte man diese Einheiten unterscheiden, denn die Bewertung muss außerhalb des Agenten sein, sonst wäre die Möglichkeit gegeben, die Bewertung an die schlechten Aktionen anzupassen, statt die Aktionen zu verbessern.

Folgende *Lernmethoden* werden unterschieden:

Überwachtes Lernen (supervised learning). Diese Methode geht von der Situation aus in der es einen allwissenden Lehrer gibt. Die Lerneinheit kann dem Agenten bei jeder Aktion sagen, ob diese richtig war und was die richtige Aktion gewesen wäre. Das entspricht einem unmittelbaren Feedback über die exakt richtige Aktion. Alternativ kann man eine feste Menge von richtigen und falschen Beispielen vorgeben und damit dann ein Lernverfahren starten.

Unüberwachtes Lernen (unsupervised learning). Dies ist der Gegensatz zum überwachten Lernen. Es gibt keine Hinweise, was richtig sein könnte. Damit Lernen möglich ist, braucht man in diesem Fall eine Bewertung der Güte der Aktion.

Lernen durch Belohnung/Bestrafung (reinforcment learning). Dieser Ansatz verfolgt das Prinzip „mit Zuckerbrot und Peitsche“. Hiermit sind Lernverfahren gemeint, die gute Aktionen belohnen, schlechte bestrafen, d.h. Aktionen bewerten, aber die richtige Aktion bzw. den richtigen Parameterbereich nicht kennen.

Man kann man die Lernverfahren noch unterscheiden nach der Vorgehensweise. Ein *inkrementelles* Lernverfahren lernt ständig dazu. Ein anderer Ansatz ist ein *Lernverfahren mit Trainingsphase*: Alle Beispiele werden auf einmal gelernt.

Man kann die Lernverfahren auch entsprechend der Rahmenbedigungen einordnen:

- Beispielwerte sind exakt oder ungefähr bekannt bzw. mit Fehlern behaftet
- Es gibt nur positive bzw. positive und negative Beispiele

3.1.2 Einige Maßzahlen zur Bewertung von Lern- und Klassifikationsverfahren

Wir beschreiben kurz Vergleichsparameter, die man zur Abschätzung der Güte von Klassifikatorprogrammen bzw. Lernverfahren verwendet.

Beispiel 3.1.2. *Beispiele, um sich besser zu orientieren:*

- *Klassifikation von „Vogel“ anhand bekannter Attribute, wie kann-fliegen, hat-Federn, usw.*
- *Vorhersage, dass ein Auto noch ein Jahr keinen Defekt hat aufgrund der Parameter wie Alter, gefahrene Kilometer, Marke, Kosten der letzten Reparatur, usw.*
- *Medizinischer Test auf HIV: Antikörper*
- *Vorhersage der Interessen bzw. Kaufentscheidung eines Kunden aufgrund der bisherigen Käufe und anderer Informationen (online-Buchhandel).*
- *Kreditwürdigkeit eines Kunden einer Bank, aufgrund seines Einkommens, Alters, Eigentumsverhältnisse, usw (Adresse?).*

Ein *Klassifikator* ist ein Programm, das nur binäre Antworten auf Anfragen gibt: ja / nein. Die Aufgabe ist dabei, Objekte, beschrieben durch ihre Attribute, bzgl. einer anderen Eigenschaft zu klassifizieren, bzw. eine zukünftiges Ereignis vorherzusagen. Typische Beispiele sind die Bestimmung von Tier- Pflanzenarten anhand eines Exemplars, oder die Diagnose einer Krankheit anhand der Symptome.

Wir beschreiben die *abstrakte Situation* genauer. Es gibt:

- eine Menge M von Objekten (mit innerer Strukur)
- das Programm $P : M \to \{0, 1\}$, welches Objekte klassifiziert
- die wahre Klassifikation $K : M \to \{0, 1\}$

Bei Eingabe eines Objekts $x \in M$ gilt:

- Im Fall $K(x) = P(x)$ liegt das Programm richtig. Man unterscheidet in

  richtig-positiv Wenn $P(x) = 1$ und $K(x) = 1$.

  richtig-negativ Wenn $P(x) = 0$ und $K(x) = 0$.

- Im Fall $K(x) \neq P(x)$ liegt das Programm falsch. Hier wird noch unterschieden zwischen

  falsch-positiv Wenn $P(x) = 1$, aber $K(x) = 0$.

  falsch-negativ Wenn $P(x) = 0$, aber $K(x) = 1$.

Die folgenden Werte entsprechen der Wahrscheinlichkeit mit der das Programm P eine richtige positive (bzw. negative) Klassifikation macht. Es entspricht der Wahrscheinlichkeit, mit der eine Diagnose auch zutrifft. Hierbei wird angenommen, dass es eine Gesamtmenge M aller Objekte gibt, die untersucht werden.

Recall (Richtig-Positiv-Rate, Sensitivität, Empfindlichkeit, Trefferquote; sensitivity, true positive rate, hit rate):

Der Anteil der richtig klassifizierten Objekte bezogen auf alle tatsächlich richtigen.

$$\frac{|\{x \in M \mid P(x) = 1 \wedge K(x) = 1\}|}{|\{x \in M \mid K(x) = 1\}|}$$

Richtig-Negativ-Rate (true negative rate oder correct rejection rate, Spezifität)

Der Anteil der als falsch erkannten bezogen auf alle tatsächlich falschen:

$$\frac{|\{x \in M \mid P(x) = 0 \wedge K(x) = 0\}|}{|\{x \in M \mid K(x) = 0\}|}$$

Die folgenden Werte entsprechen der Wahrscheinlichkeit mit der ein als positiv klassifiziertes Objekt auch tatsächlich richtig klassifiziert ist, bzw. die Wahrscheinlichkeit mit der eine positive Diagnose sich als richtig erweist. Oder anders herum: eine negativ Diagnose die Krankheit ausschließt.

Der Wert der Präzision ist ein praktisch relevanterer Wert als der recall, da diese aussagt, wie weit man den Aussagen eines Programms in Bezug auf eine Klassifikation trauen kann.

Precision (Präzision, positiver Vorhersagewert, Relevanz, Wirksamkeit, Genauigkeit, positiver prädiktiver Wert, positive predictive value)

Der Anteil der richtigen unten den als scheinbar richtig erkannten

$$\frac{|\{x \in M \mid P(x) = 1 \wedge K(x) = 1\}|}{|\{x \in M \mid P(x) = 1\}|}$$

Negative-Vorhersage-Rate Der Anteil richtig als falsch klassifizierten unter allen als falsch klassifizierten

$$\frac{|\{x \in M \mid P(x) = 0 \wedge K(x) = 0\}|}{|\{x \in M \mid P(x) = 0\}|}$$

Im medizinischen Bereich sind alle diese Werte wichtig. Bei seltenen Krankheiten kann ein guter Recall, d.h. der Anteil der Kranken, die erkannt wurden, mit einer sehr schlechten Präzision verbunden sein. Zum Beispiel, könnte ein Klassifikator für Gelbfieber wie folgt vorgehen: Wenn Körpertemperatur über 38,5 C, dann liegt Gelbfieber vor. In Deutschland haben 10.000 Menschen Fieber mit 38,5 C aber nur 1 Mensch hat Gelbfieber, der dann auch Fieber hat. Dann ist der Recall 1, aber die Precision ist 0.0001, also sehr schlecht. Hier muss man also möglichst beide Größen ermitteln, und den Test genauer machen (precision erhöhen).

3.2 Wahrscheinlichkeit und Entropie

In diesem Abschnitt führen wir den Begriff der *Entropie* ein, wobei wir zunächst eine kurze Wiederholung zu diskreten Wahrscheinlichkeiten geben.

3.2.1 Wahrscheinlichkeit

Sei X ein Orakel, das bei jeder Anfrage einen Wert aus der Menge $\{a_1, \ldots, a_n\}$ ausgibt, d.h. X ist analog zu einer Zufallsvariablen. Man interessiert sich für die *Wahrscheinlichkeit* p_i, dass das Orakel den Wert a_i ausgibt. Macht man (sehr) viele Versuche, so kommt in der Folge der Ergebnisse $b_1, \ldots, b_m$, für ein festes i der Anteil der a_i in der Folge dem Wert p_i immer näher. Man nennt die Zahlen $p_i, i = 1, \ldots, n$ auch *diskrete Wahrscheinlichkeitsverteilung* (der Menge a_i), bzw. des Orakels X.

Zum Beispiel ist beim Münzwurf mit den Ausgängen $Kopf$ und $Zahl$ in einer ausreichend langen Folge in etwa die Hälfte $Kopf$, die andere Hälfte $Zahl$, d.h. man würde hier Wahrscheinlichkeiten $0,5$ und $0,5$ zuordnen.

Es gilt immer $0 \leq p_i \leq 1$ und $\sum_i p_i = 1$. Sind die a_i Zahlen, dann kann man auch den *Erwartungswert* ausrechnen: $E(X) = \sum_i p_i a_i$. Das ist der Wert, dem die Mittelwerte der (Zahlen-)Folgen der Versuche immer näher kommen.

Wenn man die Arbeitsweise von X kennt, dann kann man mehr Angaben machen. Z.B. das sogenannte *Urnenmodell*:

> X benutzt einen Eimer in dem sich Kugeln befinden, rote, blaue und grüne. Bei jeder Anfrage wird „zufällig“ eine Kugel gezogen, deren Farbe ist das Ergebnis, und danach wird die Kugel wieder in den Eimer gelegt.

In diesem Fall sind die Wahrscheinlichkeiten p_{rot}, p_{blau}, $p_{\text{grün}}$ jeweils genau die *relativen Häufigkeiten* der roten, blauen, bzw. grünen Kugeln unter den Kugeln, die sich in der Urne jeweils vor dem Ziehen befinden.

3.2.2 Entropie

Zunächst führen wir den Begriff des Informationsgehalts ein, der von einigen Lernverfahren benötigt wird.

Wenn man eine diskrete Wahrscheinlichkeitsverteilung $p_i, i = 1, \ldots, n$ hat, z.B. von Symbolen $a_i, i = 1, \ldots, n$, dann nennt man

$$I(a_k) := \log_2(\frac{1}{p_k}) = -\log_2(p_k) \geq 0$$

den *Informationsgehalt* des Zeichens a_k. Das kann man interpretieren als Grad der Überraschung beim Ziehen des Symbols a_k aus einer entsprechenden Urne, bzw. bei der Übermittlung von Zeichen durch einen Kommunikationskanal. D.h. das Auftreten eines seltenen Symbols hat einen hohen Informationsgehalt. Wenn man nur ein einziges Symbol hat, dann ist $p_1 = 1$, und der Informationsgehalt ist $I(a_1) = 0$. Eine intuitive Erklärung des Informationsgehalts ist die mittlere Anzahl der Ja/Nein-Fragen, die man stellen muss, um die gleiche Information zu bekommen.

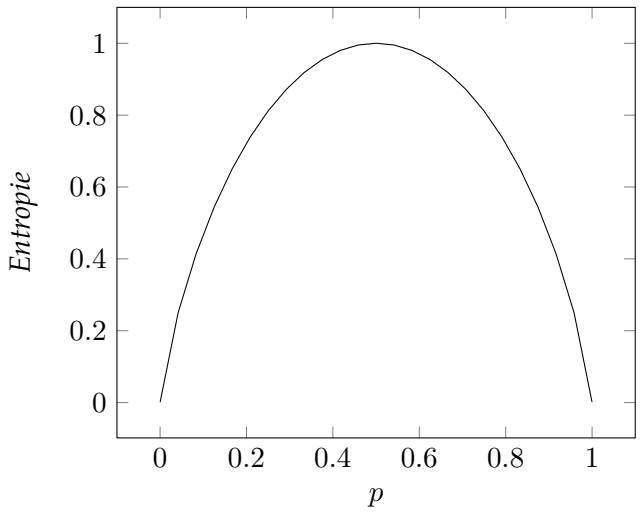
Beispiel 3.2.1. *Zum Beispiel im Falle von 8 Objekten, die gleich oft vorkommen, ergibt sich* $\log(0.125) = -3$ *für jedes Objekt, d.h. der Informationsgehalt jedes Zeichens ist* 3 *und auch der* mittlere Informationsgehalt*, ermittelt aus der gewichteten Summe ist* 3*. Kommen zwei davon, sagen wir mal* a_1, a_2*, sehr häufig vor und die anderen praktisch nie, dann ergibt sich als mittlerer Informationsgehalt in etwa* $0.5 * \log_2(0.5) + 0.5 * \log_2(0.5) + 6 * 0.001 * \log_2(0.001) \approx 1$.

Die *Entropie* oder der *mittlere Informationsgehalt* der Symbole in der Wahrscheinlichkeitsverteilung wie oben kann dann berechnet werden als

$$I(X) = \sum_{i=1}^{n} p_i * \log_2(\frac{1}{p_i}) = -\sum_{i=1}^{n} p_i * \log_2(p_i) \geq 0.$$

Bei Kompressionen eines Files oder bei Kodierung von Nachrichten über einem Kanal ist das eine untere Schranke für die mittlere Anzahl von Bits pro Symbol, die man bei bester Kompression bzw binärer Kodierung erreichen kann.

Beispiel 3.2.2. *Nimmt man ein Bernoulli-Experiment, d.h. zwei Zeichen,* $Kopf$ *und* $Zahl$ *wobei* $Kopf$ *mit der Wahrscheinlichkeit* p *und* $Zahl$ *mit Wahrscheinlichkeit* $1 - p$ *auftritt, dann ergibt sich in etwa die Kurve:*



D.h. die Entropie (der mittlere Informationsgehalt eines Münzwurfs) ist maximal, wenn man das Zeichen nicht vorhersagen kann. Bei einer Wahrscheinlichkeit von $p = 0,9$ *kann man vorhersagen, dass* $Kopf$ *sehr oft auftritt. Das ist symmetrisch zu* $p = 0,1$*. Die Entropie ist in beiden Fällen* $0,469$*.*

3.3 Lernen mit Entscheidungsbäumen

Wir betrachten nun das Lernen mit sogenannten *Entscheidungsbäume*. Diese Bäume repräsentieren das Klassifikator-Programm. Hierbei wird zunächst mit einer Menge von Beispielen, der Entscheidungsbaum aufgebaut, der im Anschluss als Klassifikator verwendet werden kann. Unser Ziel wird es sein, Algorithmen vorzustellen, die einen *guten* Entscheidungsbaum erstellen. Güte meint hierbei, dass der Baum möglichst flach ist. Man kann sich dies auch anders verdeutlichen: Hat man Objekte mit vielen Attributen, so möchte man die Attribute herausfinden, die markant sind, für die Klassifikation, und diese zuerst testen. Unter guten Umständen, braucht man dann viele Attribute gar nicht zu überprüfen. Als wesentliches Hilfsmittel der vorgestellten Algorithmen wird sich das Maß der Entropie erweisen.

3.3.1 Das Szenario und Entscheidungsbäume

Wir fangen allgemein mit dem zu betrachtenden Szenario an. Gegeben ist eine Menge von Objekten, von denen man einige Eigenschaften (Attribute) kennt. Diese Eigenschaften kann man darstellen mit einer fest vorgegebenen Menge von n Attributen. D.h. man kann jedes Objekt durch ein n-Tupel der Attributwerte darstellen.

Definition 3.3.1. *Wir definieren die wesentlichen Komponten des Szenarios:*

- *Es gibt eine endliche Menge A von Attributen.*

- *Zu jedem Attribut $a \in A$ gibt es eine Menge von möglichen Werten W_a. Die Wertemengen seien entweder endlich, oder die reellen Zahlen: $\mathbb{R}$.*
- *Ein Objekt wird beschrieben durch eine Funktion $A \to \times_{a \in A} W_a$. Eine alternative Darstellung ist: Ein Objekt ist ein Tupel mit $|A|$ Einträgen, bzw. ein Record, in dem zu jedem Attribut $a \in A$ der Wert notiert wird.*
- *Ein* Konzept *K ist repräsentiert durch eine Prädikat P_K auf der Menge der Objekte. D.h. ein Konzept entspricht einer Teilmenge aller Objekte, nämlich der Objekte o, für die $P_K(o) =$ `True` ergibt.*

Beispiel 3.3.2. *Bücher könnte man beschreiben durch die Attribute: (Autor, Titel, Seitenzahl, Preis, Erscheinungsjahr). Das Konzept „billiges Buch" könnte man durch Preis ≤ 10 beschreiben. Das Konzept „umfangreiches Buch" durch Seitenzahl ≥ 500.*

Für die Lernverfahren nimmt man im Allgemeinen an, dass jedes Objekt zu jedem Attribut einen Wert hat, und daher der Wert „unbekannt" nicht vorkommt. Im Fall unbekannter Attributwerte muss man diese Verfahren adaptieren.

Definition 3.3.3 (Entscheidungsbaum)**.** *Ein* Entscheidungsbaum *zu einem Konzept K ist ein endlicher Baum, der an inneren Knoten zum Wert eines Attributes folgende Abfragen machen kann:*

- *bei reellwertigen Attributen gibt es die Alternativen $a \leq v$ oder $a > v$ für einen Wert $v \in \mathbb{R}$, Es gibt einen Teilbaum für Ja und einen für Nein.*
- *bei diskreten Attributen wird der exakte Wert abgefragt. Es gibt pro möglichem Attributwert einen Teilbaum*

Die Blätter des Baumes sind mit Ja oder Nein markiert. Das entspricht der Antwort auf die Frage, ob das eingegebene Objekte zum Konzept gehört oder nicht.

Diskrete Attribute sollten pro Pfad im Baum nur einmal vorkommen, stetige Attribute können im Pfad mehrmals geprüft werden.

D.h. ein Entscheidungsbaum B_K ist die Darstellung eines Algorithmus zum Erkennen, ob ein vorgelegtes Objekt O zum Konzept K gehört.

Jeder Entscheidungsbaum definiert ein Konzept auf den Objekten. Die Entscheidungsbäume sind so definiert, dass für jedes Objekt nach Durchlauf des Entscheidungsbaumes ein Blatt mit Ja oder Nein erreicht wird.

Die Mengen der Objekte, bei denen der Pfad mit einem Ja endet, sind in diesem Konzept, die anderen Objekte nicht.

- Wenn es nur diskrete Attribute gibt, dann entsprechen die Konzepte genau den Entscheidungsbäumen: Zu jedem Konzept kann man offenbar eine (aussagenlogische) Formel in DNF angeben: die $a_1 = v_1 \wedge \ldots a_n = v_n$ als Konjunktion enthält, wenn

das Tupel $(v_1, \ldots, v_n)$ im Konzept enthalten ist. Diese kann man leicht in einen Entscheidungsbaum überführen.

- Bei Verwendung von reellwertigen Attributen kann nicht jedes Konzept durch einen endlichen Entscheidungsbaum beschrieben werden: z.B. alle geraden Zahlen. Auch in einfachen Fällen, in denen das Konzept durch $\bigcup I_i$, d.h. als Vereinigung von unendlich vielen reellen Intervallen, dargestellt ist, gilt das.

Beispiel 3.3.4. *Als praktische Anwendung kann man reale Konzepte mittels einer endlichen Menge von Attributwerten bezüglich einer vorher gewählten Menge von Attributen beschreiben. Das ist i.A. eine Approximation des realen Konzepts.*

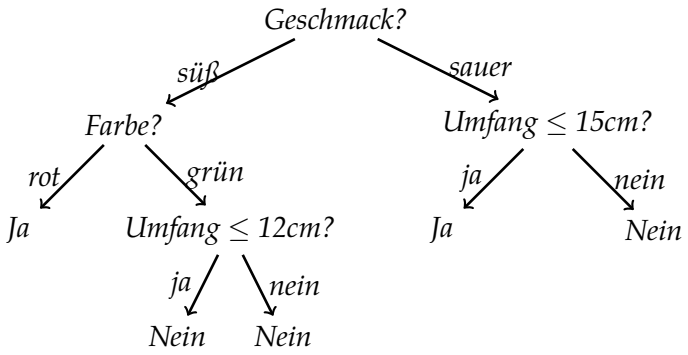
Tiere könnte man z.B. durch folgende Attribute beschreiben:

Größe	*reell*
Gewicht	*reell*
Kann fliegen	*Boolesch*
Nahrung	*pflanzlich / tierisch / Allesfresser*
Körpertemperatur	*reell*

Für die Menge der Insekten könnte man aufgrund dieser Attribute einen Entscheidungsbaum hinschreiben, allerdings würden dann auch Nichtinsekten mit Ja klassifiziert.

Beispiel 3.3.5. *Seien die Objekte Äpfel mit Attributen: Geschmack (süß/sauer), Farbe (rot/grün), Umfang (in cm).*

Ein Entscheidungsbaum zum Konzept: „guter Apfel“ könnte sein:



Man sieht schon, dass dieser Entscheidungsbaum überflüssige Abfragen enthält: Da die Frage nach „Umfang ≤ 12cm?“ nur in „Nein“-Blättern endet, kann die Abfrage direkt durch ein „Nein“-Blatt ersetzt werden.

Es gibt verschiedene Algorithmen, die die Aufgabe lösen sollen, einen Entscheidungsbaum für ein Konzept zu lernen, wobei man beispielsweise eine Menge von positiven Beispielen und eine Menge von negativen Beispielen vorgibt.

Ein *guter Entscheidungsbaum* ist ist einer der eine möglichst kleine mittlere Anzahl von Anfragen bis zur Entscheidung benötigt; und auch noch möglichst klein ist.

Der im Folgenden verwendetet Entropie-Ansatz bewirkt, dass das Verfahren einen Entscheidungsbaum erzeugt der eine *möglichst kleine mittlere Anzahl* von Anfragen bis zur Entscheidung benötigt. Einen Beweis dazu lassen wir weg. Das Verfahren ist verwandt zur Konstruktion von Huffman-Bäumen bei Kodierungen.

3.3.2 Lernverfahren ID3 und C4.5

Es wird angenommen, dass alle Objekte vollständige Attributwerte haben, und dass es eine Menge von positiven Beispielen und eine Menge von negativen Beispielen für ein zu lernendes Konzept gibt, die möglichst gut die echte Verteilung abbilden. Für rein positive Beispielmengen funktionieren diese Verfahren nicht.

Wichtig für die Lernverfahren ist es, herauszufinden, welche Attribute für das Konzept irrelevant bzw. relevant sind. Nachdem ein Teil des Entscheidungsbaumes aufgebaut ist, prüfen die Lernverfahren die Relevanz weiterer Attribute bzw. Attributintervalle.

Das Lernverfahren ID3 (Iterative Dichotomiser 3) iverwendet den Informationsgehalt der Attribute bezogen auf die Beispielmenge. Der Informationsgehalt entspricht der mittleren Anzahl der Ja/Nein-Fragen, um ein einzelnes Objekt einer Klasse zuzuordnen. Das Lernverfahren versucht herauszufinden, welche Frage den größten Informationsgewinn bringt, wobei man sich genau auf die in einem Entscheidungsbaum erlaubten Fragen beschränkt. Das Ziel ist daher die mittlere Anzahl der Fragen möglichst klein zu halten.

Sei M eine Menge von Objekten mit Attirbuten. Wir berechnen den Informationsgehalt der Frage, ob ein Beispiel positiv/negativ ist in der Menge aller positiven / negativen Beispiele. Sei p die Anzahl der positiven und n die Anzahl der negativen Beispiele für das Konzept. Wir nehmen eine Gleichverteilung unter den Beispielen an, d.h. wir nehmen an, dass die relative Häufigkeit die reale Verteilung in den Beispielen widerspiegelt. Die Entropie bzw. der Informationsgehalt ist dann:

$$I(M) = \frac{p}{p+n} * \log_2(\frac{p+n}{p}) + \frac{n}{p+n} * \log_2(\frac{p+n}{n})$$

Sei a ein Attribut. Wir schreiben $m(a)$ für den Wert des Attributs a eines Objekts $m \in M$. Hat man ein mehrwertiges Attribut a mit den Werten $w_1, \ldots, w_n$ abgefragt, dann zerlegt sich die Menge M der Beispiele in die Mengen $M_i := \{m \in M \mid m(a) = w_i\}$, wobei $w_i, i = 1, \ldots, k$ die möglichen Werte des Attributes sind. Seien p_i, n_i für $i = 1, \ldots, k$ die jeweilige Anzahl positiver (negativer) Beispiele in M_i, dann ergibt sich nach Abfragen des Attributs an Informationsgehalt (bzgl. positiv/negativ), wobei $I(M_i)$ der Informationsgehalt (bzgl. positiv/negativ) der jeweiligen Menge M_i ist.

$$I(M|a) = \sum_{i=1}^{k} P(a = w_i) * I(M_i)$$

D.h. es wird der nach relativer Häufigkeit gewichtete Mittelwert der enstandenen Infor-

mationsgehalte der Kinder berechnet (was analog zu einer bedingten Wahrscheinlichkeit nun ein bedingter Informationsgehalt ist). Für jedes solches Kind mit Menge M_i gilt wiederum:

$$I(M_i) = \frac{p_i}{p_i + n_i} * \log_2(\frac{p_i + n_i}{p_i}) + \frac{n_i}{p_i + n_i} * \log_2(\frac{p_i + n_i}{n_i})$$

Insgesamt können wir daher $I(M|a)$ berechnen als:

$$I(M|a) = \sum_{i=1}^{k} \frac{p_i + n_i}{p + n} * \Big(\frac{p_i}{p_i + n_i} * \log_2(\frac{p_i + n_i}{p_i}) + \frac{n_i}{p_i + n_i} * \log_2(\frac{p_i + n_i}{n_i})\Big)$$

Im Falle, dass $M_i = \emptyset$ ist, nehmen wir an, dass der Beitrag zur Summe 0 ist. Um Fallunterscheidungen zu vermeiden, nehmen wir an, dass Produkte der Form $\frac{0}{a} * \log_2(\frac{a}{0})$ als 0 zählen. Das ist berechtigt, da der Grenzwert von $\lim_{x \to 0} x * \log_2(x) = 0$ ist.

Definition 3.3.6 (ID3: Entscheidungsbaum Lernen)**.** *ID3 startet mit einem leeren Baum und als Eingabe einer Menge M von positiven und negativen Beispielen. Die Wurzel ist am Anfang der einzige* offene Knoten *mit Menge M.*

Die Abbruchbedingung für einen offenen Knoten ist: Die Menge M besteht nur noch aus positiven oder nur noch aus negativen Beispielen. In diesem Fall wird der Knoten geschlossen, indem er mit „Ja“ bzw. „Nein“ markiert wird. Ein unschöner Fall ist, dass die Menge M leer ist: In diesem Fall muss man auch abbrechen, kennt aber die Markierung des Blattes nicht, man hat beide Möglichkeiten Ja bzw. Nein.

Für jeden offenen Knoten mit mit Menge M wird jeweils das Attribut ausgewählt, das den größten Informationsgewinn bietet. D.h. dasjenige a, für das der

$$\text{Informationsgewinn: } I(M) - I(M|a)$$

maximal ist. Der offene Knoten wird nun geschlossen, indem das entsprechende Attribut am Knoten notiert wird und für jede Attributsausprägung wird ein Kind samt der entsprechenden Menge von Objekten berechnet. Die enstandenen Knoten sind offene Knoten und werden nun im Verfahren rekursiv (mit angepassten Menge M) bearbeitet.

Beachte, wenn die Abbruchbedingung erfüllt ist, ist der Informationsgehalt an diesem Blatt 0. Normalerweise gibt es eine weitere praktische Verbesserung: Es gibt eine Abbruchschranke: wenn der Informationsgewinn zu klein ist für alle Attribute, dann wird der weitere Aufbau des Entscheidungsbaum an diesem Knoten abgebrochen, und es wird „Ja“ oder „Nein“ für das Blatt gewählt.

Anmerkungen zum Verfahren:

- Durch diese Vorgehensweise wird in keinem Ast ein diskretes Attribut zweimal abgefragt, da der Informationsgewinn 0 ist.

- Der Algorithmus basiert auf der Annahme, dass die vorgegebenen Beispiele repräsentativ sind. Wenn dies nicht der Fall ist, dann weicht das durch den Entscheidungsbaum definierte Konzept evtl. vom intendierten Konzept ab.

- Wenn man eine Beispielmenge hat, die den ganzen Tupelraum abdeckt, dann wird genau das Konzept gelernt.

Beispiel 3.3.7. *Wir nehmen als einfaches überschaubares Beispiel Äpfel und die Attribute Geschmack* $\in \{süß, sauer\}$ *und Farbe* $\in \{rot, grün\}$*. Das Konzept sei „guter Apfel".*

Es gibt vier Varianten von Äpfeln, $\{(süß, rot), (süß, grün), (sauer, rot), (sauer, grün)\}$*. Wir geben als Beispiel vor, dass die guten Äpfel genau* $\{(süß, rot), (süß, grün)\}$ *sind. Wir nehmen mal an, dass pro Apfelvariante genau ein Apfel vorhanden ist.*
Es ist offensichtlich, dass die guten genau die süßen Äpfel sind, und die Farbe egal ist. Das kann man auch nachrechnen, indem man den Informationsgewinn bei beiden Attributen berechnet:

Der Informationsgehalt $I(M)$ *vor dem Testen eines Attributes ist:*

$$0.5log_2(2) + 0.5log_2(2) = 1$$

Nach dem Testen des Attributes „Geschmack" ergibt sich als Informationsgehalt $0,5 * (log_2(1) + 0) + 0,5 * (0 + log_2(1)) = 0$*, d.h. Der Informationsgewinn ist maximal.*

Nach dem Testen des Attributes „Farbe" ergibt sich als Informationsgehalt $0,5 * (0,5log_2(2) + 0,5log_2(2)) + 0,5 * (0,5log_2(2) + 0,5log_2(2)) = 0,5 * 1 + 0,5 * 1 = 1$*, d.h. man gewinnt nichts.*

Als Variation des Beispiels nehmen wir irgendeine Anzahl der Äpfeln in jeder Kategorie an:

süß,rot	*süß,grün*	*sauer,rot*	*sauer,grün*
10	*20*	*4*	*6*

D.h. es gibt 30 gute und 10 schlechte Äpfel.
Der Informationsgehalt ist vor dem Testen:

$$0.75log_2(1,333) + 0.25log_2(4) \approx 0,311 + 0,5 = 0,811$$

Nach dem Testen des Attributs „Geschmack" ergibt sich:

$$\begin{aligned} & \frac{30}{40} * \left(\frac{30}{30}\log_2(1) + \frac{0}{30}\log_2(0)\right) \\ + \quad & \frac{10}{40} * \left(\frac{10}{10}\log_2(1) + \frac{0}{10}\log_2(0)\right) \\ = \quad & 0 \end{aligned}$$

d.h. Der Informationsgewinn ist maximal.
Im Falle, dass die Farbe getestet wird, ergibt sich:

$$\frac{14}{40} * 0,8631 + \frac{26}{40} * 0,7793) \approx 0,80867\ldots$$

D.h. ein minimaler Informationsgewinn ist vorhanden. Der kommt nur aus der leicht unterschiedlichen Verteilung der guten Äpfel innerhalb der roten und grünen Äpfel und innerhalb aller Äpfel. Genauer gesagt: der Gewinn kommt daher, dass die Beispielmenge der 40 Äpfel nicht genau die Wahrheit abbildet.

Wird die Wahrheit richtig abgebildet, d.h. sind die Verteilungen gleich, dann:

süß,rot	*süß,grün*	*sauer,rot*	*sauer,grün*
10	*20*	*3*	*6*

Dann ergibt sich 0.7783 als Entropie $I(M)$ und auch für die Auswahl der Farbe danach, d.h. es gibt keinen Informationsgewinn für das Attribut Farbe.

Beispiel 3.3.8. *Wir erweitern das Beispiel der einfachen Äpfel um eine Apfelnummer. Der Einfachheit halber gehen die Nummern gehen von 1 bis 4. Zu beachten ist, dass dieses Attribut eine Besonderheit hat: es kann nicht der ganze Tupelraum ausgeschöpft werden, da es ja zu jeder Nummer nur einen Apfel geben soll. Das spiegelt sich auch in den prototypischen Beispielen:*

Es gibt vier Äpfel, $\{(1, süß, rot), (2, süß, grün), (3, sauer, rot), (4, sauer, grün)\}$. Wir geben als Beispiel vor, dass die guten Äpfel gerade $\{(1, süß, rot), (2, süß, grün)\}$ sind. Wir rechnen den Informationsgewinn der drei Attribute aus.

Der Informationsgehalt $I(M)$ vor dem Testen eines Attributes ist:

$$0.5log_2(2) + 0.5log_2(2) = 1$$

Nach dem Testen des Attributes „Geschmack" ergibt sich als Informationsgehalt $0,5 * (log_2(1) + 0) + 0,5 * (0 + log_2(1)) = 0$, d.h. Der Informationsgewinn ist maximal.

Nach dem Testen des Attributes „Farbe" ergibt sich als Informationsgehalt $0,5 * (0,5log_2(2) + 0,5log_2(2)) + 0,5 * (0,5log_2(2) + 0,5log_2(2)) = 0,5 * 1 + 0,5 * 1 = 1$, d.h. man gewinnt nichts.

Nach dem Testen des Attributs „Nummer" ergibt sich $\sum 1 * log(1)$, somit insgesamt 0. Der Informationsgewinn ist ebenfalls maximal. Weiter unten werden wie sehen, dass der Informationsgewinn in diesen sinnlosen Fällen durch Normierung kleiner wird.

Beispiel 3.3.9. *Wir nehmen als erweitertes Beispiel für Äpfel und die Attribute Geschmack $\in \{süß, sauer\}$ und Farbe $\in \{rot, gelb, grün\}$. Das Konzept sei „schmeckt-wie-er-aussieht". Es gibt sechs Kombinationen der Attribute: $\{(süß, rot), (süß, grün), (süß, gelb), (sauer, rot), (sauer, grün), (sauer, gelb)\}$. Wir geben als Beispiel die Menge $\{(süß, rot), (sauer, grün), (süß, gelb), (sauer, gelb)\}$ vor. Wir berechnen den Informationsgewinn bei beiden Attributen:*

Der Informationsgehalt $I(M)$ vor dem Testen irgendeines Attributs ist:

$$4/6 * log_2(6/4) + 1/3log_2(3) = 0.9183$$

$$4/6 * log_2(6/4) + 1/3log_2(3) = 0.9183$$

Nach dem Testen des Attributes „Geschmack" ergibt sich als Informationsgehalt $I(süss) = I(sauer) \approx 0.9183$. Die Gesamtinformation nach Abfrage des Attributs „Geschmack" ist: $0.5 * 0.9183 + 0.5 * 0.9183 = 0.9183$.

d.h. Der Informationsgewinn ist null.

Nach dem Testen des Attributes „Farbe" ergibt sich als I(grün) = I(rot) = 1, I(gelb) = 0. Die Gesamtinformation nach Abfrage der Farbe ist: $1/3 * 1 + 1/3 * 1 = 2/3 \approx 0.667$. *D.h. man hat Information gewonnen. Im Endeffekt muss man bei diesem Beispiel doch beide Attribute abfragen*

Man kann das Verfahren auch für reellwertige Attribute verwenden, wobei man als Grenzabfrage „$> w?$" nur endlich viele Werte ausprobieren muss, die sich aus den Werten der entsprechenden Attribute in den Beispielen ergeben. Es ist klar, dass ein Konzept wie „Fieber" aus den aktuell gegebenen Temperaturen und der Klassifizierung Fieber j/n nur annähernd gelernt werden kann.

Die Methode ID3 funktioniert recht gut, aber wenn ein Attribut zuviele Ausprägungen hat, wird die Frage nach diesem Attribut bevorzugt, da es im Extremfall (Personalnummer. o.ä.) dazu kommen kann, dass die Mengen $\{m \in M \mid m(a) = v\}$ einelementig werden, und somit der Informationsgewinn maximal ist.

Als Anmerkungen zum Entscheidungsbaumlernen:
Es ist auf jeden Fall besser als direkt die Elementbeziehung in der Menge aller positiven abzufragen, da das Programm sehr viel Platz brauchen könnte: so groß wie der gesamte Objektraum. Der Entscheidungsbaum kann exponentiell groß sein, wenn in jedem Pfad alle Attribute abgefragt werden.
Aber: bei der Konstruktion über eine Trainingsmenge mit ID3 wird der Entscheidungsbaum nicht gößsser als die Trainingsmenge, da diese bei jeder Konstruktion von Zwischenknoten kleiner wird.

3.3.2.1 C4.5 als verbesserte Variante von ID3

Das von Quinlan vorgeschlagene System C4.5 benutzt statt des Informationsgewinns einen normierten Informationsgewinn, wobei der obige Wert durch die Entropie des Attributs (d.h. der Verteilung bzgl. der Attributwerte) dividiert wird. Somit vergleicht man Attribute anhand

$$\text{Informationsgewinn} * \text{Normierungsfaktor} \qquad D.h.$$

$$\big(I(M) - I(M \mid a)\big) * \text{Normierungsfaktor}$$

Das bewirkt, dass Attribute mit mehreren Werten nicht mehr bevorzugt werden, sondern fair mit den zweiwertigen Attributen verglichen werden. Ohne diese Normierung werden mehrwertige Attribute bevorzugt, da diese implizit mehrere Ja/Nein-Fragen stellen dürfen, während ein zweiwertiges Attribut nur einer Ja/Nein-Frage entspricht. Dieser Vorteil wird durch den Normierungsfaktor ausgeglichen, der den Informationsgewinn auf binäre Fragestellung normiert, d.h. dass man den Informationsgewinn durch ein Attribut mit 4 Werten durch 2 dividiert, da man 2 binäre Fragen dazu braucht.

Der Normierungsfaktor für ein Attribut a mit den Werten $w_i, i = 1, \ldots, k$ ist:

$$\frac{1}{\sum_{i=1}^{k} P(a = w_i) * \log_2(\frac{1}{P(a = w_i)})}$$

Bei einem Booleschen Attribut, das gleichverteilt ist, ergibt sich als Normierungsfaktor $0,5 * 1 + 0,5 * 1 = 1$, während sich bei einem Attribut mit n Werten, die alle gleichverteilt sind, der Wert

$$\frac{1}{n * \frac{1}{n} * \log_2(n)} = \frac{1}{\log_2(n)}$$

ergibt.

Durch diese Vorgehensweise wird die Personalnummer und auch die Apfelnummer als irrelevantes Attribut erkannt. Allerdings ist es besser, diese Attribute von vorneherein als irrelevant zu kennzeichnen, bzw. erst gar nicht in die Methode einfließen zu lassen.

Beispiel 3.3.10. *Im Apfelbeispiel s.o. ergibt sich bei Hinzufügen eines Attributes Apfelnummer mit den Ausprägungen* $1, 2, 3, 4$, *als Normierungsfaktor für Apfelnummer:*

$$\frac{1}{\frac{1}{4} * 2 + \ldots + \frac{1}{4} * 2} = 0.5$$

Damit wird die Abfrage nach dem Geschmack vor der Apfelnummer bevorzugt.

3.3.2.2 Übergeneralisierung (Overfitting)

Tritt auf, wenn die Beispiele nicht repräsentativ sind, oder nicht ausreichend. Der Effekt ist, dass zwar die Beispiele richtig eingeordnet werden, aber der Entscheidungsbaum zu fein unterscheidet, nur weil die Beispiele (zufällig) bestimmte Regelmäßigkeiten aufweisen.

Beispiel 3.3.11. *Angenommen, man will eine Krankheit als Konzept definieren und beschreibt dazu die Symptome als Attribute:*
Fieber: Temperatur, Flecken: j/n, Erbrechen: j/n, Durchfall: j/n, Dauer der krankheit: Zeit, Alter des Patienten, Geschlecht des Patienten, ...
Es kann dabei passieren, dass das Lernverfahren ein Konzept findet, dass beinhaltet, dass Frauen

zwischen 25 und 30 Jahren diese Krankheit nicht haben, nur weil es keine Beispiele dafür gibt. Auch das ist ein Fall von overfitting.

Besser wäre es in diesem Fall, ein Datenbank aller Fälle zu haben. Die Erfahrung zeigt aber, dass selbst diese Datenbank aller Krankheiten für zukünftige Fragen oft nicht ausreicht, da nicht jede Frage geklärt werden kann: z.B. Einfluss des Gendefektes XXXXX auf Fettsucht.

Abschneiden des Entscheidungsbaumes: Pruning

Beheben kann man das dadurch, dass man ab einer gewissen Schranke den Entscheidungsbaum nicht weiter aufbaut, und den weiteren Aufbau an diesem Knoten stoppt: **Abschneiden des Entscheidungsbaumes (Pruning)**.

Wenn kein Attribut mehr einen guten Informationsgewinn bringt, dann besteht der Verdacht, dass alle weiteren Attribute eigentlich irrelevant sind, und man das Verfahren an dem Blatt stoppen sollte. Dies kann man bei bekannter Verteilung mittels eines statistischen Test abschätzen.

Hierbei ist es i.a. so, dass es an dem Blatt, an dem abgebrochen wird, noch positive und negative Beispiele gibt. Die Markierung des Knoten wählt man als Ja, wenn es signifikant mehr positive als negative Beispiel gibt, und als Nein, wenn es signifikant mehr negative als positive Beispiel gibt. Das ist natürlich nur sinnvoll, wenn man weiß, das es falsche Beispiele geben kann.

Hat man verrauschte Daten, z.B. mit Messfehler behaftete Beispiele, dann ist Lernen von Entscheidungsbäumen mit Pruning die Methode der Wahl.

3.4 Versionenraum-Lernverfahren

Wir betrachten ein weiteres Verfahren zum Lernen eines Konzepts über Objekten mit mehrwertigen Attributen, wobei wir davon ausgehen, dass Objekte durch diskrete Attribute und deren Werte beschrieben werden, aber das die Konzeptsprache noch Subsumtionsalgorithmen auf den Konzeptbeschreibungen zur Verfügungstellt. D.h. es gibt Algorithmen, die die Teilmengenbeziehung von Konzepten entscheiden können.

Wir stellen Objekte in Tupelschreibweise dar $(w_1, \ldots, w_n)$, wobei w_i der Wert des i. Attributs A_i ist. Betrachte z.B. Äpfel mit den Attributen Farbe (mit den Ausprägungen rot, grün, gelb), Geschmack (mit den Ausprägungen süß, sauer), Herkunft (mit den Ausprägungen Deutschland, Spanien, Italien) und Größe (mit den Ausprägunen S,M,L). Dann stellen wir einen roten, süßen Apfel aus Spanien mittlerer Größe als (rot, süß, Spanien, M) dar.

Hypothesen (und auch Konzepte) sind Mengen von Objekten repräsentiert durch die folgende Syntax: ⟨ rot, süß, Spanien, M ⟩ ist das Konzept, dass genau das Objekt (rot, süß, Spanien, M) enthält (also die einelementige Menge {(rot, süß, Spanien, M)}).

Wir erlauben anstelle eines Attributwerts auch ein Fragezeichen ?. So repräsentiert z.B. ⟨ rot, süß, ?, M ⟩ die Menge aller roten und süßen Äpfel mittlerer Größe (unabhängig von ihrer Herkunft). Sei h eine Hypothese und e ein Objekt, dann schreiben wir $e \in h$ falls e in

der Menge der Objekte liegt, die durch h repräsentiert werden. Zusätzlich erlauben wir noch $\emptyset$ als Zeichen für einen Attributwert. Sobald ein solcher Wert im Tupel enthalten ist, repräsentiert es das leere Konzept (Hypothese) welches keine Objekte enthält[1].

Auf den Konzepten / Hypothesen ist eine Ordnung $\leq$ definiert, die Spezialisierung bzw. Generalisierung ausdrückt. z.B. gilt $\langle \emptyset, \text{sauer} \rangle < \langle ?, \text{sauer} \rangle < \langle ?, ? \rangle$ Diese Ordnung ist i.a. nicht total.

Sei B eine Menge von Beispielen (Objekten) und h eine Hypothese. Dann nennen wir h konsistent für B wenn für alle $b \in B$ gilt:

- Wenn b positives Beispiel ist, dann gilt $b \in h$.
- Wenn b negatives Beispiel ist, dann gilt $b \notin h$.

Das Versionenraum-Lernverfahren ist ein inkrementelles Lernverfahren, d.h. die positiven und negativen Beispiele werden einzeln abgearbeitet und die Hypothese jeweils verfeinert.

Die Grundidee dabei ist recht einfach:

Aus der Menge aller Hypothesen, merke als Zustand, die maximale Menge der Hypothesen, die den bisherig gesehenen Beispielen nicht widerspricht. D.h. die Menge enthält die Hypothesen, die konsistent mit den bisherigen Beispielen sind und maximal ist: Alle bisher gesehenen *positiven* Beispiele müssen in jeder Hypothese der Menge enthalten sein, alle bisher gesehenen *negativen* Beispiele sind in keiner Hypothese der Menge enthalten.

Die so beschriebene Menge nennt man *Versionenraum*. Das zugehörige Lernverfahren verwaltet den Versionenraum und nimmt Anpassungen für jedes neue Beispiel vor. Das Verfahren ist inkrementell, da alte Beispiele nie wieder betrachtet werden müssen.

Zur kompakten Repräsentation des Versionenraums werden zwei Mengen von Hypothesen verwaltet:

- die untere Grenze des Versionenraums S (die speziellsten Hypothesen): Für jede Hypothese $h \in S$ gilt: h ist konsistent für die Menge der bisher gesehenen Beispiele und es gibt kein h' mit $h' < h$ das konsistent für die Menge der bisher gesehenen Beispiele ist.
- die obere Grenze des Versionenraums G (die allgemeinsten Hypothesen): Für jede Hypothese $h \in G$ gilt: h ist konsistent für die Menge der bisher gesehenen Beispiele und es gibt kein h' mit $h' > h$ das konsistent für die Menge der bisher gesehenen Beispiele ist.

Wenn $S = \{s_1, \ldots, s_n\}$ und $G = \{g_1, \ldots, g_m\}$, dann ist der Versionenraum genau: $\{h \mid \exists i, j : s_i \leq h \leq g_j\}$.

Bei Eingabe eines neuen Beispiels e werden die Mengen S und G neu berechnet:

Wenn e ein positives Beispiel ist:

[1] Es wäre hier auch möglich ein Symbol für das leere Konzept zu verwenden, anstatt die Kennzeichnung am Attributwert vorzunehmen

- Die Hypothesen $h \in S \cup G$ mit $e \in h$ müssen nicht verändert werden.
- Für jedes $h \in S$ mit $e \notin h$: h ist zu speziell und muss verallgemeinert werden. Ersetze h durch alle h' wobei
  - h' ist eine minimale Verallgemeinerung von h bzgl. e , d.h. $h' > h$, $e \in h'$ und es gibt keine Verallgemeinerung h'' von h bzgl. e mit $h' > h''$.
  - Zudem werden nur solche h' eingefügt, die nicht zu allgemein sind, d.h. es muss eine Hypothese $g \in G$ geben mit $g \geq h'$.

  Schließlich (um S kompakt zu halten): Entferne alle Hypothesen aus S die echte allgemeiner sind als andere Hypothesen aus S
- Für jedes $h \in G$ mit $e \notin h$: h ist zu speziell aber h kann nicht verallgemeinert werden (da $h \in G$). Daher: Lösche h aus G.

Wenn e ein negatives Beispiel ist:

- Die Hypothesen $h \in S \cup G$ mit $e \notin h$ müssen nicht verändert werden.
- Für jedes $h \in S$ mit $e \in h$: h ist zu allgemein aber h kann nicht spezialisiert werden (da $h \in S$). Daher: Lösche h aus S.
- Für jedes $h \in G$ mit $e \in h$: h ist zu allgemein und muss spezialisiert werden. Ersetze h durch alle h' wobei
  - h' ist eine minimale Spezialisierung von h bzgl. e, d.h. $h' < h$, $e \notin h'$ und es gibt keine Spezialisierung h'' von h bzgl. e mit $h'' > h'$.
  - Zudem werden nur solche h' eingefügt, die nicht zu speziell sind, d.h. es muss eine Hypothese $s \in S$ geben mit $s \leq h'$.

  Schließlich (um G kompakt zu halten): Entferne alle Hypothesen aus G die echte spezieller sind als andere Hypothesen aus G

Das Versionenraum-Lernverfahren startet mit der größtmöglichen Menge uns setzt daher:

- S enthält die speziellste Hypothese: $S = \{\langle \emptyset ?, \ldots, ? \emptyset \rangle\}$
- G enthält die allgemeinste Hypothese: $G = \{\langle ?, \ldots, ? \rangle\}$

Mögliche Ausgänge des Algorithmus sind:

- Wenn $S = G$ und S, G einelementig, d.h. $S = G = \{h\}$ dann ist h das gelernte Konzept
- Wenn S oder G ist die leere Menge: In diesem Fall ist der Versionenraum kollabiert. D.h. es gibt keine konsistente Hypothese für die vorgegebenen Trainingsbeispiele.

- S und G sind nicht leer aber auch nicht einelementig und gleich: Der aufgespannte Versionenraum enthält das Konzept aber dieses ist nicht eindeutig identifizierbar. D.h. alle Hypothesen im Versionenraum sind konsistent.

Beispiel 3.4.1. *Wir betrachten die Apfel-Objekte und die folgende Beispielmenge:*
(*süß,rot,Italien,L*)$^+$
(*süß,gelb,Spanien,S*)$^-$
(*süß,gelb,Spanien,L*)$^+$
(*sauer,rot,Deutschland,L*)$^-$
Hierbei sind die $^+$ *markierten Beispiele positiv und die mit* $^-$ *markierten Beispiele negativ.*
Das Versionenraum-Lernverfahren startet mit

$$S = \{\langle \emptyset, \emptyset, \emptyset, \emptyset \rangle\}$$
$$G = \{\langle ?, ?, ?, ? \rangle\}$$

1. *Einfügen des Beispiels* (*süß,rot,Italien,L*)$^+$*:*
   - *Da* (*süß,rot,Italien,L*) $\in \langle ?, ?, ?, ? \rangle$*, wird die Menge G nicht verändert.*
   - *Da* (*süß,rot,Italien,L*) $\notin \langle \emptyset, \emptyset, \emptyset, \emptyset \rangle$ *muss diese Hypothese der Menge S verallgemeinert werden. Die minimale Verallgemeinerung ist* $\langle$*süß,rot,Italien,L*$\rangle$ *und daher wird S neu gesetzt als* $S = \{\langle$*süß,rot,Italien,L*$\rangle\}$.

2. *Einfügen des Beispiels* (*süß,gelb,Spanien,S*)$^-$*:*
   - *Da* (*süß,gelb,Spanien,S*) $\notin \langle$*süß,rot,Italien,L*$\rangle$*, wird S nicht verändert.*
   - *Da* (*süß,gelb,Spanien,S*) $\in \langle ?, ?, ?, ? \rangle$*, muss diese Hypothese aus G spezialisiert werden. Minimale Spezialisierungen sind:*
     $\langle$*sauer*$, ?, ?, ?\rangle$ $\langle ?$*,rot*$, ?, ?\rangle$ $\langle ?$*,gruen*$, ?, ?\rangle$ $\langle ?, ?$*,Italien*$, ?\rangle$
     $\langle ?, ?$*,Deutschland*$, ?\rangle$ $\langle ?, ?, ?$*,L*$\rangle$ $\langle ?, ?, ?$*,M*$\rangle$

     Davon sind jedoch einige zu speziell, da es keine Hypothese $s \in S$ *gibt, die spezieller oder gleich ist. Daher verbleiben nur* $\langle ?$*,rot*$, ?, ?\rangle$*,* $\langle ?, ?$*,Italien*$, ?\rangle$ *und* $\langle ?, ?, ?$*,L*$\rangle$*. Wir setzen daher* $G = \{\langle ?$*,rot*$, ?, ?\rangle, \langle ?, ?$*,Italien*$, ?\rangle, \langle ?, ?, ?$*,L*$\rangle\}$.

3. *Einfügen des Beispiels* (*süß,gelb,Spanien,L*)$^+$*:*
   - *Da* (*süß,gelb,Spanien,L*) $\notin \langle ?$*,rot*$, ?, ?\rangle \in G$ *muss die Hypothese gelöscht werden.*
   - *Da* (*süß,gelb,Spanien,L*) $\notin \langle ?, ?$*,Italien*$, ?\rangle \in G$ *muss die Hypothese gelöscht werden.*
   - *Da* (*süß,gelb,Spanien,L*) $\in \langle ?, ?, ?$*,L*$\rangle \in G$ *bleibt Hypothese in G.*
   - *Ergibt* $G = \{\langle ?, ?, ?$*,L*$\rangle\}$
   - *Da* (*süß,gelb,Spanien,L*) $\notin \langle$*süß,rot,Italien,L*$\rangle \in S$ *muss die Hypothese verallgemeinert werden. Die minimale Verallgemeinerung ist:* $\langle$*süß,?,?,L*$\rangle$ *und daher* $S = \{\langle$*süß,?,?,L*$\rangle\}$

4. *Einfügen des Beispiels* $(sauer,rot,Deutschland,L)^-$*:*
   - *Da* $(sauer,rot,Deutschland,L) \notin \langle süß,?,?,L\rangle$*, bleibt* S *unverändert.*
   - *Da* $(sauer,rot,Deutschland,L) \in \langle ?,?,?,L\rangle$*, muss die Hypothese spezialisiert werden.Minimale Spezialiserungen sind:*
     $\langle süß,?,?,L\rangle$ $\langle ?,gelb,?,L\rangle$ $\langle ?,grün,?,L\rangle$
     $\langle ?,?,Italien,L\rangle$ $\langle ?,?,Spanien,L\rangle$
     jedoch ist nur $\langle süß,?,?,L\rangle$ *nicht zu speziell. Daher* $G = \langle süß,?,?,L\rangle$

Alle Beispiele sind abgearbeitet und es gilt $S = G = \{\langle süß,?,?,L\rangle\}$*. Daher ist* $\langle süß,?,?,L\rangle$ *das gelernte Konzept.*

Beispiel 3.4.2. *Betrachte als weiteres Beispiel die Menge:*
$(süß,rot,Italien,L)^+$
$(süß,gelb,Italien,L)^-$
Das Versionenraum-Lernverfahren startet mit

$$\begin{aligned} S &= \{\langle \emptyset,\emptyset,\emptyset,\emptyset\rangle\} \\ G &= \{\langle ?,?,?,?\rangle\} \end{aligned}$$

1. *Einfügen des Beispiels* $(süß,rot,Italien,L)^+$*:*
   - *Da* $(süß,rot,Italien,L) \in \langle ?,?,?,?\rangle$*, wird die Menge* G *nicht verändert.*
   - *Da* $(süß,rot,Italien,L) \notin \langle \emptyset,\emptyset,\emptyset,\emptyset\rangle$ *muss diese Hypothese der Menge* S *verallgemeinert werden. Die minimale Verallgemeinerung ist* $\langle süß,rot,Italien,L\rangle$ *und daher wird* S *neu gesetzt als* $S = \{\langle süß,rot,Italien,L\rangle\}$*.*

2. *Einfügen des Beispiels* $(süß,gelb,Italien,L)^-$*:*
   - *Da* $(süß,gelb,Italien,L) \notin \langle süß,rot,Italien,L\rangle$ *wird* S *nicht verändert.*
   - *Da* $(süß,gelb,Italien,L) \in \langle ?,?,?,?\rangle$*, muss die Hypothese spezialisiert werden.*
     Minimale Spezialisierungen sind:
     $\langle sauer,?,?,?\rangle$ $\langle ?,grün,?,?\rangle$ $\langle ?,rot,?,?\rangle$ $\langle ?,?,Deutschland,?\rangle$
     $\langle ?,?,Spanien,?\rangle$ $\langle ?,?,?,S\rangle$ $\langle ?,?,?,M\rangle$
     Davon verbleibt nur $\langle ?,rot,?,?\rangle$*, da die anderen zu speziell sind. Daher setzen wir* $G = \{\langle ?,rot,?,?\rangle\}$*.*

Alle Beispiele wurden abgearbeitet und

$$\begin{aligned} S &= \{\langle süß,rot,Italien,L\rangle\}. \\ G &= \{\langle ?,rot,?,?\rangle\} \end{aligned}$$

Das bedeutet, dass alle Hypothesen h *mit*

$$\langle süß,rot,Italien,L\rangle \leq h \leq \langle ?,rot,?,?\rangle$$

konsistent mit der Beispielmenge sind.

Beispiel 3.4.3. *Betrachte das vorherige Beispiel, erweitert um ein weiteres Beispiel:*
(süß,rot,Italien,L)$^+$
(süß,gelb,Italien,L)$^-$
(süß,grün,Italien,L)$^+$
Die Abarbeitung der ersten beiden Beispiele resultiert in:
$S = \{\langle$*süß,rot,Italien,L*$\rangle\}$.
$G = \{\langle ?,$*rot*$, ?, ?\rangle\}$
Wir betrachten das Einfügen von (süß,grün,Italien,L)$^+$*:*

- *Da (süß,grün,Italien,L)* $\notin \langle ?,$*rot*$, ?, ?\rangle$ *und die Hypothesen in G nicht verallgemeinert werden dürfen, wird die Hypothese gelöscht. D.h.* $G = \emptyset$
- *Da (süß,grün,Italien,L)* $\notin \langle$*süß,rot,Italien,L*$\rangle$*, muss das Konzept verallgemeinert werden zu* $\langle$*süß,?,Italien,L*$\rangle$*. Da es in G jedoch keine Konzepte gibt, ist das Konzept (wie jegliches Konzept in diesem Fall) zu allgemein und wird gelöscht. Daher setze* $S = \emptyset$*.*

Da $G = S = \emptyset$ *gibt es keine konsistente Hypothese für die Beispielmenge. Der Grund liegt darin, dass die beiden Beispiele (süß,rot,Italien,L)*$^+$ *und (süß,grün,Italien,L)*$^+$ *als minimales Konzept das Konzept* $\langle$*süß,?,Italien,L*$\rangle$ *erfordern. Aber dieses minimale Konzept bereits das negative Beispiel (süß,gelb,Italien,L)*$^-$ *beinhaltet.*
Abbhilfe kann hier nur eine mächtigere Konzeptsprache schaffen.

Ein weiteres Problem der Versionenraum-Methode ist, dass sie nicht mit verrauschten Daten umgehen kann. In diesem Fall kann sie keine konsistente Hypothese finden.

Man kann Varianten der Konzeptsprachen verwenden die ausdruckstärker sind. Z.B. ist eine ausdrucksstärkere Konzeptsprache:

- Erlaube „$a = M_a$“ für Attribute a, wobei M_a eine Teilmenge der möglichen Ausprägungen von a ist ist. Damit kann man Quader im Objektraum erzeugen. Diese Sprache nennen wir *Quader-Konzepte*.
- Erlaube Disjunktionen der Quader-Konzepte. Damit kann man bereits alle Konzepte darstellen, wenn die Menge der Attribute und Ausprägungen endlich ist.

4

Einführung in das Planen

4.1 Planen

Beim Planen werden Aufgaben der folgenden Form betrachtet und gelöst:
Finde eine Folge von Aktionen, um aus einer gegebenen Anfangssituation eine Zielsituation zu erreichen, wobei die Effekte von Aktionen bekannt sind.
Z.B. Handlungsplan für einen Roboter auf einer abstrakten Ebene.

4.1.1 Planen in der Klötzchen-Welt

Eine einfache Modellwelt, in der man das Vorgehen und die Problematik auf der Planungsebene untersuchen kann, ist die **Blockwelt**, oder auch Klötzchenwelt.
Folgende Annahmen gelten

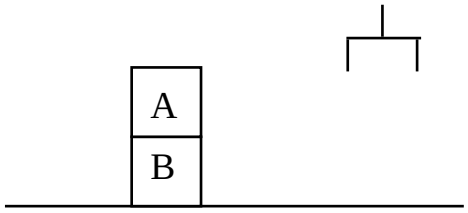
- Auf einem Tisch stehen Würfel -Klötzchen auf bzw. nebeneinander. Für alle Würfel ist genug Platz vorhanden.
- Es gibt einen Greifarm (Hand), der jeden (freien) Würfel hochheben und woanders hinstellen kann, d.h. entweder auf den Tisch, oder auf einen anderen Würfel.
- Ein Würfel steht entweder auf dem Tisch, oder auf einem anderen Würfel oder wird vom Greifarm gehalten.
- typische Operationen sind: HochHeben(x), Absetzen(x), Stapeln(x, y), VomStapel(x, y).

4.1.2 Situationslogik

Dies ist eine Methode, veränderliche Zustände eines Systems in Prädikatenlogik auszudrücken. Dazu wird jedes Prädikat, das veränderlich sein kann, um einen Situationsparameter ergänzt; man kann sich vorstellen, dass es eine diskrete Zeit ist.
Z.B. `StrasseNass(RobertMayerStr, S)` Hier ist `S` ein Situationsparameter.
Die Darstellung einer Situation der Klötzchenwelt in Prädikatenlogik kann so aussehen:

$$
\begin{aligned}
& \text{AufDemTisch}(B, S_0) \\
\wedge\ & \text{Auf}(A, B, S_0) \\
\wedge\ & \text{Hand}(\text{Leer}, S_0) \\
\wedge\ & \text{Frei}(A, S_0)
\end{aligned}
$$

Hierbei bezeichnen A, B Namen von Klötzchen, S_0 ist die aktuelle Situation.



Folgende Operatoren gibt es, um die Klötzchenwelt zu manipulieren:

VomStapel ist eine Operation, die einen Klotz von einem Stapel nimmt.
In der Situationslogik muss man dazu eine Formel (Axiom) hinschreiben, die den Effekt dieser Operation beschreibt:

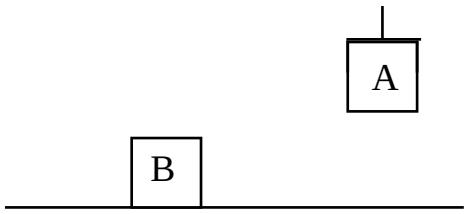
$\forall x, y, s$: Hand(Leer, s) $\wedge$ Frei(x, s) $\wedge$ Auf(x, y, s)
$\Rightarrow$

$$\begin{aligned} & \text{Hand}(x, \text{VomStapel}(x, y, s)) \\ \wedge\ & \text{Frei}(y, \text{VomStapel}(x, y, s)) \\ \wedge\ & \neg\text{Hand}(\text{Leer}, \text{VomStapel}(x, y, s)) \\ \wedge\ & \neg\text{Frei}(x, \text{VomStapel}(x, y, s)) \\ \wedge\ & \neg\text{Auf}((x, y, \text{VomStapel}(x, y, s))) \end{aligned}$$

Hier ist s die Situation vor Ausführen der Operation und VomStapel(x, y, s) die Situation nach Ausführen der Operation.

Wenn eine Situation S_0 gegeben war, und $S_1 :=$ VomStapel(A, B, S_0),
dann erhalten wir als neue Fakten:

Hand(A, S_1)
Frei(B, S_1)
$\neg$Hand(Leer, S_1)
$\neg$Frei(A, S_1)
$\neg$Auf(A, B, S_1)



Was ist mit dem AufDemTisch(B, S_1)?
D.h. welchen Wahrheitswert hat die Aussage AufDemTisch(B, S_1) im Zustand S_1?
Leider kann man das aus den bisherigen Axiomen und der Beschreibung nicht folgern!

Man bräuchte: (sog. **Rahmenaxiome,successor-state axioms**), die pro Aktion exakt sagen, was sich alles durch diese Aktion **nicht** ändert.

$$\forall x, y, z, s : \text{AufDemTisch}(z, s) \wedge z \neq x$$
$$\Rightarrow$$
$$\text{AufDemTisch}(z, \text{VomStapel}(x, y, s))$$

4.1.3 Das Rahmenproblem (Frame problem)

Wie beschreibt und behandelt man die Aspekte der Welt, die sich unter Operatoren nicht ändern?

Man kann das Problem noch unterscheiden in

- Rahmenproblem der Repräsentation (Sprache).
- Rahmenproblem der Inferenz.

In der Prädikatenlogik bzw. Situationslogik tritt das Rahmenproblem in beiden Varianten explizit auf. Man muss sowohl bei der Formulierung als auch bei der Inferenz sehr viele Fakten beschreiben und mitführen, die sich nicht ändern.

Aber: Auch wenn man die PL1-Situationslogik verwirft, hat man das Rahmenproblem noch:

In jedem Planungsformalismus muss das Rahmenproblem beachtet und gelöst werden.

Die Formulierung in Prädikatenlogik ist praktisch zu ineffizient. Einerseits kann man das Problem der vielen Rahmenaxiome durch automatische Generierung oder durch automatisches Verwenden in Schlüssen beheben, aber die Schlussmechanismen die man so erhält, sind zu schwerfällig und ineffizient, denn man muss jeweils von der Anfangssituation aus für jede Aussage nachrechnen, ob sich durch die Aktion etwas geändert hat. Ein Problem, das man u.a. durch die Unübersichtlichkeit zusätzlich bekommt, ist die Unsicherheit, ob die Axiome die Modellwelt korrekt beschreiben und ob sie noch widerspruchsfrei sind. Wenn sie nicht widerspruchsfrei sind, kann man bekanntlich alles ableiten.

4.1.4 Qualifikationsproblem

Welche Eigenschaften muss man im Modell explizit beschreiben ? Welche Eigenschaften lässt man weg?

Dieses Problem steckt in jeder Modellierung, da diese nicht alle Aspekte der realen Welt erfassen kann. z.B. ungenau oder wegabstrahiert sind:

Größe der Blöcke
Gewicht der Blöcke
Kann die Hand etwas verlieren ?
Stapel fällt bei Sturm um,
der Greifarm greift manchmal ungenau zu,
...

D.h. Man kann die Wirklichkeit nicht exakt im Modell erfassen,
Jedes Modell abstrahiert bestimmte Eigenschaften.

Das sind eigentlich verschiedene Problematiken:

- Modelle abstrahieren immer von der realen Welt.
- Das gemeinte Problem ist das der „expliziten Programmierung “. Diese muss jeden Teilaspekt der realen Welt, der mit Programmen behandelt werden soll, schon im Voraus vollständig beschreiben und in ein Programm gießen. Das kann mit Methoden der Parameteranpassung bzw. mittels der Methoden des maschinellen Lernens abgemildert werden, aber (nach aktuellem Stand der Technik) nicht vermieden werden.
- Die logischen Modelle gehen von einer Übereinstimmung der Welt mit der internen Repräsentation aus. D.h. die Welt ist nicht wirklich extern. In realen Situation muss man stets davon ausgehen, dass die interne Repräsentation nicht bzw. nicht ganz mit der Welt übereinstimmt. D.h. man muss mit Sensoren kontrollieren, ob die Aktion erfolgreich war, oder ob sich nicht Klötzchen heimlich an einen anderen Ort begeben haben (z.B. Klötzchen-Turm fällt um).

4.2 STRIPS-Planen

Stanford **R**esearch **I**nstitute **P**roblem **S**olver

Situationen sind eine Konjunktion (Menge) von (positiven und negativen) Grund-Fakten ohne Variablen und ohne Funktionssymbole, die sich nicht widersprechen. Konstanten beschreiben die Objekte der betrachteten Welt. Es gilt die unique-names assumption: Verschiedene Namen bezeichnen verschiedene Objekte. Literale werden nur gebildet mit Prädikatensymbolen und Konstanten, jedoch werden keine Funktionssymbole verwendet.

Man kann auch alternativ die Closed-World Annahme machen, (d.h. die nicht in der Menge enthaltenen Fakten werden als falsch angesehen) und nur positive Fakten speichern. Es ist etwas effizienter beim Hinschreiben der Beispiele bzw beim Speichern, aber es macht keinen grundsätzlichen Unterschied.

Ein **Operator** hat folgende Komponenten:

- Es gibt eine Menge von Variablen (allquantifiziert)
- Es gibt einen Vorbedingungsteil (Konjunktion von Literalen, positiv oder negativ)
- Effekte der Regel: besteht aus positiven und negativen Fakten, die die entsprechenden Fakten in der Situation überschreiben sollen. Man kann das auch als Dazu- und der Lösch-liste gruppieren, wenn man nur mit positiven Fakten umgehen will.

Eine konkrete Anwendung eines Operators muss vollinstanziiert sein, d.h. alle Variablen sind durch Konstanten substituiert.

Ein linearisierter **Plan** ist eine Folge von vollinstanziierten Operatoren.

Die Anwendung eines vollinstanziierten Operators auf eine Situation S ist möglich, wenn die Vorbedingungen erfüllt sind in S, und ergibt in diesem Falle eine Nachfolgesituation. Wenn die Bedingungen nicht erfüllt sind, ist es ein Fail.

Die Anwendung eines Plans $P = O_1, \ldots, O_n$ auf eine Anfangssituation S_1 ist möglich, wenn es eine Folge $S_1, S_2, \ldots, S_{n+1}$ von Situation gibt, so dass O_i sich auf S_i anwenden lässt, und $O_i(S_i) = S_{i+1}$ für alle i. Die Situation S_{n+1} ist das Resultat.

Normalerweise ist eine Start S_0 und Zielsituation Z gegeben und es wird ein (vollinstanziierter, linearisierter) Plan P gesucht, der S_0 in Z überführt.

Wir beschreiben informell Beispiele für Operatoren in der Klötzchenwelt:

Aufheben(x):		
Vorbedingung:		x steht auf dem Tisch
		x ist frei
		die Hand hält nichts
Effekt:	Dazu:	die Hand hält x
	Löschen	x steht auf dem Tisch
		x ist frei
		die Hand hält nichts.

Hinstellen(x):		
Vorbedingung:		die Hand hält x
Effekt:	Dazu:	x steht auf dem Tisch
		x ist frei
		die Hand hält nichts
	Löschen	die Hand hält x

Stapeln(x,y):		
Vorbedingung:		die Hand hält x
		y ist frei
Effekt:	Dazu:	x steht auf y
		x ist frei
		die Hand hält nichts
	Löschen	die Hand hält x
		y ist frei

VomStapel(x,y):		
Vorbedingung:		x steht auf y
		x ist frei
		die Hand hält nichts
Effekt:	Dazu:	die Hand hält x
		y ist frei
	Löschen	x steht auf y
		x ist frei
		die Hand hält nichts

Die formale Beschreibung dieser Operatoren ist:

Aufheben(x):		
Vorbedingung:		$AufDemTisch(x)$
		$Frei(x)$
		$Hand(Leer)$
Effekt:	Dazu:	$Hand(x)$
	Löschen:	$AufDemTisch(x)$
		$Frei(x)$
		$Hand(Leer)$

Hinstellen(x):		
Vorbedingung:		$Hand(x)$
Effekt:	Dazu:	$AufDemTisch(x)$
		$Frei(x)$
		$Hand(Leer)$
	Löschen:	$Hand(x)$

Stapeln(x,y):		
Vorbedingung:		$Hand(x)$
		$Frei(y)$
Effekt:	Dazu:	$Auf(x,y)$
		$Frei(x)$
		$Hand(Leer)$
	Löschen:	$Hand(x)$
		$Frei(y)$

VomStapel(x,y):		
Vorbedingung:		$Auf(x,y)$
		$Frei(x)$
		$Hand(Leer)$
Effekt:	Dazu:	$Hand(x)$
		$Frei(y)$
	Löschen:	$Auf(x,y)$
		$Frei(x)$
		$Hand(Leer)$

4.2.1 Prozedur zum Planen

Idee: gehe rückwärts von einer Zielsituation aus (*Regressionsplaner*).

Gesucht ist ein Plan als Folge von Operator-Instanzen.

Die Prozedur $Plane$ ist rekursiv und hat als Argumente:

- Menge der noch zu erfüllenden Teilziele Z (Konjunktion von Grund-Literalen)
- Startzustand S (Konjunktion von Grund-Literalen)

$Plane(S, Z)$ (S: Situation, Z: Teilziele,)

- Wenn $Z = \emptyset$ oder $Z \subseteq S$ Return $\emptyset$
- wähle offenes Ziel Z_1 aus Z
- wähle Instanz O eines Operators, so dass Z_1 ein Effekt von O ist, und kein Effekt von O einem Teilziel aus Z widerspricht.
  Das Verfahren verbessert sich, wenn man Z insgesamt auf Konsistenz testet.
- Erzeuge Plan P zum Erfüllen der Vorbedingung vor O durch rekursiven Aufruf: $P := Plane(S, (Z \setminus \text{Effekte von } O) \cup \text{Vorbedingungen von } O)$.
- RETURN $P; O$

Beispiel 4.2.1. *Für einen Planungsablauf*

Anfangssituation: $Auf(B,A), Frei(B), Hand(Leer), Tisch(C), Tisch(A), Frei(C)$

Ziel: $Auf(A,B), Auf(B,C)$
Teilziel: $Auf(B,C)$
Operatorinstanz: $Stapeln(B,C)$
Vorbedingung: Hand(B), Frei(C)
rekursives Planen: Hand(B), Frei(C)
Erfülle: Hand(B)
Operatoren: $VomStapel(B,x)$ *oder* $Aufheben(B)$
usw.

Der Plan, der sich ergibt:
VomStapel(B,A), Stapeln(B,C), Aufheben(A), Stapeln(A,B).

4.2.2 Implizite Annahmen und Einschränkungen dieser Methode:

- Linearitätsannahme:
  Einem Plan liegt eine totale (lineare) zeitliche Ordnung zugrunde. Der Planungsalgorithmus benutzt diese.
- Lösung des Rahmenproblems in STRIPS:
  Jeder Operator ändert genau das, was als sein Effekt angegeben ist. Alle nicht angegebenen Fakten bleiben erhalten.
- In der einfachen angegeben Prozedur oben:
  das Ziel wird direkt angesteuert, das Erfinden von neuen Zwischenzielen ist nicht möglich.

4.2.3 Das Problem der Zwischenzielinteraktion (Sussman-Anomalie)

Dieses Problem ist bedingt durch die Erzeugung eines linearen Plans durch Regression. Insbesondere durch die rekursiv ausgeführte Planungsmethode der Erfüllung von Zwischenzielen, wenn Z als Stack implementiert ist.

Die Vorgehensweise ergibt für die im Bild dargestellt Situationsänderung immer zu umständliche Pläne, egal wie man die Selektion der Zwischenziele ausführt.

Die beste Variante ist:
VomStapel(C,A), Hinstellen(C), Hochheben(A),
Stapeln(A,B), VomStapel(A,B) Hinstellen(A),
Hochheben(B), Stapeln(B,C), Hochheben(A),
Stapeln(A,B).

Grund ist die Vorgehensweise der Methode:
1. zuerst ein Teilziel vollständig erfüllen,
2. dann nächstes Teilziel erfüllen.

Man kann dies akzeptieren und eventuell, wie in Kompilern eine „Gucklochoptimierung "nachschalten, um die Pläne nachträglich zu optimieren. D.h. man entfernt nutzlose Zwischenaktionen. Dieses Problem verschwindet beim Planen mittels partieller Ordnungen.

4.3 Planen mit partieller Ordnung: POP

Man geht von derselben Repräsentation der Situationen aus, aber verallgemeinert die Pläne, indem man statt Linearisierung nur noch eine partielle Ordnung der Operatorausführung verlangt. Das hat verschiedene Vorteile:

- Der Planalgorithmus ist nicht mehr gezwungen, eine genaue Ordnung festzulegen. Die könnte nämlich irrelevant sein. Bei einer Suche hat das den Effekt, dass man beim Fehlschlagen viele irrelevante Varianten probieren und ausschliessen muss, bevor man backtracking machen kann.

- Der fertige Plan erlaubt verschiedene Linearisierungen. Dies erhöht die Flexibilität. Aber der fertige Plan ist auch geeignet, parallel ausgeführt zu werden, wenn die Umgebung parallele Abarbeitung mit evtl. mehreren Greifarmen (Robotern, Prozessoren) zulässt.

Wir definieren einen Plan, dessen Aktionen partiell geordnet sind. Die Start und Zielsituation werden als Operatoren hinzugefügt, wobei nicht verlangt wird, dass diese vollständig spezifiziert sind.

Definition 4.3.1. *Ein* Plan *ist:*

- *Ein gerichteter Graph mit Knoten K_i.*

- *Die Startoperation ist die Wurzel, und die Zieloperation ist die Senke des gerichteten Graphen. Bedingungen an Ziel und Effekt am Start sind voll instanziiert.*

- *Jeder Knoten ist mit genau einem Operator markiert, die Variablen sind jeweils neu umbenannt.*

- *Eine Menge von Instanzbedingungen an Variablen: es darf nur $x = a$ zwischen Variable und Konstante vorkommen bzw. $x = y$ zwischen zwei Variablen.*

Wir schreiben $K_i \prec K_j$, wenn es einen Weg von K_i nach K_j gibt.

Ein Plan ist konsistent, *wenn der Graph azyklisch ist und die Instanzbedingungen nicht widersprüchlich sind; d.h. $x = a, x = b$ ist nicht daraus herleitbar.*

Eine Linearisierung eines Planes *ist eine Folge der Knoten, die kompatibel mit dem gerichteten Graphen ist. Normalerweise meint man dann nur die Folge der Operatoren.*

Ein konsistenter Plan P ist vollständig, *wenn jede Linearisierung auf jeder Situation ohne Fehlschlagen ausführbar ist.*

Ein Kriterium für Vollständigkeit ist:

Für jede Vorbedingung c eines Operators am Knoten K_i gibt es einen Knoten K_j, so dass $K_j \prec K_i$ gilt und K_j hat c als Effekt; und für alle weiteren Knoten K_k, die in einer Linearisierung zwischen K_j und K_i sein könnten, d.h. für die weder $K_k \prec K_j$ noch $K_i \prec K_k$ gilt, ist $\neg c$ nicht Effekt von K_k.

Dieses Kriterium garantiert, dass jede Vorbedingung erfüllt werden kann, ohne wieder zerstört zu werden.
Es garantiert auch die parallele und widerspruchsfreie Ausführbarkeit des Plans.

Dieses Kriterium ist nicht leicht zu testen, aber hat den Vorteil, dass man für die Pläne etwas mehr Freiheit hat, und nicht alles spezifizieren muss.

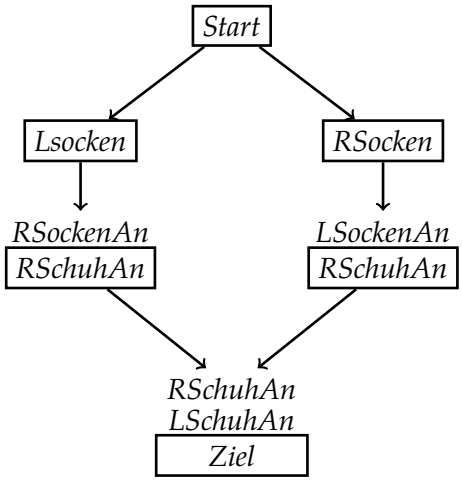
Man kann die gerichteten Kanten auch als erzeugende Relationen hinschreiben, die eine transitive Prä-ordnung erzeugen Man kann diese Bedingung auch explizit in eine Datenstruktur aufnehmen, indem man Unterstützungs-Links in den gerichteten Graphen einführt. Das kann allerdings zu einem größeren Suchraum führen.

Definition 4.3.2. *Ein Erweiterung die in Russel und Norvig betrachtet wird, ist die Möglichkeit, die Operatoren nicht voll zu instanziieren*

Beispiel 4.3.3. *Betrachte das einfache Beispiel, Socken und Schuhe anzuziehen. Die Operatoren sind:*

- *Start*
- *LSockenAnziehen. Effekt: LSockenAn*
- *RSockenAnziehen. Effekt: RSockenAn*
- *LSchuhAnziehen; Bedingung: LSockenAn; Effekt: LSchuhAn*
- *RSchuhAnziehen; Bedingung: RSockenAn; Effekt: RSchuhAn*
- *Ziel: Bedingung: LSchuhAn, RSchuhAn.*

Ein Plan dazu ist:



Der Plan kann offenbar auch parallel ausgeführt werden. Das Ziel spezifiziert nur einen Teil der Situation, nämlich $LSchuhAn, RSchuhAn$.

Er abstrahiert etwas von der Wirklichkeit, da er auch ausführbar ist, wenn man schon Socken und Schuhe an hat.

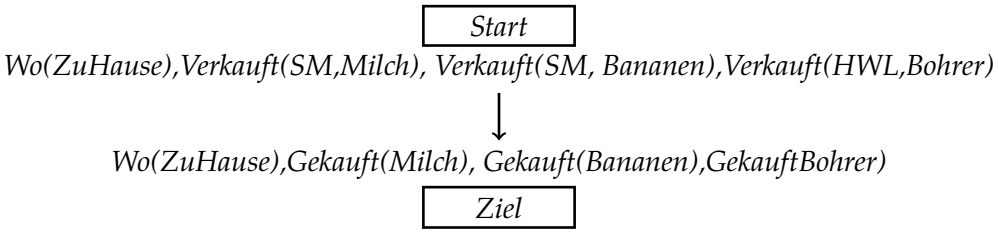
Im Buch von Russel und Norvig ist ein schönes Planungsbeispiel für ein Vorgehen bei Planen mittels partieller Ordnung.

Beispiel 4.3.4. *Die Aufgabe ist, Milch, Bananen, und einen Bohrer zu kaufen: Milch und Bananen gibt es im Supermarkt, den Bohrer im Heimwerkerladen. Dazu soll man einen Plan entwickeln, so dass man von zu hause startet, die beiden Geschäfte aufsucht, die Artikel kauft und wieder nach Hause geht.*

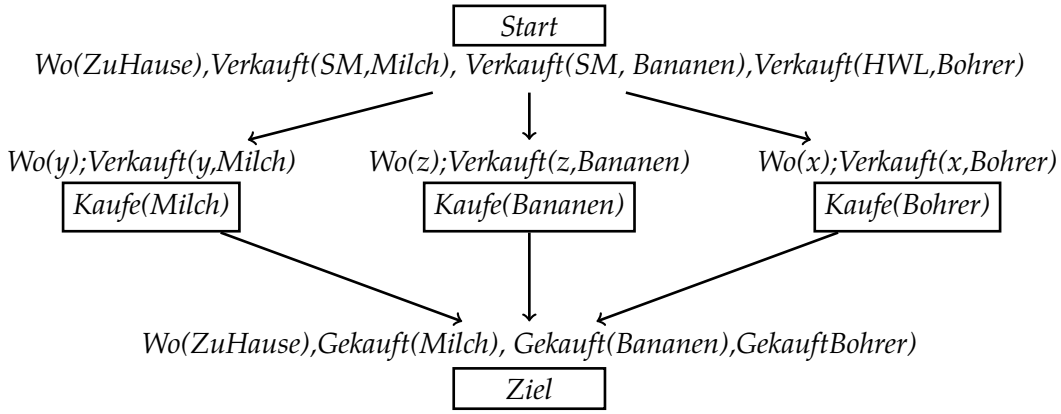
Konstanten fangen mit Großbuchstaben an, Variablen mit Kleinbuchstaben.

- *Start. Effekt: Wo(ZuHause), Verkauft(SM,Milch), Verkauft(SM,Bananen), Verkauft(HWL,Bohrer).*
- *Ziel. Bedingung: Gekauft(Milch), Gekauft(Bananen), Gekauft(Bohrer), Wo(ZuHause).*
- *Gehe(dort): Bedingung: Wo(hier), $hier \neq dort$; Effekt , $\neg$Wo(hier),Wo(dort).*
  Variablen sind $hier, dort$.
- *Kaufe(artikel): Bedingung: Wo(laden), Verkauft(laden,artikel); Effekt $Gekauft(artikel)$.*
  Variablen sind laden, artikel.

Die Planung startet mit einem konsistenten Plan, der nur aus Start und Ziel besteht, mit der Ordnungsbedingung: Start $\prec$ Ziel, und der noch nicht vollständig ist, da z.B. nicht alle Vorbedingungen des Ziels erfüllt werden.

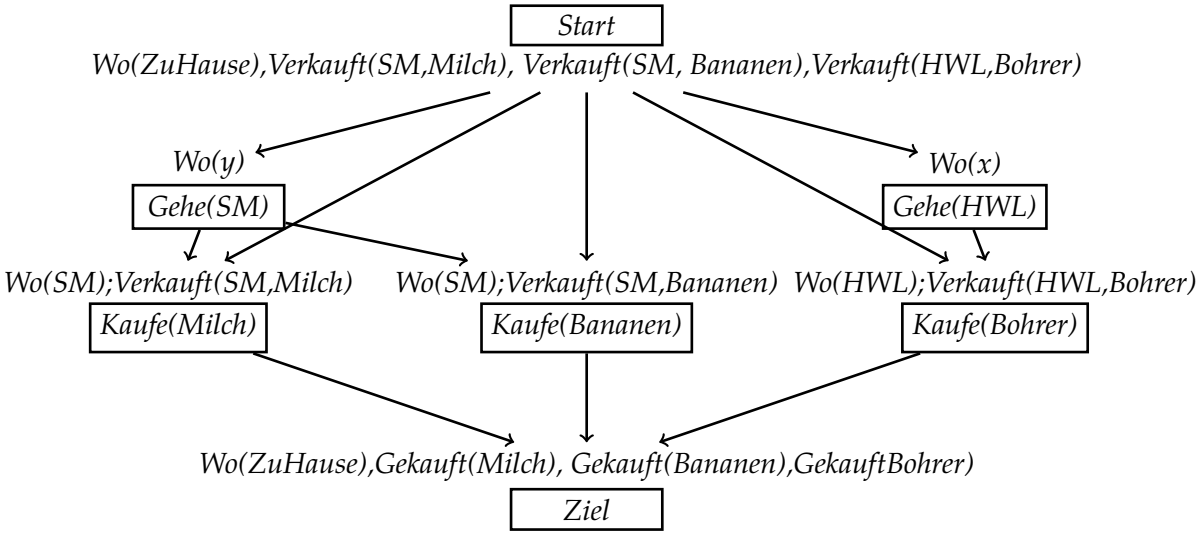


Nach Einfügen der Kaufoperatoren:



Die Variablen x, y, z sind noch nicht instanziiert. Sie können mit $x = HWL, y = SM, z = SM$ instanziiert werden,

Fügt man die notwendigen $Gehe$-Operatoren ein, so erhält man folgenden partiellen Plan, wobei die Variablen x, y noch nicht instanziiert sind.

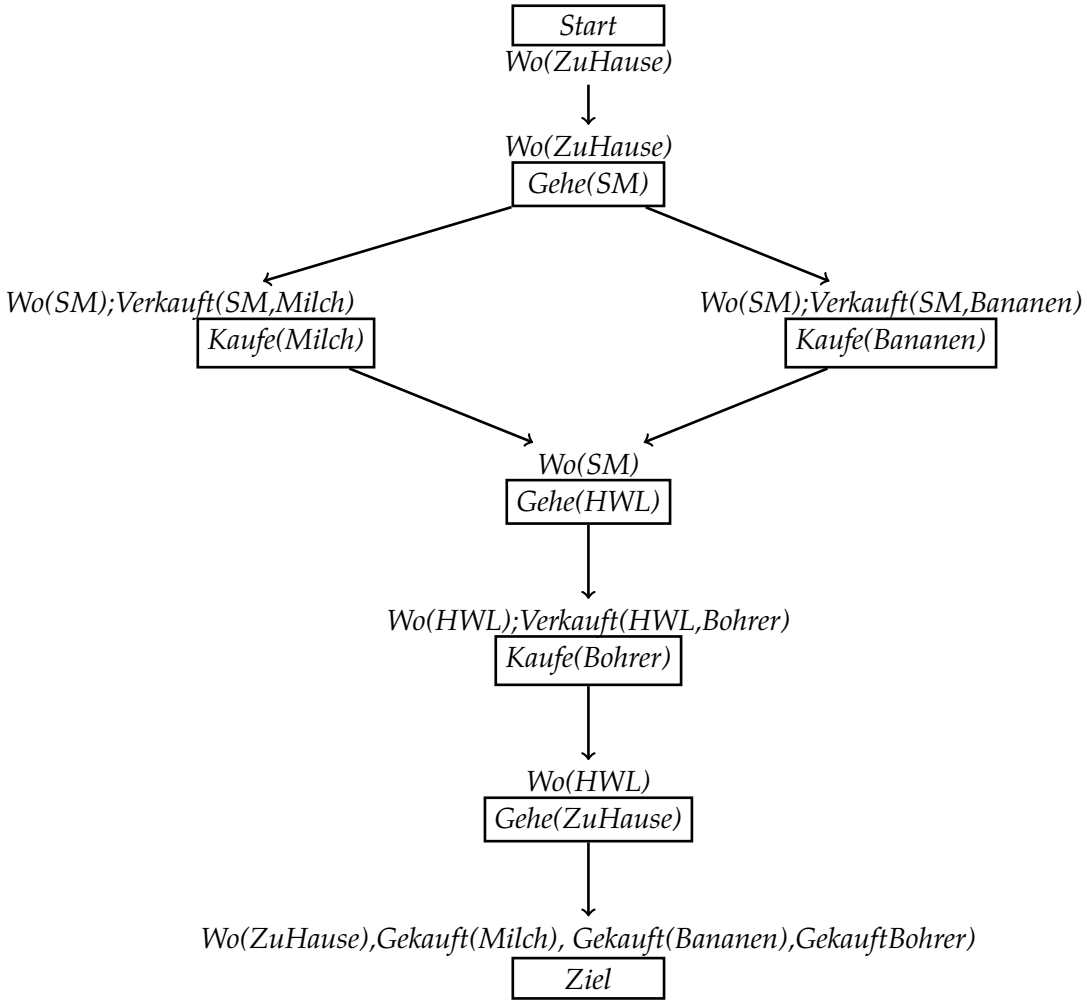


Man kann x, y nicht gleichzeitig mit $zuHause$ instanziieren, da sich dann die Gehe-Aktionen widersprechen. Der Planungsalgorithmus kann das evtl. nicht sofort erkennen und muss an dieser Stelle evtl. ausprobieren, d.h. Nach dem Ausprobieren backtracken.

Wählt man eine davon zuerst aus, dann kann man zB $x = zuHause$ wählen, und somit zuerst zum Supermarkt gehen. Danach muss die Vorbedingung $Gehe(HWL)$ erfüllt werden. Dies geht, indem man $Gehe(HWL)$ nach $Gehe(SM)$ fordert. Da aber dann die Vorbedingung des Kaufens im Supermarkt gestört werden könnten, kann man versuchen+ es später als $Kaufe(Milch)$ und als $Kaufe(Bananen)$ einzureihen. Eine naheliegende Instanziierung des y ist SW. Danach ist alles erfüllt, bis auf die letzte Bedingung, dass man auch wieder zu hause ankommen soll:

Es fehlt somit ein $Gehe(zuHause)$ vor dem Ziel, und nach allen anderen. Die Vorbedingung ist $Wo(HWL)$. Der Plan ist vollständig.

Man sieht, dass $Gehe()$ eine Sequentialisierung bewirkt. Der fertige Plan ist:



4.3.0.1 Kommentar zu Argumenten der Operatoren

Dieses Beispiel hat Operatoren, bei denen nicht alle in den Bedingungen und Effekten verwendeten Variablen auch Argumente der Operatoren sind. Z.B. $Gehe$ und $Kaufe$.
Bei $Kaufe$ fehlt das Argument, wo gekauft wurde. Bei $Gehe$ fehlt das "woherïn der Argumentliste in Russel und Norvig. Das hat den Nachteil, dass nicht alle Effekte aus den

Argumentlisten hervorgehen.

Eine bessere Vereinbarung ist: Alle Variablen, die in den Effekten vorkommen, sollten auch Argumente des Operators sein. Variablen, die nur in der Vorbedingung vorkommen, können weggelassen werden.

4.3.1 Erweiterungen

Hierarchisches Planen: Man hat verschiedene Abstraktionsebenen der Pläne. Die Aktionen und Zustände sind entsprechend zu erweitern.

Planung kann erfolgen durch Schrittweise Verfeinerungen.
z.B. Reiseroute von der Uni Frankfurt nach Chicago.
Zuerst macht man einen groben Plan, dann eine Feinplanung .

Zeit-Constraints

Hat man Bedingungen, wie lange eine Aktion dauert, bzw. Ressourcenbeschränkungen, dann verkompliziert sich das Planen. Hier gibt es Analogien zum Job Shop Scheduling Probleme, so dass man Methoden des Operations Research anwenden kann.

Literatur

Michalewicz, Z. (**1992**). *Genetic Algorithms + Data Structures = Evolution Programs*. Springer, 3 edition.

Russell, S. J. & Norvig, P. (**2010**). *Artificial Intelligence - A Modern Approach (3. internat. ed.)*. Pearson Education.

Smith, B. (**1982**). *Prologue to Reflection and Semantics in a Procedural Language*. Readings in Knowledge Representation. Morgan Kaufmann, California.

Wegener, I., editor (**1996**). *Highlights aus der Informatik*. Springer, Berlin.