Wissensbasierte Systeme

Master of Science

Prof. Dr. Rethmann

Fachbereich Elektrotechnik und Informatik
Hochschule Niederrhein

WiSe 2023/24

Übersicht

Inhalt

Künstliche Intelligenz

- Geschichte und Anwendungen
- Wissensrepräsentation
- Zustandsraumsuche
- Aussagen- und Prädikatenlogik
- Einführung in PROLOG

Literatur

- S. Russell, P. Norvig: *Artificial Intelligence – A Modern Approach.* Prentice Hall, 2002.
- George F. Luger: *Künstliche Intelligenz – Strategien zur Lösung komplexer Probleme.* Addison-Wesley, 2001.
- C. Beierle, G. Kern-Isberner: *Methoden wissensbasierter Systeme.* Vieweg Verlag, 2003.
- G. Görz, C.-R. Rollinger, J. Schneeberger (Hrsg.): *Handbuch der künstlichen Intelligenz.* 4. Auflage, Oldenbourg Verlag, 2003.
- U. Lämmel, J. Cleve: *Lehr- und Übungsbuch Künstliche Intelligenz.* Fachbuchverlag Leipzig, 2001.
- J. Heinsohn, R. Socher-Ambrosius: *Wissensverarbeitung – Eine Einführung.* Spektrum Akademischer Verlag, 1999.

Literatur

- Uwe Schöning: *Logik für Informatiker.* Spektrum Akademischer Verlag, 2000.
- Ivan Bratko: *PROLOG – Programming for Artificial Intelligence.* 3rd edition, Addison-Wesley, 2000.
- Richard S. Sutton and Andrew G. Barto. *Reinforcement learning - an introduction*. Adaptive computation and machine learning. MIT Press, 1998.
- Csaba Szepesvári. *Algorithms for Reinforcement Learning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2010.
- I. Gerdes, F. Klawonn, R. Kruse: *Evolutionäre Algorithmen.* Vieweg Verlag, 2004.
- Hans-Jürgen Zimmermann: *Operations Research.* Vieweg Verlag, 2005.
- Andreas Zell: *Simulation Neuronaler Netze.* Addison-Wesley, 1994.

Übersicht

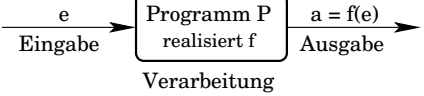
Übersicht

Ein Programm P realisiert eine Funktion f, so dass zu einer Eingabe e eine Ausgabe $a = f(e)$ berechnet wird. $\rightarrow$ EVA-Prinzip: Eingabe, Verarbeitung, Ausgabe.

Das Programm P und damit die Funktion f wird explizit in einer Programmiersprache wie C# oder Java implementiert.

Jedes Programm berechnet also eine Funktion $f : \mathbb{D} \rightarrow \mathbb{W}$, wobei $\mathbb{D}$ der Definitions- und $\mathbb{W}$ der Wertebereich ist.

Funktion f bildet Argument $z \in \mathbb{D}$ auf den Wert $f(z) \in \mathbb{W}$ ab. Notation: $z \mapsto f(z)$



Beispiel: Sortiere die gegebenen Zahlen $x_1, \ldots, x_n$ aufsteigend. Dann bildet f eine Eingabe $e = (x_1, \ldots, x_n)$ auf eine Ausgabe $a = f(e) = (x_{\pi(1)}, \ldots, x_{\pi(n)})$ ab, wobei π eine Permutation ist und $x_{\pi(i)} \leq x_{\pi(j)}$ für $1 \leq i < j \leq n$ gilt.

Algorithmen wie Quicksort, Mergesort oder Heapsort realisieren die Funktion f, indem sie beschreiben, WIE eine Zahlenfolge sortiert wird.

Was tun wir aber, wenn wir die zu realisierende Funktion f nicht kennen? Wie soll f dann als Programm realisiert werden?

> Wie sieht eine Funktion f aus, die bei Eingabe eines Bildes, bestehend aus 32×32 Pixeln mit bis zu 256 Grauwerten pro Pixel, die dort enthaltenen Ziffern erkennt? Es soll also eine Funktion $f : \{0, \ldots, 255\}^{32 \times 32} \to \{0, 1, \ldots, 9\}$ berechnet werden.

Antworten auf solche Fragen liefert das Machine Learning, ein Teilbereich der künstlichen Intelligenz.

> Mittels neuronaler Netze lässt sich das Problem der Ziffernerkennung heute sehr gut lösen. Im wesentlichen basiert die gute Erkennung darauf, dass sich die Netze erinnern, eine ähnliche Ziffer schon einmal während des Trainingsprozesses gesehen zu haben und daher der Eingabe eine Ziffer zuordnen können.

Wie könnte mathematisch ein Maß für die Ähnlichkeit zweier Ziffern definiert werden?

Was können Menschen besser als Maschinen?

Bilder erkennen, Texte übersetzen, Sprache verarbeiten, Spiele spielen, Planen, Beweisen, Schlussfolgern, Lernen, neue Situationen erfassen und beurteilen, usw.

Menschen lernen durch Beispiele bzw. Analogie und nutzen Ähnlichkeiten: Vorhandenes Wissen wird an neue Situationen angepasst. Ist das Intelligenz?

Voraussetzung ist, dass für ähnliche Situationen bereits Problemlösungen bekannt sind. Es wird also eine große Menge an Fällen bzw. Beispielen benötigt.

Wenn eine neue Situation N ähnlich zu einer anderen, bereits bekannten Situation A ist, und diese Situation A früher mit einem Verfahren V gelöst wurde, müssen zwei Fälle unterschieden werden:

- Falls V für A erfolgreich war, dann wende V auf N an,
- sonst wähle ein anderes Verfahren oder passe V an, sodass es auf N angewendet werden kann.

Was können Menschen besser als Maschinen?

Beim fallbasierten Schließen besteht die Wissensbasis nicht aus generischen Regeln, sondern aus einer Sammlung von Fällen, in denen spezifische frühere Erfahrungen gespeichert sind.

Neue Probleme, mit denen das System konfrontiert wird, werden dadurch gelöst, dass der relevanteste Fall aus der Falldatensammlung heraus gesucht wird und dessen Lösung in geeigneter Form auf das neue Problem übertragen wird.

Schließen kann als erinnerungsbasierter Prozess verstanden werden, wie wir bereits bei den neuronalen Netzen gesehen haben.

Dazu müssen Ähnlichkeiten bestimmt werden, wozu wir wieder Funktionen angeben müssen! Sind diese Funktionen nicht bekannt, setzen wir wieder Methoden des Machine Learnings ein.

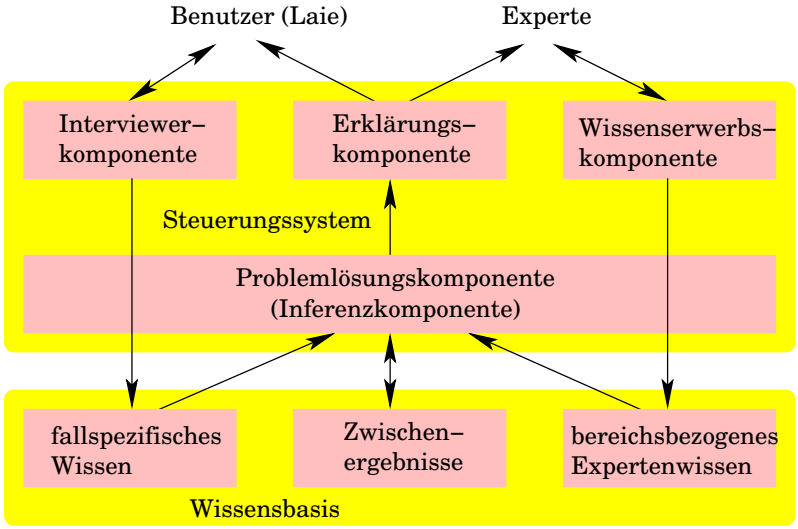
Was ist Wissen?

- Philosophie: Wissen ist wahrer, gerechtfertigter Glaube.
- Wissensrepräsentation: Wissen ist Information über die Welt, ohne Anspruch auf deren Wahrheitsgehalt.
- Beispiel: Im Mittelalter wusste man, dass die Erde eine Scheibe ist!

Im weiteren Sinn ist jede Programmierung Wissensverarbeitung: Jedes Programm kodiert und verarbeitet Wissen. Im engeren Sinn wird Wissensverarbeitung der KI zugeordnet.

- Artificial intelligence (AI) is the study of any device that perceives its environment and takes actions that maximize its chance of successfully achieving its goals. or: AI is whatever hasn't been done yet.
- Machine learning (ML) is the study of computer algorithms that improve automatically through experience.
- Expert system is a computer system that emulates the decision-making ability of a human expert. Expert systems are designed to solve complex problems by reasoning through bodies of knowledge. (wikipedia)

Trennung von Wissen und Verarbeitung:



Steuerungssystem:

- *Problemlösungskomponente (Inferenzkomponente):* Interpretiert das Expertenwissen, um eine Lösung des vom Benutzer gegebenen Problems zu finden.
- *Interviewerkomponente:* Führt den Benutzerdialog, liest die Messdaten ein und stellt fallspezifisches Wissen dar.
- *Erklärungskomponente:* Liefert dem Benutzer eine Begründung für die Problemlösung. Manche Menschen akzeptieren die von einer Maschine vorgeschlagene Lösung nur dann, wenn das Programm erklärt, wie es zu der Lösung gekommen ist.
- *Wissenserwerbskomponente:* Aufbau und Wartung des Fachwissens.

Wissensbasis:

- *Expertenwissen:* Fachwissen aus dem Anwendungsbereich, häufig in Form von Regeln.
- *fallspezifisches Wissen:* Daten, die der Anwender zu seinem konkreten Problem beisteuert, häufig in Form von Fakten.
- *Zwischenergebnisse / Problemlösungen:* Werden von der Inferenzkomponente erzeugt.

Vorteil dieser Architektur:

- Die Wissensbasis ist austauschbar!

Arten von Wissen:

- Relationales Wissen spielt im Alltag eine große Rolle:
  - Person X ist mit Y verheiratet.
  - Der Motor ist ein Teil vom Auto.
- Vererbung von Eigenschaften:
  - Der Golf ist ein Auto. Ein Auto hat einen Motor.

  $\Rightarrow$ Der Golf hat einen Motor.
- Prozedurales Wissen schreibt eine Folge von Aktionen vor:
  - Eröffnungen beim Schach
  - das Betanken eines Autos
  - das Betreten eines Restaurants
- Logisches Wissen:
  - Jeder Mensch ist sterblich. Sokrates ist ein Mensch.

  $\Rightarrow$ Sokrates ist sterblich.

- Zeitabhängiges Wissen
  - „Es ist kalt“ ist nicht immer richtig.
  - ... täglich außer Montags ...
  - Devisenkurse, Füllstände usw. sind zeitabhängig.
  - Beim Planen sind zeitliche Relationen wichtig: *vor*, *nach*, ...

  → Temporallogik, Situationskalkül

- Unvollständiges Wissen
  - Eine Kamera erfasst nicht die gesamte Umgebung. Was liegt hinter dem nächsten Hügel?
  - Nicht alle Zusammenhänge in der Medizin sind bekannt.

  → Fehlende Informationen werden durch Annahmen ersetzt.

- Vages Wissen
  - unscharf: Der Mann ist groß. Das Wetter ist gut.
  - unsicher: Temperatursensor hat Toleranz von $\pm 1^{\circ}$C.

  → Fuzzy-Logik, Sicherheitsfaktoren

Wissen

Darstellung von Wissen:

- Neuronale Netze
- Regeln
- Semantische Netze und Frames
- Aussagen- und Prädikatenlogik
- Skripte
- Bayessche Netze

Anforderungen an die Wissensrepräsentation:

- Wissen ist leicht überprüfbar und aktualisierbar.
- Ableiten neuen Wissens / Schlussfolgern ist möglich.
- Hypothesen können bestätigt oder widerlegt werden.

Warum ist die Darstellung von Wissen wichtig?

- Wissen muss effizient verarbeitet werden.
- Qualität der Antwort muss den Erfordernissen entsprechen.
- Beispiel: Rechnen mit römischen Zahlen vs. indischen Zahlen.

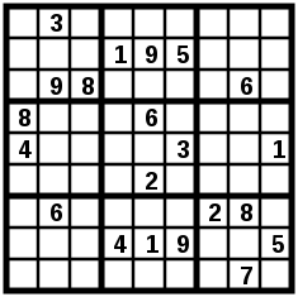
Closed-World-Assumption:

- Nur positive Fakten werden explizit gespeichert. Fehlende Information wird als Falschheit der Information gedeutet.
- Oft richtig und vernünftig, um Wissensbasis klein zu halten:
  - Wenn kein Direktflug von Bonn nach London in der Datenbank gespeichert ist, dann gibt es auch keinen.
  - Die Kannibalen[(1)] dürfen nicht unter Drogen gesetzt werden und es gibt auch keine Brücke über den Fluss.

[(1)] Drei Missionare und 3 Kannibalen wollen einen Fluss überqueren. Das Ruderboot für die Überfahrt kann nur 2 Personen aufnehmen. Es dürfen nie mehr Kannibalen als Missionare an einem Ort sein, sonst wird es unschön.

Anwendung bei Problemen, die keine algorithmische Lösung haben, z.B. einige Suchprobleme mit exponentiell großem Lösungsraum.

- Logik-Rätsel Sudoku: Fülle das Gitter mit den Ziffern 1 bis 9 so, dass jede Ziffer in jeder Spalte, in jeder Zeile und in jedem Block genau einmal vorkommt.
- Zahlenrätsel: `SEND + MORE = MONEY`
- Spiele: Tic-Tac-Toe, Mühle, Schach, Go



Übung 1. Schreiben Sie ein Programm, das ein gegebenes $n \times n$-Sudoku für $n = 9, 16, 25, 36, 49$ mittels Backtracking löst. Welche Laufzeiten ergeben sich für große Werte von n?

Methode der Wissensbasierten Systeme: Einsatz von Wissen!

- Erfolg ist nicht garantiert, aber oft besser als ohne Wissen.
- Wissen ist anwendungsspezifisch, oft als Heuristik realisiert.

Viele bekannte Spiele sind schwer:

- Sudoku ist NP-vollständig: Yato und Seta (2003)
- Mastermind ist NP-vollständig: Stuckman und Zhang (2006)
- Minesweeper ist NP-vollständig: Kaye (2000)
- Brett-Solitär ist NP-vollständig: Uehara und Iwata (1990)
- Sokoban ist PSPACE-vollständig: Dor und Zwick (1999)
- Rush Hour ist PSPACE-vollständig: Flake und Baum (2002)

Eine schöne Übersicht über die Komplexität von Spielen gibt das Papier[(2)] von G. Kendall, A.J. Parkes, K. Spoerer: A Survey of NP-Complete puzzles. ICGA Journal. 2008 (31). 13-34.

[(2)] `https://www.researchgate.net/publication/220174445_A_Survey_of_NP-Complete_puzzles`

Reduktion des Suchraums durch Wissen:

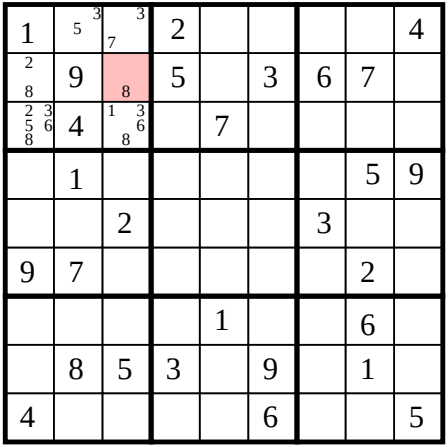
Streiche in jedem Feld aus der Liste der prinzipiell möglichen Zahlen alle die Zahlen heraus, die bereits in der jeweiligen Zeile, der Spalte oder in dem Unterblock vorkommen.

4			2	7		6		
7	9	8	1	5	6	2	3	4
	2		8	4				7
2	3	7	4	6	8	9	5	1
8	4	9	5	3	1	7	2	6
5	6	1	7	9	2	8	4	3
	8	2		1	5	4	7	9
	7			2	4	3		
		4		8	7			2

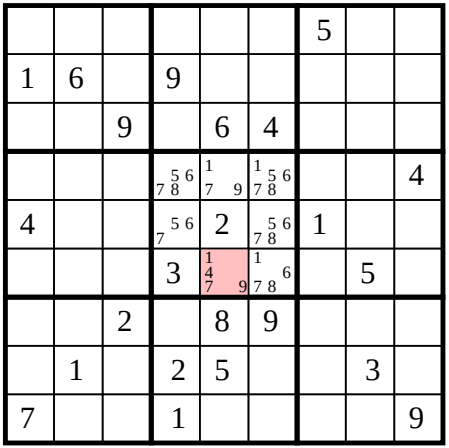
$\Longrightarrow$

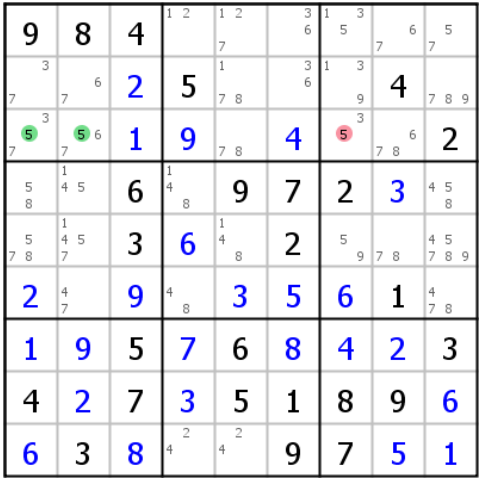
4	1 5	3 5	2	7	3 9	6	1 8 9	5 8
7	9	8	1	5	6	2	3	4
1 6	2	3 5 6	8	4	3 9	1 5	1 9	7
2	3	7	4	6	8	9	5	1
8	4	9	5	3	1	7	2	6
5	6	1	7	9	2	8	4	3
3 6	8	2	3 6	1	5	4	7	9
1 6 9	7	5 6	6 9	2	4	3	1 8 6	5 8
1 3 6 9	1 5	4	3 6 9	8	7	1 5	1 6	2

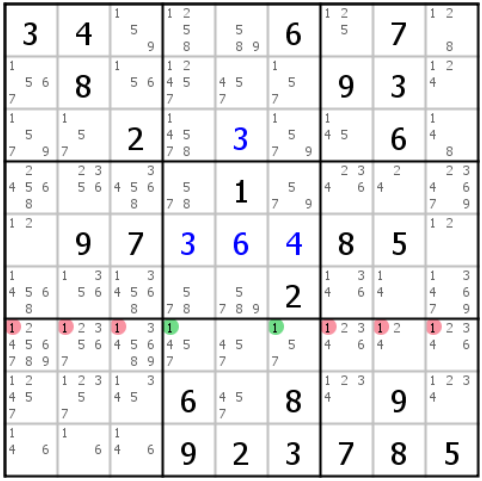
Naked Single - einzig möglicher Wert: Das Kästchen muss diesen Wert annehmen, da alle anderen Werte bereits vergeben sind.



Hidden Single - einzig möglicher Ort: Das Kästchen muss diesen Wert annehmen, da alle anderen Kästchen diesen Wert nicht annehmen können.

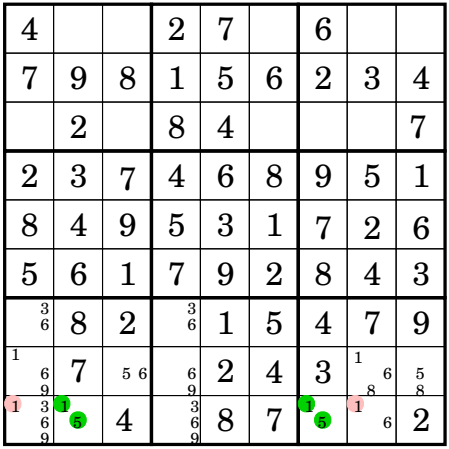




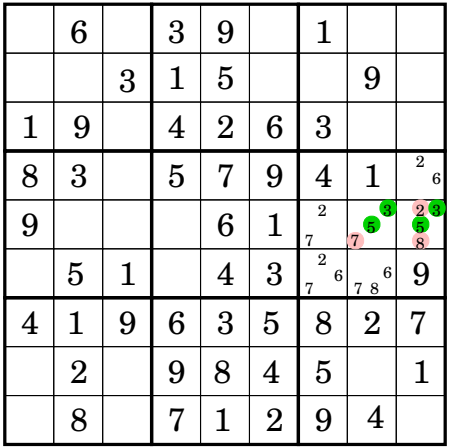


Locked Candidates: Wenn in einem Block alle Kandidaten einer bestimmten Ziffer auf eine Zeile oder Spalte beschränkt sind, kann diese Ziffer nicht außerhalb dieses Blocks in dieser Zeile oder Spalte erscheinen. Quelle: `http://hodoku.sourceforge.net/`

Naked Pair: Kommen in zwei Zellen eines Bereichs ausschließlich die zwei gleichen Werte vor, dann müssen sich diese zwei Zellen diese zwei Werte teilen.



Hidden Pair: Kommt ein Wertepaar nur in zwei Zellen vor, dann müssen sich diese beiden Zellen diese Werte teilen, da keine andere Zelle sie übernehmen kann.



Weitere Methoden zur Einschränkung des Suchraums bei Sudoku finden Sie bspw. unter: `www.sudokuoftheday.com/pages/techniques-overview.php`

Übung 2. Bauen Sie obige Techniken in Ihren Sudoku-Solver ein und vergleichen Sie die Laufzeiten der beiden Versionen.

Zahlenrätsel: Wissen über Addition beschränkt den Suchraum.

- Bei Addition zweier Ziffern ist der Übertrag 1.

$$\begin{array}{cccccc} & S & E & N & D \\ + & M & O & R & E \\ \hline M & O & N & E & Y \end{array} \quad \rightarrow \quad \begin{array}{cccccc} & S & E & N & D \\ + & 1 & O & R & E \\ \hline 1 & O & N & E & Y \end{array}$$

Zahlenrätsel: Wissen über Addition beschränkt den Suchraum.

- Bei Addition zweier Ziffern ist der Übertrag 1.

	S	E	N	D	$\rightarrow$		S	E	N	D
+	M	O	R	E		+	1	O	R	E
M	O	N	E	Y		1	O	N	E	Y

- S muss gleich 8 oder 9 sein, da sonst kein Übertrag erzielt würde. Wir setzen zunächst S auf 9.

	S	E	N	D	$\rightarrow$		9	E	N	D
+	1	O	R	E		+	1	O	R	E
1	O	N	E	Y		1	O	N	E	Y

Zahlenrätsel: Wissen über Addition beschränkt den Suchraum.

- Bei Addition zweier Ziffern ist der Übertrag 1.

	S	E	N	D	→		S	E	N	D
+	M	O	R	E		+	1	O	R	E
M	O	N	E	Y		1	O	N	E	Y

- S muss gleich 8 oder 9 sein, da sonst kein Übertrag erzielt würde. Wir setzen zunächst S auf 9.

	S	E	N	D	→		9	E	N	D
+	1	O	R	E		+	1	O	R	E
1	O	N	E	Y		1	O	N	E	Y

- Dann muss O gleich 0 sein. (1 ist bereits durch M belegt!)

	9	E	N	D	→		9	E	N	D
+	1	O	R	E		+	1	0	R	E
1	O	N	E	Y		1	0	N	E	Y

Einsatz von Wissen

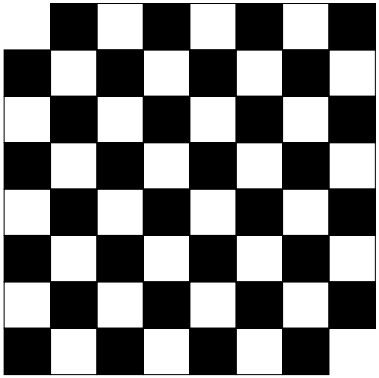
Zahlenrätsel: Was bringt der Einsatz von Wissen?

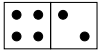
- Wir haben insgesamt 8 verschiedene Buchstaben im obigen Rätsel.
- Jeder Buchstabe stellt eine Ziffer dar, wobei verschiedene Buchstaben auch verschiedene Ziffern darstellen.
- Wir haben also im ursprünglichen Problem $10 \cdot 9 \cdot 8 \cdot 7 \cdot \ldots \cdot 3 = 1.814.400$ Kombinationen zu testen.
- Durch den Einsatz des Wissens legen wir die Werte von M und O eindeutig fest, für S verbleiben nur 2 Möglichkeiten.

$\rightarrow$ Damit reduzieren wir den Suchraum durch den Einsatz von Wissen auf $2 \cdot 7 \cdot 6 \cdot 5 \cdot \ldots \cdot 3 = 5.040$ Kombinationen.

Explizites Wissen über einen Aufgabenbereich verkürzt die Suche!

Plazierung von Dominosteinen auf einem Schachbrett, bei dem zwei diagonal gegenüberliegende Eckfelder fehlen.





Frage: Kann man das Schachbrett so mit Dominosteinen bedecken, dass alle Felder belegt sind, keine Dominosteine über den Rand herausragen und sich keine Dominosteine überlappen?

Einsatz von Wissen

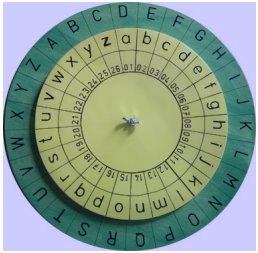
Lösungsmöglichkeiten:

- Vollständige Suche:
  Teste alle Möglichkeiten, Dominosteine auf das Brett zu legen. Stop, sobald alle Felder belegt sind oder falls alle Möglichkeiten fehlgeschlagen sind.
- Reduktion des Suchraums durch Symmetriebetrachtungen oder Heuristiken.
- Einsatz von Wissen:
  Das Brett hat 30 weiße und 32 schwarze Felder. Jeder Stein bedeckt genau ein schwarzes und ein weißes Feld. $\Rightarrow$ Es gibt keine Lösung für das Problem!

Der Einsatz von Wissen kann eine Suche enorm verkürzen!

Cäsar-Code: Ersetzte jeden Buchstaben des Textes durch den Buchstaben, der im Alphabet n Stellen danach kommt.

Beispiel: Für $n = 2$ wird aus dem Klartext `ELEFANTEN` der verschlüsslete Text `GNGHCPVGP`.



Codebrechen mittels Häufigkeitstabelle:

Buchstabe	Häufigkeit
E	17,40%
N	9,78%
I	7,55%
S	7,27%
R	7,00%

Paare	Häufigkeit
ER	4,09%
EN	4,00%
CH	2,42%
DE	1,93%
EI	1,87%

Codierter Text: FKGUKUVGKPGIGJGKOGPCEJTKEJV
Klartext: ??

Buchstabe	Anzahl	relative Häufigkeit
G	6	23,1%
K	5	19,2%
J	3	11,5%
C	2	7,7%
U	2	7,7%

Vermutlich G $\rightarrow$ E: DIESISTEINEGEHEIMENACHRICHT

Was ist künstliche Intelligenz?

- 1955 John McCarthy:
  Ziel der KI ist es, Maschinen zu entwickeln, die sich verhalten, als verfügten sie über Intelligenz.
- 1991 Encyclopedia Britannica:
  KI ist die Fähigkeit digitaler Computer oder computergesteuerter Roboter, Aufgaben zu lösen, die normalerweise mit den höheren intellektuellen Verarbeitungsfähigkeiten von Menschen in Verbindung gebracht werden.
- 1983 Elaine Rich:
  Forschung darüber, wie Computer Dinge machen, die Menschen derzeit noch besser beherrschen.

Heute:

- KI ist die Untersuchung von Berechnungsverfahren, die es ermöglichen, wahrzunehmen, zu schlußfolgern und zu handeln.
- Ermitteln, welche Annahmen über die Repräsentation und Verarbeitung von Wissen und den Aufbau von Systemen die verschiedenen Aspekte der Intelligenz erklären können.

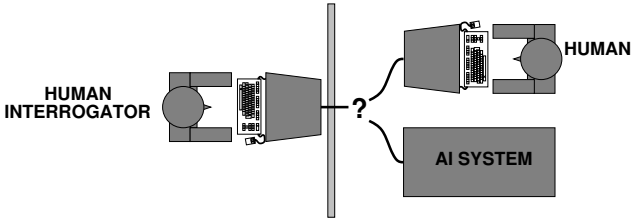
$\Rightarrow$ Synthese vor Analyse: Lerne menschliche Intelligenz zu verstehen durch die Konstruktion intelligenter Systeme.

Vorherrschender Gedanke:

- Intelligenz emergiert aus der Interaktion vieler einfacher Prozesse im Zusammenspiel und
- Prozessmodelle intelligenten Verhaltens können mit Hilfe des Computers im Detail untersucht werden.

Übersicht

Der Britische Mathematiker Alan Turing schlug 1950 in seinem Artikel „Computing Machinery and Intelligence“ einen Test zur Überprüfung intelligenten Verhaltens einer Maschine durch Vergleich mit einem Menschen vor:



- Eine Person befragt einen Menschen und ein KI-System, wobei der Tester jede beliebige Frage stellen kann.
- Der Tester kommuniziert nur indirekt über ein Textmedium. Dadurch soll vermieden werden, dass der Tester durch das äußere Erscheinen oder durch mechanische Eigenschaften wie eine synthetische Stimme beeinflusst wird.
- Der Tester soll allein aus den Antworten auf seine Fragen entscheiden, wer Mensch und wer Computer ist. Wenn er das nicht kann, dann ist die Maschine intelligent.

Anforderungen an das Computerprogramm:

- Sprachverstehen und Sprachgenerierung
- Repräsentation von Wissen
- Schlussfolgern
- Lernen

Der Artikel von Alan Turing enthält außerdem erste Gedanken zum Thema Lernen.

Übung 3. Implementieren Sie das Spiel Mastermind.

- schwarz: Ein Spielstein mit der richtigen Farbe wurde an der richtigen Stelle plaziert.
- weiß: Ein Spielstein hat zwar die richtige Farbe, steht aber an der falschen Position.

`https://de.wikipedia.org/wiki/Mastermind_(Spiel)`

Übung

Anstelle verschieden farbiger Stecker verwenden wir Ziffern von 0 bis 5.

- Der Mensch denkt sich eine Kombination von vier Ziffern aus.
- Das Programm soll mit möglichst wenigen Versuchen diese Kombination ermitteln. In jedem Schritt gibt das Programm eine Kombination aus und wartet auf die Bewertung seines Versuchs durch den menschlichen Gegenspieler.
- Durch Analysieren der Bewertungen können nun Informationen gewonnen werden, die helfen, den nächsten Zug zu verbessern.

Beispiel: Der Mensch wählt die Kombination `0,1,2,3`. Im folgenden Dialog macht der Computer jeweils einen Vorschlag, worauf der Mensch seine Bewertung abgibt.

```
1. Versuch: 3,3,3,1 --> Bewertung (s,w) ? 0,2
2. Versuch: 5,1,1,3 --> Bewertung (s,w) ? 2,0
3. Versuch: 2,1,4,3 --> Bewertung (s,w) ? 2,1
4. Versuch: 4,1,0,3 --> Bewertung (s,w) ? 2,1
Lösung: 0,1,2,3
```

Würden Sie ein solches Programm als intelligent bezeichnen?

Eliza

Joseph Weizenbaum (1966) wird zum Gesellschaftskritiker wegen des Erfolgs seines Programms Eliza:

- Oberflächliche Simulation eines Psychotherapeuten.
- Das Kommunikationsverhalten von Versuchspersonen gegenüber dem Programm entsprach demjenigen gegenüber einem menschlichen Gesprächspartner.
- Die Versuchspersonen waren zum Teil überzeugt, dass der Gesprächspartner ein tatsächliches Verständnis für ihre Probleme aufbrachte.
- Praktizierende Psychiater glaubten ernsthaft daran, mit dem Programm zu einer automatisierten Form der Psychotherapie gelangen zu können.

Eliza

- Ein Aussagesatz des Benutzers kann in einen Fragesatz umgewandelt werden, indem ein „Warum“ davor gestellt wird.
  - aus „Ich fühle mich unwohl in Gegenwart meiner Mutter.“ wird
  - „Warum fühlen Sie sich unwohl in Gegenwart Ihrer Mutter?“
- Eltern wissen, dass man durch Warum-Fragen schnell an die Wand gespielt wird. Daher: Reagiere auf Schlüsselwörter.
  - auf das Stichwort „Arbeit“ kann das Programm reagieren mit
  - „Haben Sie das Gefühl, Sie arbeiten zuviel?“ oder
  - „Kommt Ihre Familie zu kurz, wenn Sie viel arbeiten?“ oder
  - „Definieren Sie sich über Arbeit? Haben Sie keine Hobbies?“
- Zum Auflockern des Dialogs werden Phrasen eingestreut.
  - „Können Sie darauf bitte genauer eingehen?“
  - „Nehmen Sie kein Blatt vor den Mund.“
  - „Wie fühlen Sie sich jetzt?“
  - „Lassen Sie uns noch etwas bei diesem Thema bleiben.“
  - „Was denken Sie, sind die Gründe dafür?“
  - „Was könnte denn die Ursache von all dem sein?“

Anmerkungen:

- Das Erkennen von Schlüsselwörtern muss auch bei Rechtschreibfehlern korrekt funktionieren. Wir brauchen Fehlertoleranz im Dialog!
- Das Umstellen von Satzteilen ist je nach verwendeter Sprache unterschiedlich kompliziert und erfordert ein Erkennen der Satzstruktur, also der Grammatik.
- Deklination und Konjugation müssen berücksichtigt werden, um Schlüsselwörter zu erkennen: er wird sterben, sie ist gestorben, er starb, du stirbst usw.
- Wir müssen Synonyme erkennen und in die Antworten einbauen: umkommen, versterben, entschlafen, den Tod finden, von uns gehen, ums Leben kommen, dem Schöpfer gegenüber treten usw.

Anmerkungen:

- Um authentisch zu wirken, muss dem Programm die Umgangssprache bekannt sein: abkratzen, draufgehen, verrecken, den Löffel abgeben, über den Jordan gehen usw.
- Die Reaktion auf Schlüsselwörter kann von der Bedeutung eines Satzes abhängen:
  - „Mein *Vater* ist im *Krieg* gefallen“ ist sicherlich anders zu bewerten als „der *Krieg* ist der *Vater* aller Dinge“.
  - „Ich habe das Auto meines Vaters *gestohlen*“ unterscheidet sich deutlich von „ich *stehle* meiner Schwester die Show“.

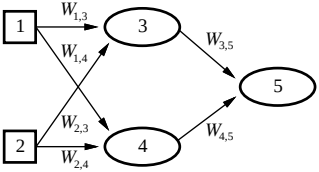
→ Die Semantik ist wichtig und muss erkannt werden!

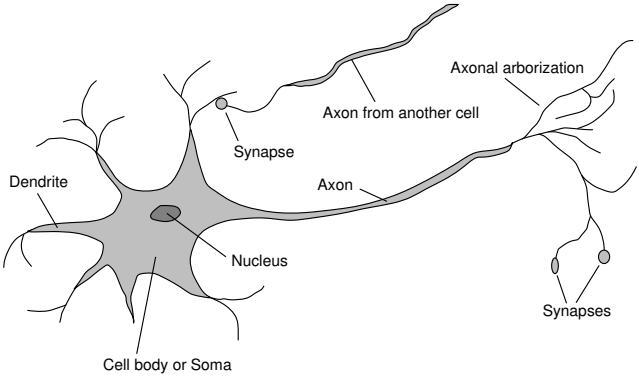
Um den Turing-Test wirklich zu bestehen, muss schon einiger Aufwand betrieben werden. Wird es jemals gelingen?

Übersicht

Cullach/Pitt 1943: Integration dreier Quellen zum Vorschlag eines Netzes künstlicher Neuronen: Physiologie der Neuronen, Aussagenlogik und Theorie der Berechenbarkeit (von Alan Turing)

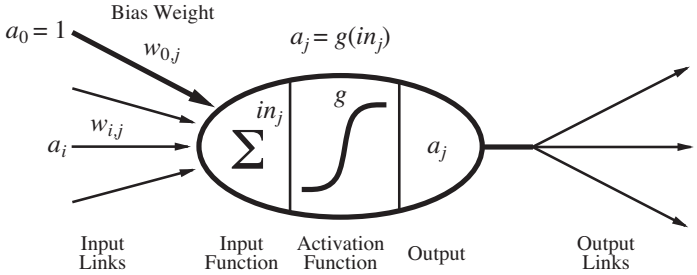
Minsky/Edmonds 1951: Build the first artificial neural network that simulated a rat finding its way through a maze.





Die von Minsky und Papert (1969) gezeigten theoretischen Grenzen von Perceptrons führen zu einem weitgehenden Ende der Forschungsförderung im Bereich neuronaler Netze.

Künstliches Neuron:

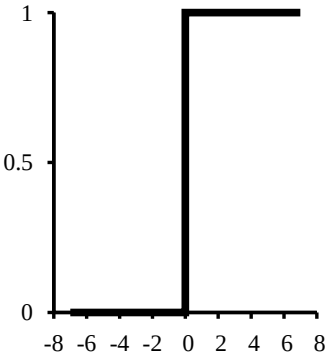


Die Input-Funktion berechnet die Summe der gewichteten Eingangssignale:

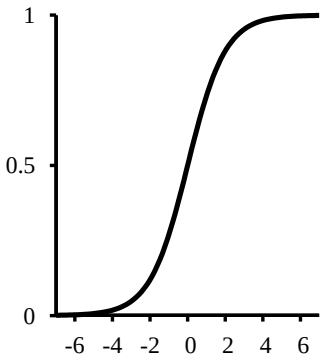
$$in_j = \sum_i a_i \cdot w_{ij}$$

Aktivierungsfunktionen bei neuronalen Netzen

Schwellwertfunktion:

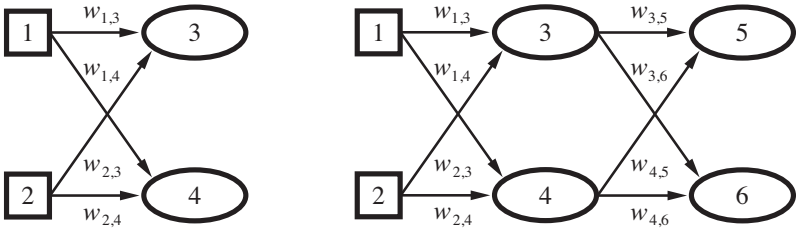


Logistische Funktion:



Häufig wird die logistische Funktion $\frac{1}{1+e^{-x}}$ oder der Tangens Hyperbolicus $\tanh(x)$ als Aktivierungsfunktion genutzt.

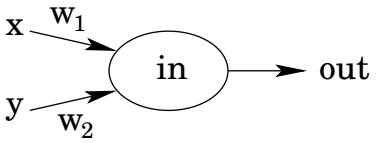
Vorwärts gerichtete neuronale Netze



- *überwachtes Lernen:* Vergleiche Ist- mit Sollausgabe zu einer Eingabe. Schließe aus der Differenz auf die vorzunehmenden Änderungen der Gewichte.
  → Delta-Regel, Backpropagation
- *unüberwachtes Lernen:* Erfolgt ausschließlich durch Eingabe der zu lernenden Muster. Das Netz verändert sich entsprechend den Eingabemustern von selbst.
  → Hebb: Oft genutzte Pfade werden verstärkt.

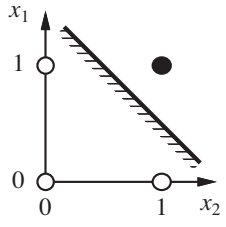
Perceptrons: single layer of artificial neurons connected by weights to a set of inputs

Linear separierbare Funktionen:

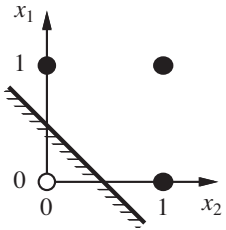


- Ein einzelnes Neuron mit zwei Eingängen x und y liefert $in = x \cdot w_1 + y \cdot w_2$.
- Die Gleichung ist linear in x und y, daher liegen alle Werte, die die Gleichung erfüllen, auf einer Geraden.
- Alle Werte auf einer der Seiten der Geraden produzieren Werte größer als der Schwellwert und aktivieren das Neuron.

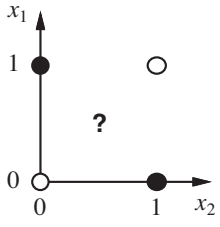
Minsky and Papert: a single layer perceptron cannot simulate a simple xor function.



(a) x_1 **and** x_2



(b) x_1 **or** x_2



(c) x_1 **xor** x_2

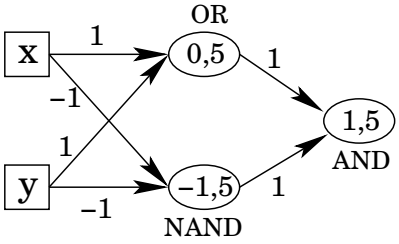
Exklusiv-Oder-Problem:

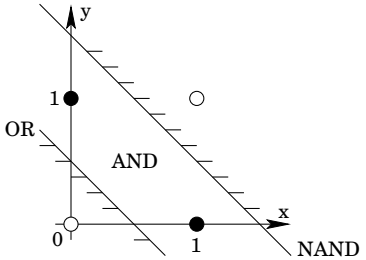
	w_1	w_2	Schwellwert
AND	1	1	1,5
OR	1	1	0,5
NAND	-1	-1	-1,5
XOR	?	?	?

- w_1 und w_2 beeinflussen die Steigung der Geraden,
- der Schwellwert beeinflusst die Position.

Fehler(?) von Minsky und Papert:

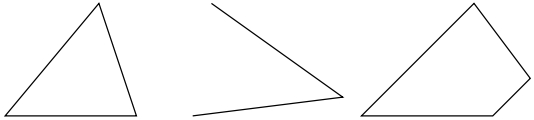
- Sie haben von der Leistung eines einstufigen Perceptrons mit einer trainierbaren Schicht unzulässiger Weise auf die Leistung eines mehrstufigen Perceptrons mit mehreren trainierbaren Schichten geschlossen.[3]
- Mehrstufige Perceptrons sind prinzipiell in der Lage, jede berechenbare Funktion zu realisieren.



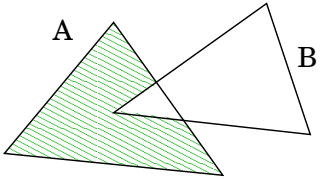


[3] Dies ist laut wikipedia so nicht richtig: What the book does prove is that in three-layered feed-forward perceptrons, it is not possible to compute some predicates unless at least one of the neurons in the first layer of neurons is connected with a non-null weight to each and every input.

Einfache, zweischichtige Netze führen zu konvexen Regionen, die offen oder geschlossen sein können.



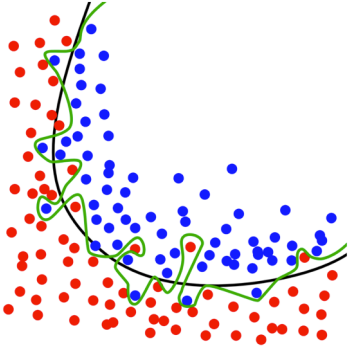
Mittels dreischichtigen Netzen können auch nicht-konvexe Regionen dargestellt werden.



Typische Anwendungen: Klassifizierungen von Daten, Prognose für Börsenwerte oder Energieverbrauch, Steuerungen in der Robotik oder von Prothesen, Zeichenerkennung.

Problem:

- Wie viele Neuronen benötigen wir in den einzelnen Schichten?
- Wann ist das Netz ausreichend trainiert?
- Wie verhindert man eine zu starke Anpassung an die Trainingsdaten?
- Bild rechts: `https://en.wikipedia.org/wiki/Overfitting`



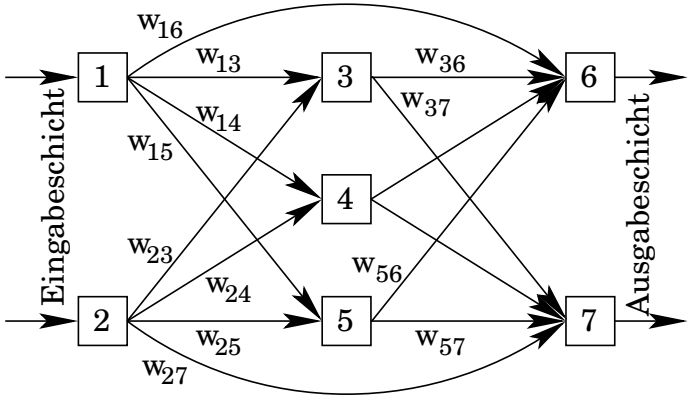
Das Buch von Rumelhart und McClelland *Parallel Distributed Processing*, 1986 verbreitet neue Erfolge mit dem wiederentdeckten Backpropagation-Algorithmus.

Der Backpropagation-Algorithmus wurde bereits 1969 von Bryson und Ho publiziert. Also im gleichen Jahr, in dem das kritische Buch von Minsky und Papert erschienen ist.

Backpropagation: Gradientenabstiegsverfahren zur Minimierung des Fehlers, also der Abweichung zwischen Ausgabe (Ist-Wert) und erwarteter Ausgabe (Soll-Wert).

- vorwärts gerichtete Netzwerke
- Matrix w speichert Gewichte der Kanten: w_{ij} ist das Gewicht der Kante von Neuron i zu Neuron j
- Aktivierungsfunktion: $\frac{1}{1+e^{-x}}$ ist überall differenzierbar

- out_i: Ausgabe des Neurons i.
- $in_j = \sum_i out_i \cdot w_{ij}$
- die Aktivierungsfunktion liefert: $out_j = \frac{1}{1+e^{-in_j}}$



Oft definiert man den Fehler des neuronalen Netzes

$$E = \frac{1}{2}\sum_j (t_j - out_j)^2$$

als Quadrat der Abweichung zwischen erwarteter Ausgabe und realer Ausgabe, damit sich negative und positive Abweichungen nicht aufheben.

Der Fehler wird minimiert, indem jedes Gewicht um einen durch die Lernrate η festgelegten Bruchteil des negativen Gradienten der Fehlerfunktion geändert wird:

$$\Delta w_{ij} = -\eta \cdot \frac{\partial E}{\partial w_{ij}}$$

Die Lernrate η steuert den Grad der Gewichtsänderung und das negative Vorzeichen bewirkt eine Veränderung entgegen dem Kurvenanstieg in Richtung eines Tals der Fehlerkurve.

Die Kettenregel liefert:

$$\Delta w_{ij} = -\eta \cdot \frac{\partial E}{\partial w_{ij}} = -\eta \cdot \frac{\partial E}{\partial in_j} \cdot \frac{\partial in_j}{\partial w_{ij}}$$

- $\frac{\partial in_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \sum_k out_k \cdot w_{kj} = out_i$
- Wir definieren als Fehlersignal: $\delta_j = -\frac{\partial E}{\partial in_j}$

Damit erhalten wir:

$$\Delta w_{ij} = \eta \cdot \delta_j \cdot out_i$$

Gewichtsänderung:

$$w_{ij}^{neu} := w_{ij}^{alt} + \Delta w_{ij}$$

Wir müssen jetzt nur noch das Fehlersignal $\delta_j = -\frac{\partial E}{\partial in_j}$ bestimmen.

Auch hier können wir wieder die Kettenregel anwenden:

$$\delta_j = -\frac{\partial E}{\partial out_j}\frac{\partial out_j}{\partial in_j}$$

Der zweite Faktor entspricht der ersten Ableitung der Aktivierungsfunktion:

$$\begin{aligned}
\frac{\partial out_j}{\partial in_j} = \frac{\partial}{\partial in_j}\frac{1}{1+e^{-in_j}} &= -\frac{-e^{-in_j}}{(1+e^{-in_j})^2} = \frac{e^{-in_j}}{(1+e^{-in_j})^2} \\
&= \frac{1}{1+e^{-in_j}} \cdot \left(\frac{e^{-in_j}}{1+e^{-in_j}}\right) \\
&= out_j \cdot \left(\frac{1+e^{-in_j}-1}{1+e^{-in_j}}\right) \\
&= out_j \cdot (1-out_j)
\end{aligned}$$

Für die Berechnung des ersten Faktors $-\frac{\partial E}{\partial out_j}$ benötigen wir eine Fallunterscheidung:

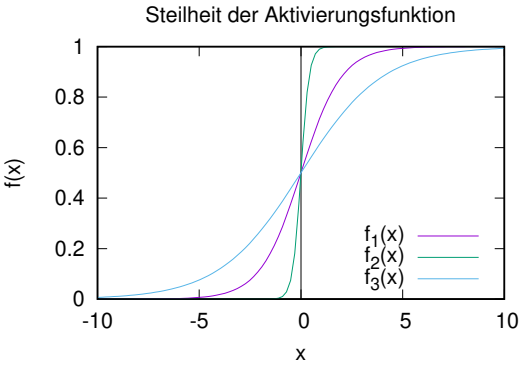
- Für ein Neuron j der Ausgabeschicht, dessen Fehler sich nicht auf andere Knoten auswirkt, gilt:

$$-\frac{\partial E}{\partial out_j} = -\frac{\partial}{\partial out_j}\left(\frac{1}{2}\sum_k (t_k - out_k)^2\right) = (t_j - out_j)$$

- Für ein Neuron j aus einer Zwischenschicht ist t_j nicht bekannt. Aber die verallgemeinerte Kettenregel liefert:

$$\begin{aligned} -\frac{\partial E}{\partial out_j} &= -\sum_k \frac{\partial E}{\partial in_k}\frac{\partial in_k}{\partial out_j} \\ &= \sum_k \left(\delta_k \frac{\partial}{\partial out_j}\sum_l out_l \cdot w_{lk}\right) = \sum_k \delta_k w_{jk} \end{aligned}$$

Die Steilheit der Kurve beim Übergang von 0 auf 1 kann durch einen zusätzlichen Faktor im Exponenten gesteuert werden:



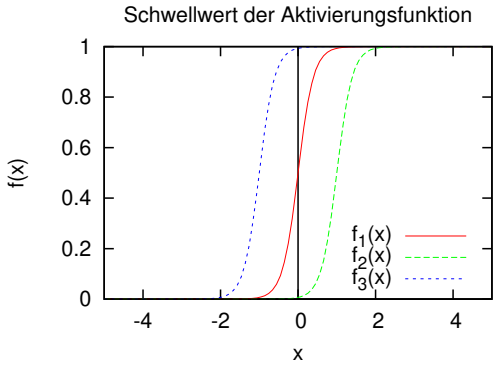
$$f_1(x) = \frac{1}{1 + e^{-x}}$$

$$f_2(x) = \frac{1}{1 + e^{-5x}}$$

$$f_3(x) = \frac{1}{1 + e^{-0.5x}}$$

Durch einen großen Faktor kann also eine Schwellwertfunktion nachgebildet werden.

Soll der Schwellenwert von 0 auf einen anderen Wert verschoben werden, wird noch ein additiver Term im Exponenten eingefügt:



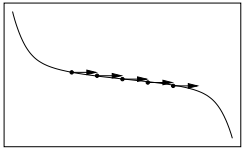
$$f_1(x) = \frac{1}{1 + e^{-5x}}$$

$$f_2(x) = \frac{1}{1 + e^{-5(x-1)}}$$

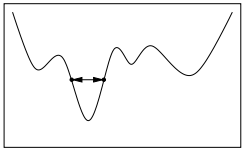
$$f_3(x) = \frac{1}{1 + e^{-5(x+1)}}$$

Festes Eingabesignal 1 über eine Kante mit Gewicht θ mit einem Neuron verbinden. So wird der Schwellenwert zu einem weiteren Gewicht, und wir müssen beim Lernen nur Gewichte optimieren.

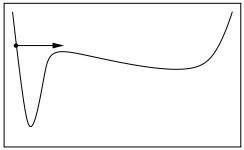
Eventuelle Schwierigkeiten beim Lernen:



Ist die Kurve sehr flach, dann ist der Anstieg betragsmäßig sehr klein, woraus eine geringe Gewichtsänderung resultiert. Der Trainingsprozess nimmt viel Zeit in Anspruch.



Oszillation in sehr steilen Schluchten: Durch den großen Anstieg erfolgt eine große Gewichtsänderung und es wird von einer Wand zur anderen gesprungen.



Durch eine steile Wand führt eine große Gewichtsänderung dazu, dass das tiefe Tal übersprungen wird und nur noch ein lokales Optimum gefunden wird.

Lernrate:

- klein → langsame Änderung der Gewichte, viele Iterationsschritte notwendig
- groß → schnelle Änderung der Gewichte, evtl. Oszillation um idealen Gewichtsvektor, evtl. wird Minimum nicht gefunden

→ monoton fallende Folge über alle Lernschritte (Problem: Bestimmung der Parameter für diese Folge?)

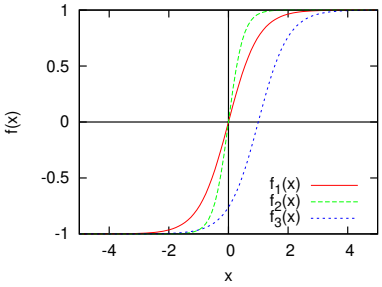
Damit auch inaktive Zustände eine Gewichtsänderung beim Training bewirken, verwendet man den Wertebereich $[-1, +1]$:

$f_1(x) = \tanh(x)$

$f_2(x) = \tanh(2x)$

$f_3(x) = \tanh(x - 1)$

$$\frac{d\ \tanh(x)}{dx} = \frac{1}{\cosh^2(x)}$$



Beispiel: NETtalk[4] lernte die Aussprache von englischem Text.

- Extrem schwierig für symbolbasierten Ansatz, da die englische Aussprache äußerst unregelmäßig ist.
- vorher: komplizierte Programme liefern schlechte Resultate.
- Aussprache eines Buchstabens ist kontextabhängig, daher wird jeweils ein Textfenster von sieben Zeichen untersucht.
- Während der Text dieses Textfenster durchläuft, gibt NETtalk die Aussprache jeweils des mittleren Buchstabens aus.
- Eigenschaften spiegeln Wesen menschlichen Lernens wider:
  - Je mehr Wörter das Netz aussprechen lernt, desto besser ist es in der Lage, neue Wörter korrekt auszusprechen.
  - Das Netz ist fehlerresistent: Werden einige Gewichte eines gut trainierten Netzes zufällig geändert, bleibt die Aussprache gut.
  - Weniger Neuronen in der verborgenen Schicht als in der Eingabeschicht → irgendeine Art von Abstraktion

[4] Sejnowski, Rosenberg: Parallel networks that learn to pronounce english text. Complex Systems, 1987.

Counterpropagation:

- Biologisches Vorbild: Man vermutet, dass im Gehirn spezialisierte Module hintereinander geschaltet sind, um eine gewünschte Berechnung auszuführen.
- Hier: Eingabe-, Kohonen- und Grossberg-Schicht.
- vollständige Vernetzung der Schichten: Jedes Neuron der Eingabeschicht ist mit jedem Neuron der Kohonenschicht verbunden. analog: Kohonen- und Grossberg-Schicht.
- normalisierte Eingabevektoren:

$$x_i^{normalisiert} := \frac{x_i}{\sqrt{x_1^2 + x_2^2 + \ldots + x_n^2}}$$

$\rightarrow$ Der normalisierte Vektor zeigt in die gleiche Richtung wie der ursprüngliche Vektor, hat aber Einheitslänge.

zugrunde liegende Idee:

- zweidimensionaler Fall: Das Skalarprodukt zwischen dem normalisierten Gewichts- und Eingabevektor ist gleich dem Cosinus des Winkels zwischen Eingabe- und Gewichtsvektor.

$$\begin{aligned} \vec{a} \cdot \vec{b} &= |\vec{a}| \cdot |\vec{b}| \cdot \cos(\measuredangle(\vec{a}, \vec{b})) \\ \vec{a} \cdot \vec{b} &= a_x \cdot b_x + a_y \cdot b_y \end{aligned}$$

  Mit Hilfe dieser Formeln können wir den Winkel zwischen den Vektoren bestimmen:

$$\cos(\measuredangle(\vec{a}, \vec{b})) = \frac{a_x \cdot b_x + a_y \cdot b_y}{|\vec{a}| \cdot |\vec{b}|}$$

- Je kleiner der Winkel zwischen Gewichts- und Eingabevektor ist, also je dichter die beiden Vektoren beieinander liegen, umso größer ist der Wert der Aktivierungsfunktion.

zugrunde liegende Idee: (Fortsetzung)

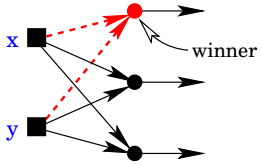
- Das Neuron der Kohonenschicht, dessen Gewichtsvektor die größte Ähnlichkeit zum Eingabevektor zeigt, hat die größte Aktivität.
- The winner takes it all: Legt man einen Eingabevektor an die Kohonenschicht an, so wird die Ausgabe des Neurons auf 1 gesetzt, dessen Gewichtsvektor die größte Ähnlichkeit mit dem Eingabevektor hat.

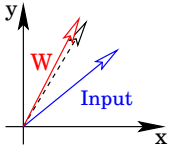
$\Rightarrow$ Betrachtet man jeden einzelnen Gewichtsvektor als Klasse oder Kategorie, so findet eine Klassifizierung des angelegten Eingabevektors in die für ihn am besten passende Kategorie statt.

Eingabeschicht: Verteilt die Eingangssignale an alle Neuronen der nächsten Schicht, der Kohonen-Schicht.

Kohonen-Schicht:

- Lege Eingabe(vektor) an und berechne für alle Neuronen den Wert der Aktivierungsfunktion.
- The winner takes it all: Die Ausgabe des Neurons mit größtem Eingangssignal wird auf 1 gesetzt, alle anderen Ausgaben werden auf 0 gesetzt.





- Die Gewichte des *Gewinner*-Neurons werden korrigiert: Das *Gewinner*-Neuron hat die größte Ähnlichkeit mit dem Eingabevektor; die Gewichtsänderungen machen die Vektoren noch *ähnlicher*.

Training der Kohonen-Schicht: Sei $\vec{w}$ der Gewichtsvektor des *Gewinner*-Neurons, α die Lernrate und $\vec{x}$ der Eingabevektor.

$$\vec{w}^{new} = \vec{w}^{old} + \alpha \cdot (\vec{x} - \vec{w}^{old})$$

Anmerkungen zur Kohonen-Schicht:

- Gewichte werden mit Zufallswerten initialisiert und beim Einsatz der Lernmethode normalisiert.
- Das Kohonen-Netz muss stabil sein, bevor das Grossberg-Netz trainiert werden kann.

Variationen der Kohonen-Schicht:

- „Bewusstseinsparameter“ einführen und in jeder Iteration aktualisieren: Das verhindert, dass einzelne Neuronen zu oft gewinnen. $\rightarrow$ Alle Knoten tragen zur Repräsentation des Musterraums bei.
- Wähle anstatt eines Gewinners eine Menge von ähnlichsten Knoten und ändere deren Gewichte differenziell.
- Teile den Nachbarknoten des Gewinners eine differenzielle Belohnung zu.

Grossberg-Schicht:

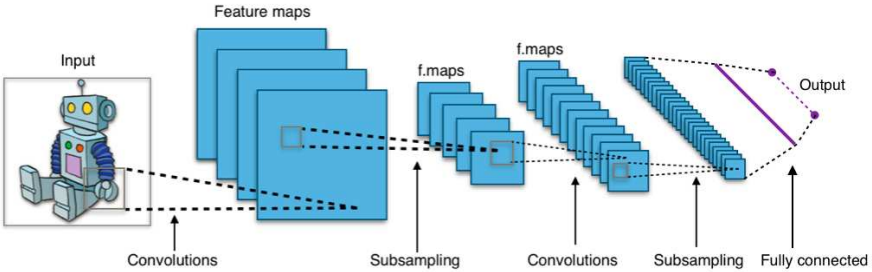
- Die einzige Aufgabe der Grossberg-Schicht ist es, die zuvor gelernten Klassen zu benennen, also die gewünschte Ausgabe zu erzeugen.
- Nach dem Anlegen eines Eingabevektors wird das *Gewinner*-Neuron i der Kohonenschicht bestimmt.
- Anschließend wird das Gewicht v_{ij} zu einem Neuron j der Grossberg-Schicht so geändert, dass der Unterschied zwischen Ist-Ausgabe und Soll-Ausgabe reduziert wird.

Training der Grossberg-Schicht: Sei i das *Gewinner*-Neuron.

$$v_{ij}^{new} = v_{ij}^{old} + \beta \cdot (t_j - v_{ij}^{old})$$

Oft wird $\beta = 0.1$ gesetzt und im Laufe des Trainings, also mit zunehmendem Lernfortschritt, langsam reduziert.

Convolutional Neural Networks sind eine Sonderform von mehrschichtigen Perzeptrons. Daher sind beide Modelle prinzipiell identisch in ihrer Ausdrucksstärke. Bild: wikipedia



Vier Annahmen verringern den Berechnungsaufwand des Netzes und lassen tiefere Netzwerke zu.

- geteilte Gewichte im Convolutional Layer
- Pooling: verwerfe Großteil der Aktivität eines Layers
- ReLu: extrem einfach zu berechnende Aktivierungsfunktion $f(x) = \max\{0, x\}$
- Dropout (entferne zufällige Neuronen) verhindert Overfitting

Der mittels diskreter Faltung ermittelte Input eines jeden Neurons wird von einer Aktivierungsfunktion, oft Rectified Linear Unit (ReLu) in den Output verwandelt.

Intuitiv: Bewege schrittweise eine kleine Faltungsmatrix über die Eingabe.

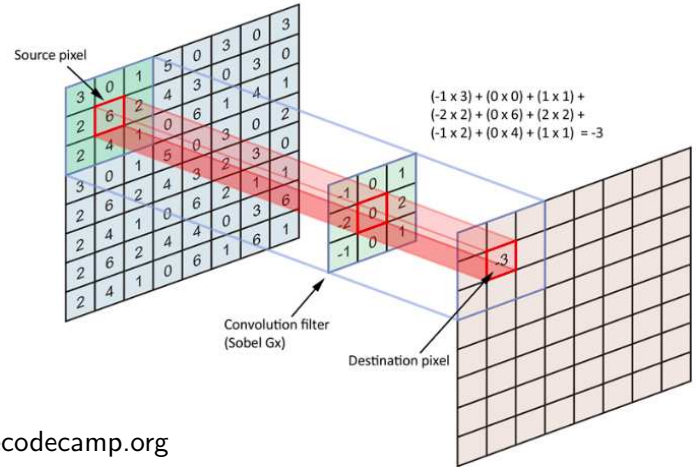
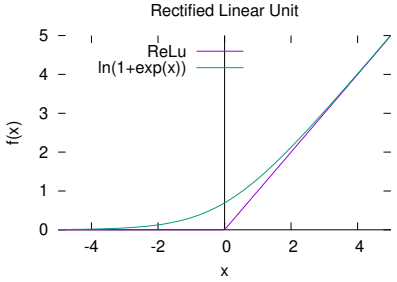


Bild: www.freecodecamp.org

Verschiedene Faltungsmatrizen ergeben unterschiedliche Feature-Maps. Die Gewichte einer Matrix werden mittels Backpropagation trainiert.

- Gewichte aller Neuronen eines Convolutional Layers sind identisch
  → schnelleres Training als bei Perzeptrons
- Backpropagation verlangt die Berechnung der Gradienten
  → nutze differenzierbare Approximation wie $f(x) = \ln(1 + e^x)$



Berechnung wäre viel zu aufwändig → wähle an der einzigen nicht differenzierbaren Stelle als Steigung Null

Im Pooling-Layer werden überflüssige Informationen verworfen.

> Zur Objekterkennung in Bildern etwa, ist die exakte Position einer Kante im Bild von vernachlässigbarem Interesse - die ungefähre Lokalisierung eines Features ist oft hinreichend.

Es gibt verschiedene Arten des Poolings. Mit Abstand am stärksten verbreitet ist das Max-Pooling: Wähle maximale Aktivität aus jedem 2×2 Quadrat.

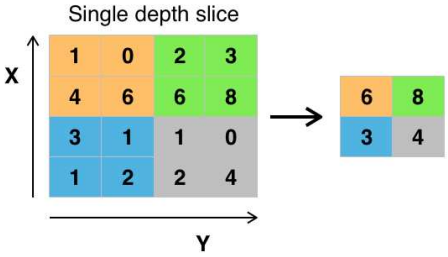


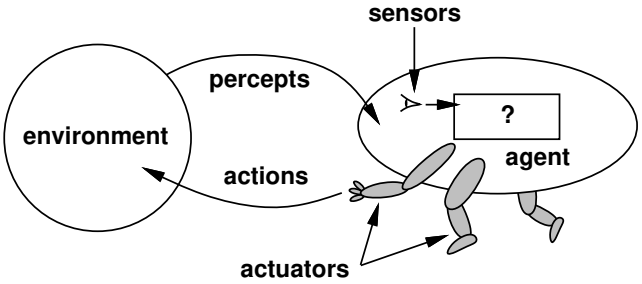
Bild: `https://de.wikipedia.org/wiki/Convolutional_Neural_Network`

Übersicht

Viele Forscher aus den Anwendungsbereichen Robotik, Entwurf von Spielen oder dem Internet haben eine zentrale Wissensbasis oder allgemein gültige Inferenzschemata seit 1995 in Frage gestellt.

Problemlösungsverfahren werden als verteilte Agenten entworfen, die kontextbezogen, autonom und flexibel sind.

Agenten nehmen Umgebung mittels Sensoren wahr und reagieren darauf autonom und rational mittels Effektoren.



Analogie zu agentenbasierten Systemen: Brotbedarf in Städten[5]

- Es wäre extrem schwierig, ein zentrales Planungsmodul zu entwickeln, dass eine Großstadt erfolgreich mit der vielfältigen Auswahl an täglich frischem Brot versorgen könnte.
- Aber die locker koordinierten Bemühungen der vielen Bäcker, Lastwagenfahrer, Lieferanten der Rohmaterialien sowie der Einzelhändler lösen das Problem sehr gut:
  - Es gibt keinen zentralen Plan.
  - Kein Bäcker hat mehr als ein begrenztes Wissen über den Brotbedarf der Stadt.
  - Jeder Bäcker versucht, sein eigenes Geschäft zu optimieren.
  - Die Lösung des globalen Problems ergibt sich aus den kollektiven Aktivitäten der unabhängigen lokalen Agenten.
- Leider neigen solche Ansätze zur Überproduktion.

[5] Luger: Künstliche Intelligenz. Pearson Studium.

Software-Agenten:

- Die Problemlösungsaufgabe wird in ihre verschiedenen Bestandteile aufgeschlüsselt.
- Die Teilaufgaben werden kaum oder gar nicht zentral koordiniert.
- Intelligenz ist in einen Kontext eingebettet. $\rightarrow$ Ein einzelner Problemlöser kann sich einer bestimmten Aufgabe zuwenden, ohne über Informationen zum Status der Lösung in der übergeordneten Problemdomäne verfügen zu müssen.
- Beispiel: Bestellvorgang
  - Ein Agent überprüft die Bestandsinformationen während
  - ein anderer Agent die Kreditwürdigkeit des Kunden prüft.
  - Beide Agenten arbeiten parallel und wissen nichts von der Entscheidung auf höherer Ebene, ob die Bestellung angenommen wird.

Agent:

- Nimmt seine Umgebung über Sensoren wahr.
- Schlussfolgert und entscheidet sich für bestimmte Handlungen.
- Beeinflusst/verändert die Umgebung durch seine Handlungen.

Agenten sind gekennzeichnet durch Rationalität:

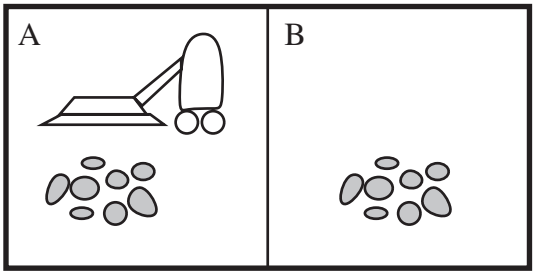
- Zur Optimierung des eigenen Nutzens oder um erfolgreich handeln zu können, muss ein Performanzmaß gegeben sein, ein Maß, um den zu erwartenden Erfolg bestimmen zu können.
- Rationalität bedeutet nicht Allwissenheit, rationales Handeln orientiert sich an
  - dem Performanzmaß,
  - aktuellen und früheren Wahrnehmungen,
  - Wissen über die Umgebung,
  - generellen Handlungsmöglichkeiten.

Eigenschaften von Agenten-Systemen:

- Jeder Agent verfügt nur über unvollständige Informationen. Seine Fähigkeiten reichen nicht aus, um das gesamte Problem zu lösen.
- Es gibt keine globale Instanz zur Steuerung des Systems zur Lösung des Gesamtproblems.
- Wissen und Eingabedaten liegen in dezentraler Form vor.
- Schlussfolgerungsverfahren werden asynchron durchgeführt.

$\rightarrow$ starke Analogie zu objektorientierten Programmen

Staubsauger-Welt[6]:



- Umgebung: Felder A, B, ggf. mit Schmutz
- Sensoren: für Position und Schmutz
- Aktuator: nach rechts/links bewegen; saugen

[6] Russel, Norvig: Artificial Intelligence – A Modern Approach. Prentice Hall

einfaches Agentenprogramm für die Staubsaugerwelt:

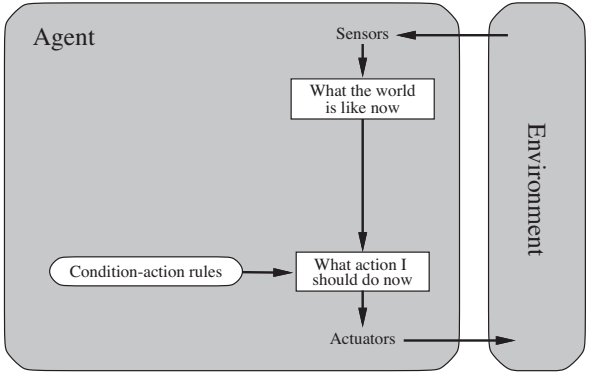
- A oder B schmutzig $\rightarrow$ saugen
- A sauber $\rightarrow$ gehe nach rechts
- B sauber $\rightarrow$ gehe nach links

Anmerkungen:

- Programm berücksichtigt vergangene Wahrnehmungen nicht.
- Wenn alle Felder sauber sind, oszilliert der Sauger zwischen den Feldern.

einfache Reflex-Agenten

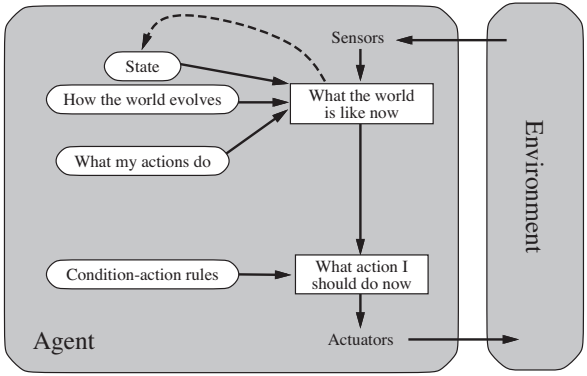
- Aktion hängt nur von aktueller Wahrnehmung ab
- Kein Gedächtnis



Modellbasierte Agenten

- Zustände werden gespeichert
- Modell der Wirklichkeit beschreibt
  - zeitliche Entwicklung der Umgebung unabhängig vom Agenten
  - die Auswirkungen der Aktionen auf die Welt

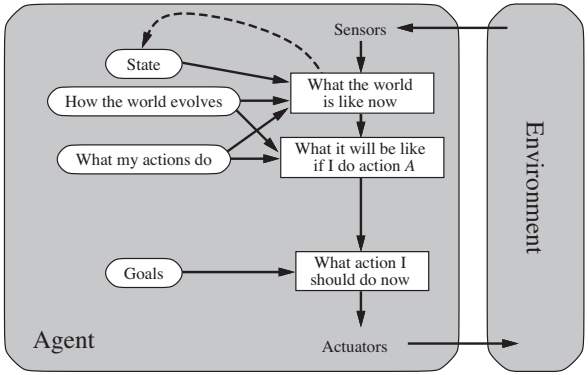
→ verhindert Oszillation



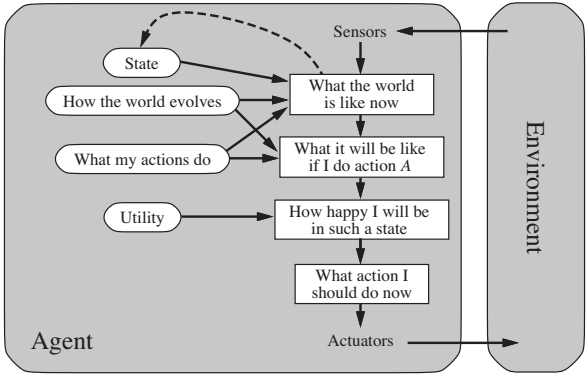
aber keine was-wäre-wenn Simulation

Zielorientierte Agenten

- Bisher: Ziel wurde implizit durch Regeln kodiert.
- Jetzt: explizite Definition von erstrebenswerten Zielen
- Aktionen nicht aus Regeln ableiten, sondern aus Überlegung „was passiert wenn“
  $\rightarrow$ nutze Planen und Suchstrategien



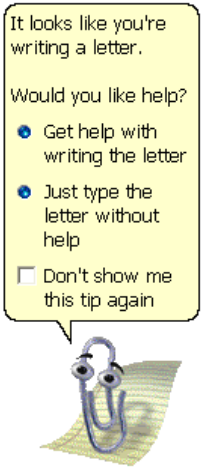
- Lösungen der zielbasierten Agenten sind oft nicht effizient: Jede Entscheidung, die irgendwie zum Ziel führt, ist okay.
- Nutzenfunktion ordnet aktuellen oder zukünftigen Zuständen Bewertungen über die Nützlichkeit zum Erreichen des Ziels zu.
- Vorteil: Günstige Wege zum Ziel können bevorzugt werden.
- Aber: Aufstellen der Nutzenfunktion verlangt viel Information.



- Betonung der Interoperabilität, auch unter nicht-idealen Bedingungen, z.B. ist Wahrnehmung nie perfekt.
- Erfolge in vielen Teilbereichen verstärken wieder den Blick auf das „Ganze“.

Beispiele:

- Microsofts Heftklammer *Clippy*
- Autonome Staubsauger oder Rasenmäher
- Mobile Informationsagenten

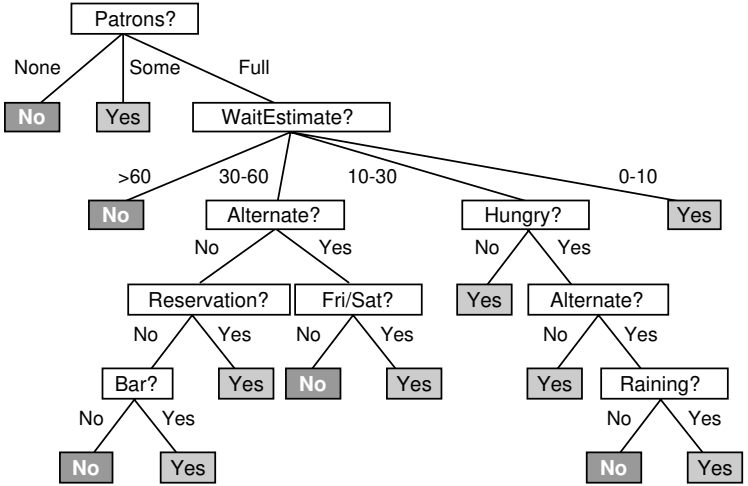


Übersicht

Die Entscheidung, ob wir bei einem Restaurant-Besuch warten wollen, bis ein Tisch frei wird, treffen wir anhand von verschiedenen Kriterien oder Attributen:

- Alt(ernativ): whether there is a suitable alternative restaurant nearby.
- Bar: whether the restaurant has a comfortable bar area to wait in.
- Fri/Sat: true on Fridays and Saturdays.
- Hun(gry): whether we are hungry.
- Pat(rons): how many people are in the restaurant (values: None, Some, Full).
- Price: the restaurant's price range ($+$, $++$, $+++$).
- Rain(ing): whether it is raining outside.
- Res(ervation): whether we made a reservation.
- Type: the kind of restaurant (French, Italian, Thai, or Burger).
- Est(imate): the wait estimated by the host (0-10 minutes, 10-30, 30-60, or > 60). Note that every variable has a small set of possible values; the value here is not an integer, rather it is one of four discrete values.

Entscheidungsbäume werden zur automatischen Klassifikation genutzt und damit auch zur Lösung von Entscheidungsproblemen.



Trainingsmenge von Beispielen:

Ex.	Alt	Bar	Fri	Hun	Pat	Price	Rain	Res	Type	Est	Goal
x_1	Yes	No	No	Yes	Some	+++	No	Yes	French	0-10	$y_1 = \text{Yes}$
x_2	Yes	No	No	Yes	Full	+	No	No	Thai	30-60	$y_2 = \text{No}$
x_3	No	Yes	No	No	Some	+	No	No	Burger	0-10	$y_3 = \text{Yes}$
x_4	Yes	No	Yes	Yes	Full	+	Yes	No	Thai	10-30	$y_4 = \text{Yes}$
x_5	Yes	No	Yes	No	Full	+++	No	Yes	French	> 60	$y_5 = \text{No}$
x_6	No	Yes	No	Yes	Some	++	Yes	Yes	Italian	0-10	$y_6 = \text{Yes}$
x_7	No	Yes	No	No	None	+	Yes	No	Burger	0-10	$y_7 = \text{No}$
x_8	No	No	No	Yes	Some	++	Yes	Yes	Thai	0-10	$y_8 = \text{Yes}$
x_9	No	Yes	Yes	No	Full	+	Yes	No	Burger	> 60	$y_9 = \text{No}$
x_{10}	Yes	Yes	Yes	Yes	Full	+++	No	Yes	Italian	10-30	$y_{10} = \text{No}$
x_{11}	No	No	No	No	None	+	No	No	Thai	0-10	$y_{11} = \text{No}$
x_{12}	Yes	Yes	Yes	Yes	Full	+	No	No	Burger	30-60	$y_{12} = \text{Yes}$

DecisionTree(E : examples, A : attributes, *default* : classification)
if $E = \emptyset$ **then**
  return *default*
else if all E have the same classification c **then**
  return c
else if $A = \emptyset$ **then**
  throw Error ▷ examples with different classification
else
  $a :=$ ChooseAttribute(A, E)
  $T :=$ a new decision tree with root a
  for all attribute value w_i of a **do**
    $E_i := \{e \in E \mid a(e) = w_i\}$
    $T_i :=$ DecisionTree(E_i, $A - \{a\}$, MajorityVal(E))
    add a branch to T with label w_i and subtree T_i
  return T

Der Aufbau (Induktion) der Bäume erfolgt rekursiv.

Dazu müssen Trainingsdaten vorliegen, also zu jedem Objekt des Datensatzes muss die Klassifikation des Zielattributs bekannt sein.

Bei jedem Induktionsschritt wird das Attribut gesucht, mit welchem sich die Trainingsdaten in diesem Schritt bezüglich des Zielattributs am besten klassifizieren lassen. $\rightarrow$ ChooseAttribute

Als Maß für die Bestimmung der besten Klassifizierung kommen Entropie-Maße zur Anwendung.

Das ermittelte Attribut wird nun zur Aufteilung der Daten verwendet. Auf die so entstandenen Teilmengen wird die Prozedur rekursiv angewendet, bis in jeder Teilmenge nur noch Objekte mit einer Klassifikation enthalten sind (Greedy Algorithmus).

Generelles Prinzip des Lernens: *Occams Razor*

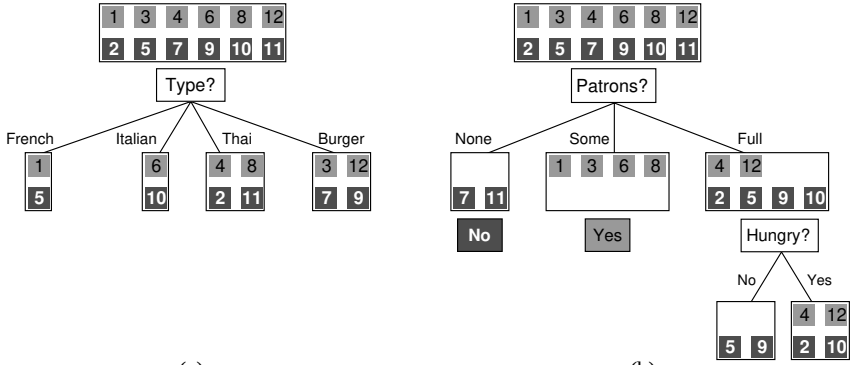
> Bevorzuge die einfachste Hypothese, die konsistent mit allen Beobachtungen ist.

Bei Entscheidungsbäumen wird immer der kleinste Baum gesucht:

> Die Chance, dass irgendein kleiner Baum, der insgesamt viele falsche Entscheidungen trifft, zufälligerweise mit allen Beispielen konsistent ist, ist sehr gering. Ein kleiner Baum, der konsistent mit allen Beispielen ist, ist daher eher korrekt als ein großer Baum.

Alternativ: Konstruiere den Baum so, dass für jedes Beispiel ein Pfad von der Wurzel zu einem Blatt besteht. Dann werden die gegebenen Beobachtungen zwar exakt gespeichert, aber wir können davon keine sinnvolle Generalisierung auf andere Fälle erwarten, da kein Muster aus den Beispielen extrahiert wird. Man spricht hier von einer Überanpassung des Modells an die Daten, vom sogenannten *Overfitting*.

Type is a poor attribute, because it leaves us with four possible outcomes, each of which has the same number of positive as negative examples.



Patrons is a fairly important attribute, because if the value is *None* or *Some*, then we are left with example sets for which we can answer definitively. If the value is *Full*, we are left with a mixed set of examples.

ID3-Algorithmus: Sei S die Menge der Trainingsbeispiele mit ihrer jeweiligen Klassifizierung, sei $a \in A$ das zu prüfende Attribut, sei $V(a)$ die Menge der möglichen Attributwerte von a und sei S_v die Untermenge von S, für die das Attribut a den Wert v annimmt.

Der Informationsgewinn $Gain(S, a)$, der durch Auswahl des Attributs a erzielt wird, errechnet sich dann als Differenz der Entropie von S und der erwarteten Entropie von S bei Fixierung von a:

$$Gain(S, a) = \text{Entropie}(S) - \sum_{v \in V(a)} \frac{|S_v|}{|S|} \cdot \text{Entropie}(S_v)$$

Wähle ein Attribut mit dem maximalen Gewinn aus der folgenden Menge aus:

$$\left\{ a' \in A \mid Gain(S, a') = \max_{a \in A} \{ Gain(S, a) \} \right\}$$

Diese Wahl führt zur Bevorzugung von Attributen mit vielen Wahlmöglichkeiten und damit zu einem breiten Baum.

Der Informationsgehalt $I(c)$ eines Zeichens c hängt von der Wahrscheinlichkeit p_c seines Auftretens ab:

$$I(c) = \log_2\left(\frac{1}{p_c}\right) = \log_2\left(p_c^{-1}\right) = -\log_2(p_c)$$

Shannon definierte die Entropie H als Erwartungswert des Informationsgehalts:

$$\text{Entropie}(S) = H(S) = \sum_{c \in S} p_c \cdot I(c) = -\sum_{c \in S} p_c \log_2(p_c).$$

Setze $0 \cdot \log_2(0) = 0$ entsprechend dem Grenzwert $\lim_{x \to 0} x \log_2(x)$.

Übersicht

Fallbasiertes Schließen

Anstatt Beispiele als Ausgangspunkt für Verallgemeinerungen zu verwenden, wie beim Erstellen von Entscheidungsbäumen, kann man die Beispiele direkt zur Lösung neuer Probleme nutzen.

Ausgangshypothese: Ähnliche Fälle haben ähnliche Lösungen.

- Immanuel Kant: Alles Wissen stammt aus Erfahrung.
- Konfuzius: Der Mensch hat dreierlei Wege klug zu handeln. Erstens durch Nachdenken - das ist der edelste, zweitens durch Nachahmen - das ist der leichteste, und drittens durch Erfahrung - das ist der bitterste.
- Descartes: Jede von mir gelöste Aufgabe wurde zum Muster, welches ich im weiteren für die Lösung anderer Aufgaben benutzte.

„Käpt'n, Käpt'n, wir haben Wasser im Boot!“ – „Dann zieh' doch den Stöpsel 'raus, damit es ablaufen kann“

Case-Based Reasoning (CBR):

- maschinelles Lernverfahren zur Problemlösung durch Analogieschluss.
  → kein regelbasierter, sondern erinnerungsbasierter Prozess
- Zentrales Element: Fallbasis, in der bereits gelöste Probleme als Fall gespeichert sind. Ein Fall besteht mindestens aus einer Problembeschreibung und einer zugehörigen Problemlösung.
- Ziehe zur Lösung eines gegebenen Problems die Lösung eines ähnlichen und früher bereits gelösten Problems heran.

Vorteile:

- Experten müssen keine Regeln explizit festlegen sondern Wissen kann implizit dargestellt werden.
- Wissensbasis wird durch Bearbeitung und Speicherung neuer Fälle ständig erweitert und das System lernt automatisch.

Anwendungsbereiche:

- Rechtsprechung: Argumentation erfolgt häufig und explizit nach Sachlage früherer, vergleichbarer Fälle.
- Medizin: Jeder Patient ist ein Fall, auf den später als Erfahrungswissen zurückgegriffen werden kann.
- e-commerce: Verkaufsagenten suchen Produkte aus dem Angebot heraus, indem sie ähnliche Kunden vergleichen.

  „Kunden, die Produkt A kauften, kauften häufig auch Produkt B“

Wichtig: Erklärungskomponente ist beim CBR sehr ausgeprägt, Problemlösungen sind sehr gut verständlich.

Qualität eines CBR-Systems ist abhängig von

- seiner Erfahrung, also dem Umfang der Fallbasis
- seiner Fähigkeit, neue Situationen mit gespeicherten Fällen in Verbindung zu bringen
- seiner Fähigkeit, alte Lösungen an neue Probleme anzupassen
- der Art und Weise, wie neue Lösungen evaluiert und bei Bedarf korrigiert werden
- der Integration neuer Erfahrungen in die Fallbasis

Prozess mit vier Phasen:

- *Retrieve:* Ausgehend zur gegebenen Problembeschreibung in der Fallbasis ein möglichst ähnliches Problem ermitteln.
  → Ähnlichkeit der Problembeschreibungen bestimmen
- *Reuse:* Die Lösung des ähnlichsten Falls als ersten Lösungsvorschlag übernehmen.
- *Revise:* Falls das aktuelle Problem nicht genau so zu lösen ist wie das frühere, überprüfe die zuvor gewonnene Lösung und passe sie gegebenenfalls an die konkreten Bedingungen an.
- *Retain:* Der überarbeitete Fall wird in der Fallbasis gespeichert und steht damit für zukünftige Anfragen zur Verfügung.
  → Auf diese Weise lernt das System mit jedem weiteren gelösten Problem hinzu und verbessert so seine Leistungsfähigkeit.

Wie kann eine Fallbasis realisiert werden? → nächste Folie

Attribut-Wert-Repräsentation: Repräsentiere einen Fall durch Attribut-Wert-Paare.

- Die Menge der Attribute (A_i) wird entweder für alle Fälle fest vorgegeben oder variiert von Fall zu Fall.
- Jedem Attribut ist ein Wertebereich (Typ) zugeordnet, z.B. Integer oder Text.
- Falls die Attributmenge fest vorgegeben ist, entspricht ein Fall einem n-stelligen Vektor $F = (a_1, \ldots, a_n) \in T_1 \times \ldots \times T_n$, wobei $a_i \in T_i$ (a_i sind die Werte und T_i die Typen/Wertebereiche).
- Unbekannte Attributwerte können durch die Einführung eines speziellen Symbols *unknown* dargestellt werden. Daraus folgt: $a_i \in T_i \cup \{unknown\}$.
- Bei variabler Attributmenge ist ein Fall eine Menge $F = \{A_1 = a_1, \ldots, A_n = a_n\}$ mit $a_i \in T_i$. Attribute, die nicht vorkommen, sind unbekannt.

Vorteile:

- Es ist eine sehr einfache Repräsentation, die leicht zu verstehen ist.
- Eine Ähnlichkeitsbestimmung ist einfach und effizient möglich.
- Eine Speicherung z.B. in Datenbanken ist einfach und daher ist ein effizientes Retrieval möglich.

Nachteil: Es sind keine strukturellen oder relationalen Informationen repräsentierbar.

Objektorientierte Modelle: Zusammengehörige Attribute werden zu Objekten zusammengefasst. Ein Fall besteht somit aus einer Menge von Objekten. Zwischen diesen Objekten bestehen Beziehungen, die auch Relationen genannt werden.

- Eine *a-kind-of*-Relation (taxonomische Relation) drückt Abstraktions- und Verfeinerungsbeziehungen zwischen Objekten der Domäne aus.

  Beispiel: Auto *a-kind-of* Transportmittel.
- Eine *is-part-of*-Relation (kompositionelle Relation) drückt die Zusammensetzung von Objekten aus Teilobjekten aus.

  Beispiel: Fenster *is-part-of* Auto.

Nachteile sind aufwändigere Ähnlichkeitsberechnung und Retrieval.

Wie kann die Reuse-Phase algorithmisch realisiert werden? Wir benötigen dazu einen *Ähnlichkeitsbegriff* und repräsentieren einen Fall x durch ein Tupel $x = (x_1, \ldots, x_n)$, wobei jedes x_i aus dem Wertebereich des i-ten Merkmals stammt.

Bestimme die Gesamt-Ähnlichkeit zweier Fälle $x = (x_1, \ldots, x_n)$ und $y = (y_1, \ldots, y_n)$ durch einen Abgleich der einzelnen Merkmale:

$$sim(x, y) = f(sim_1(x_1, y_1), \ldots, sim_n(x_n, y_n)),$$

Dabei ist sim_i eine reelle Funktion, die die Ähnlichkeit zwischen verschiedenen Werten eines Merkmals bestimmt.

Bei zwei-wertigen Attributen $x_i \in \{0, 1\}$ wird oft die Hamming-Ähnlichkeit genutzt.

$$d_H(x, y) = \sum_{i=1}^{n} |x_i - y_i| = \sum_{i:x_i \neq y_i} 1$$

Umrechnung in ein Ähnlichkeitsmaß zwischen 0 und 1:

$$\begin{aligned} sim_H(x, y) &= 1 - \frac{d_H(x, y)}{n} = 1 - \frac{\sum_{i=1}^{n} |x_i - y_i|}{n} \\ &= \frac{n - \sum_{i=1}^{n} |x_i - y_i|}{n} = \frac{\sum_{i=1}^{n} (1 - |x_i - y_i|)}{n} \end{aligned}$$

Für ein Ähnlichkeitsmaß s gilt:

$$s(i, j) = s(j, i) \qquad 1 = s(i, i) \geq s(i, j) \qquad 0 \leq s(i, j) \leq 1$$

Falls einige Attribute wichtiger als andere sind, verwendet man stattdessen oft die gewichtete Hamming-Ähnlichkeit:

$$\begin{aligned} sim_H^w(x,y) &= \frac{\sum_{i=1}^n w_i \cdot (1 - |x_i - y_i|)}{\sum_{i=1}^n w_i} \\ &= \frac{\sum_{i=1}^n w_i - \sum_{i=1}^n w_i \cdot |x_i - y_i|}{\sum_{i=1}^n w_i} \\ &= 1 - \frac{\sum_{i=1}^n w_i \cdot |x_i - y_i|}{\sum_{i=1}^n w_i} = 1 - \frac{\sum_{i:x_i \neq y_i} w_i}{\sum_{i=1}^n w_i} \end{aligned}$$

Verallgemeinerte Ähnlichkeiten:

$$sim(x,y) = \frac{\sum_{i=1}^n w_i \cdot sim_i(x_i, y_i)}{\sum_{i=1}^n w_i}$$

Oft ist die Lösung des (soeben gefundenen) alten Falls keine mögliche Lösung des aktuellen Problems, weil neue Aspekte vorliegen und keine vollständige Gleichheit gegeben ist.

Als *Adaption* bezeichnet man den Vorgang, eine (vorgeschlagene) Lösung, die sich als nicht ganz richtig für eine (gegebene) Problembeschreibung erweist, so zu manipulieren, dass sie besser zu dem Problem passt.

Dabei kann etwas Neues in die alte Lösung eingefügt werden, etwas wird aus der Lösung entfernt, eine Komponente wird gegen eine andere ausgetauscht oder ein Teil der alten Lösung wird transformiert.

Für einige Anwendungen des fallbasierten Schließens, z.B. Klassikation, Diagnose oder Entscheidungsunterstützung, kann oftmals auf die Adaption der gefundenen Lösung verzichtet werden.

Um Ähnlichkeit und Korrektheit des Auswahlverfahrens bestimmen zu können, ist man auf eine empirische Bewertung angewiesen.

Ein vergleichbares Problem besteht im Bereich des Information-Retrieval. Dort ist eine große Anzahl von Dokumenten gegeben, aus der die für den Anwender relevanten Informationen bestimmt werden müssen.

Dafür gibt es verschiedene Retrieval-Ansätze und -Systeme, die empirisch miteinander verglichen werden müssen.

Hierzu wird eine Leistungskennzahl (Retrieval-Effektivität) von Retrieval-Systemen ermittelt.

Die Begriffe Precision und Recall aus dem Information-Retrieval können auf den Auswahlprozess geeigneter Fälle übertragen werden.

Im Information-Retrieval bezeichnet der Begriff *Precision* die Fähigkeit eines Systems, in einem Anwendungskontext zwischen relevanten und irrelevanten Informationen zu unterscheiden. Die Precision P ist wie folgt definiert:

$$P := \frac{|\{\text{relevante Fälle}\} \cap \{\text{ausgewählte Fälle}\}|}{|\{\text{ausgewählte Fälle}\}|}$$

Falls nur relevante Fälle (aber ggf. nicht alle relevanten Fälle) ausgewählt werden (Precision $= 1$), ist der repräsentierte Ähnlichkeitsbegriff korrekt.

Beispiel: Es gibt 20 relevante Fälle.

- Das System hat 5 relevante Fälle und keine anderen gefunden. $\rightarrow$ Precision $= 1$
- System hat 20 relevante Fälle und 20 nicht relevante gefunden. $\rightarrow$ Precision $= 0{,}5$

Problem: Menge der relevanten Fälle ist oft nicht bekannt. (Internet-Suchmaschine)

Im Information-Retrieval bezeichnet der Begriff *Recall* die Fähigkeit eines Systems, in einem Anwendungskontext die relevanten Informationen zu bestimmen. Der Recall R ist wie folgt definiert:

$$R := \frac{|\{\text{relevante Fälle}\} \cap \{\text{ausgewählte Fälle}\}|}{|\{\text{relevante Fälle}\}|}$$

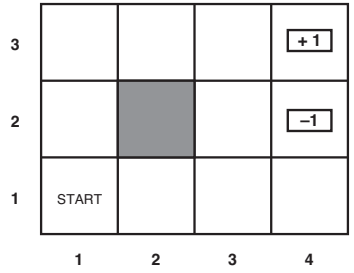
Falls alle relevanten Fälle (und evtl. noch weitere nicht relevante Fälle) bestimmt wurden (Recall $= 1$), ist der repräsentierte Ähnlichkeitsbegriff vollständig.

Beispiel: Es gibt 20 relevante Fälle.

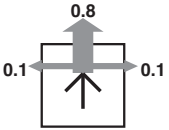
- Das System hat 5 relevante Fälle und keine anderen gefunden. $\rightarrow$ Recall $= 0{,}25$
- System hat 20 relevante Fälle und 500 nicht relevante gefunden. $\rightarrow$ Recall $= 1$

Übersicht

Treffe Entscheidungen in einer stochastischen Umgebung, bei denen der Nutzen eines Agenten von einer Folge von Entscheidungen abhängig ist. Beispiel: Ein Agent befinde sich in der dargestellten 4×3-Umgebung.



(a)



(b)

In jedem Schritt muss er eine Aktion auswählen. Die Interaktion mit der Umgebung endet, wenn der Agent einen Zielzustand erreicht hat (markiert mit $+1$ und -1).

Mögliche Aktionen sind in Zustand s durch $A(s)$ gegeben: hoch, runter, links, rechts.

Die Umgebung sei vollständig beobachtbar, sodass der Agent immer weiß, wo er ist.

Wäre die Umgebung deterministisch, könnte sehr einfach ein Plan (= Folge von Aktionen) zum Erreichen des Endzustands gefunden werden: [hoch, hoch, rechts, rechts, rechts]. Leider sind die Aktionen unzuverlässig:

- Jede Aktion erziele die beabsichtigte Wirkung nur mit einer W'keit von 0,8, sonst bewegt die Aktion den Agenten im rechten Winkel zur beabsichtigten Richtung.
- *Wenn der Agent gegen eine Wand stößt, bleibt er auf demselben Feld stehen.* Daher sind in jedem Zustand s alle Aktionen hoch, runter, rechts, links erlaubt.

Beispiel: Vom Startfeld (1,1) bewegt die Aktion *hoch* den Agenten mit einer W'keit von 0,8 nach (1,2), mit einer W'keit von 0,1 bewegt er sich nach rechts zu (2,1), mit W'keit 0,1 bewegt er sich nach links, stößt gegen die Wand und bleibt in (1,1) stehen.

Also führt die Sequenz [hoch, hoch, rechts, rechts, rechts] um die Barriere herum und erreicht den Zielzustand (4,3) mit einer W'keit von $0,8^5 = 0,32768$. Es besteht auch eine kleine Chance, das Ziel versehentlich auf dem umgekehrten Weg zu erreichen, nämlich mit einer W'keit von $0,1^4 \cdot 0,8 = 0.00008$, insgesamt führt die Sequenz also mit einer W'keit von $0,32776$ zum Ziel.

Übergangsmodell

Das Übergangsmodell (transition model) beschreibt das Ergebnis jeder Aktion in jedem Zustand. Hier ist das Ergebnis stochastisch, daher schreiben wir $P(s' \mid s, a)$, um die W'keit zu bezeichnen, den Zustand s' zu erreichen, wenn die Aktion a im Zustand s ausgeführt wird.

Annahme: Es handelt sich um *Markov-Übergänge*, d.h. dass die W'keit, s' von s aus zu erreichen, nur von s und nicht vom Verlauf früherer Zustände abhängt. Wir können uns $P(s' \mid s, a)$ also als eine 3-dimensionale Tabelle vorstellen.

Da wir nicht an irgendeiner Lösung interessiert sind, sondern an einer möglichst guten, in diesem Fall eine möglichst kurze Folge, müssen wir eine Nutzenfunktion angeben.

Umgebungsverlauf

Da das Entscheidungsproblem sequentiell ist, hängt die Nutzenfunktion von einer Folge von Zuständen - dem Umgebungsverlauf - und nicht von einem einzelnen Zustand ab.

Wir legen fest, dass der Agent in jedem Zustand s eine Belohnung $R(s)$ erhält, die positiv oder negativ sein kann, aber begrenzt sein muss. Für unser spezielles Beispiel ist die Belohnung gleich $-0,04$ in allen Zuständen außer den Endzuständen, die die Belohnungen $+1$ bzw. -1 haben.

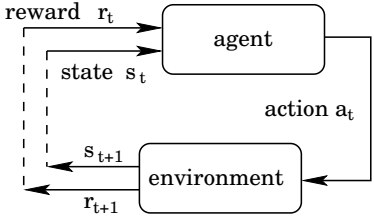
Der Nutzen eines Umgebungsverlaufs ist (vorerst) einfach die Summe der erhaltenen Belohnungen. $\rightarrow$ additive Gewinne

Beispiel: Wenn der Agent nach 10 Schritten den Zustand $+1$ erreicht, beträgt sein Gesamtnutzen $0,6$.

Bemerkung: Die negative Belohnung von -0.04 gibt dem Agenten einen Anreiz, den Endzustand (4,3) schnell zu erreichen.

Ein sequentielles Entscheidungsproblem für eine vollständig beobachtbare, stochastische Umgebung mit einem Markov-Übergangsmodell und additiven Gewinnen heißt *Markov-Entscheidungsprozess* und besteht aus

- einer Menge S von Zuständen und einem Anfangszustand $s_0 \in S$,
- eine Menge A von möglichen Aktionen, wobei $A(s)$ die möglichen Aktionen im Zustand s angibt,
- einem Übergangsmodell mit W'keiten $P(s' \mid s, a)$ und
- einer Gewinnfunktion R, die jedem Zustand s einen Wert $r = R(s)$ zuordnet.



Wie kann eine Lösung für das sequentielles Entscheidungsproblem aussehen?

- Eine festgelegte Aktionssequenz (also ein Plan) kann das Problem nicht lösen, weil der Agent in einem anderen Zustand als dem des Ziels enden könnte.
- Daher muss eine Lösung angeben, was der Agent für jeden Zustand tun soll, den der Agent erreichen kann.

Eine derartige Lösung wird als *Strategie* (englisch policy) bezeichnet. Es ist üblich, eine Strategie als Funktion $\pi : S \to A$ zu bezeichnen, und $\pi(s)$ ist die von der Strategie π empfohlene Aktion für den Zustand s.

Wenn der Agent eine vollständige Strategie hat, dann weiß der Agent unabhängig vom Ergebnis einer Aktion immer, was als nächstes zu tun ist.

Eine Strategie ist im einfachsten Fall eine Hash-Tabelle, in der zu jedem Zustand eine auszuführende Aktion enthalten ist, es können aber auch komplexe Suchverfahren sein.

Bei großen Zustandsräumen wird die Funktion π oft approximiert, dazu eignen sich *Fourierreihen* oder *neuronale Netze*.

Jedes Mal, wenn eine bestimmte Strategie vom Ausgangszustand aus ausgeführt wird, kann die stochastische Natur der Umwelt zu einem anderen Umgebungsverlauf führen.

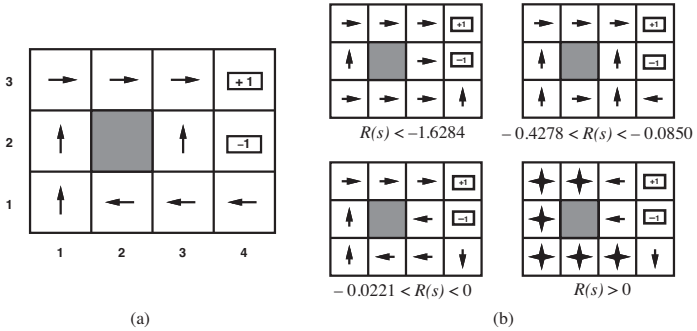
- Die Qualität einer Strategie wird daher anhand des *erwarteten Nutzens* der möglichen Umgebungsverläufe gemessen, die durch diese Strategie möglich sind.
- Eine *optimale Strategie* ist eine Strategie, die den höchsten erwarteten Nutzen bringt. Wir verwenden π^*, um eine optimale Strategie zu bezeichnen.

Ist π^* bekannt, dann entscheidet der Agent, was er tun soll, indem er seine aktuelle Wahrnehmung (über Sensoren) nutzt. Zu dem ermittelten aktuellen Zustand s führt der Agent die Aktion $\pi^*(s)$ aus.

Eine Strategie stellt die Funktion des Agenten explizit dar und ist daher eine Beschreibung eines *einfachen Reflex-Agenten*, der aus den für einen nutzenbasierten Agenten verwendeten Informationen berechnet wird.

Die Frage ist natürlich, wie eine optimale Strategie berechnet werden kann. Das wollen wir uns später ansehen.

Ausgleich zwischen Risiko und Gewinnänderungen



- $R(s) \leq -1,6284$: Der Agent läuft direkt auf den nächsten Ausgang zu.
- $-0,4278 \leq R(s) \leq -0,0850$: Der Agent nimmt den kürzesten Weg zum Zustand $+1$ und akzeptiert das Risiko, in den Zustand -1 zu gelangen.
- $-0,0221 < R(s) < 0$: Gehe keinerlei Risiken ein. In (4,1) und (3,2) bewegt sich der Agent vom Zustand -1 weg.
- $R(s) > 0$: Der Agent vermeidet beide Ausgänge und erhält einen unendlichen Gesamtgewinn, weil er nie einen Terminalzustand erreicht.

Betrachten wir einen Umgebungsverlauf $U([s_0, s_1, \ldots, s_n])$. Wir unterscheiden die Entscheidungsfindung bei einem

- *endlichen Horizont:* Es gibt eine feste Zeit N, was danach passiert, ist egal, also $U([s_0, s_1, \ldots, s_{N+k}]) = U([s_0, s_1, \ldots, s_N])$ für alle $k > 0$.
  - Beispiel: Der Agent befinde sich im Feld (3,1) und $N = 3$. Um Feld (4,3) mit dem Wert $+1$ erreichen zu können, muss der Agent direkt darauf zugehen: Aktion hoch
  - Für $N = 100$ könnte eine sichere Route gewählt werden: Aktion links

  Bei einem endlichen Horizont kann sich die optimale Aktion für einen bestimmten Zustand mit der Zeit ändern $\rightarrow$ optimale Taktik ist nicht stationär.
- *unendlichen Horizont:* Es gibt kein festes Zeitlimit und daher auch keinen Grund, sich zu unterschiedlichen Zeiten im gleichen Zustand unterschiedlich zu verhalten. $\rightarrow$ Die optimale Aktion hängt nur vom aktuellen Zustand ab und die optimale Taktik ist *stationär*.

Achtung: Unendlicher Horizont bedeutet nicht unbedingt, dass alle Zustandsfolgen unendlich sind; es gibt nur keine feste Deadline.

Der Nutzen einer Zustandsfolge $[s_0, s_1, s_2, \ldots]$ ist:

$$U([s_0, s_1, s_2, \ldots]) = R(s_0) + \gamma \cdot R(s_1) + \gamma^2 \cdot R(s_2) + \ldots = \sum_{t=0}^{\infty} \gamma^t \cdot R(s_t)$$

Der *Diskontierungsfaktor* γ ist eine Zahl zwischen 0 und 1.

- Für $\gamma = 1$ erhalten wir unsere oben im Beispiel genutzte Nutzenfunktion: Alle Gewinne werden gleich gewichtet. $\rightarrow$ additive Gewinne
- Für $\gamma < 1$ werden die weit in der Zukunft liegenden Gewinne schwächer gewichtet. $\rightarrow$ verminderte Gewinne
- Wenn γ nahe 0 liegt, werden Gewinne in der fernen Zukunft vernachlässigt.

Der *Diskontierungsfaktor* γ wird eingeführt, damit auch bei unendlich langen Folgen, also insbesondere bei unendlichem Horizont, der Nutzen endlich ist.

- Die Gewinne seien durch $R_{\max}$ begrenzt und es gelte $\gamma < 1$, dann gilt

$$U([s_0, s_1, s_2, \ldots]) = \sum_{t=0}^{\infty} \gamma^t \cdot R(s_t) \leq \sum_{t=0}^{\infty} \gamma^t \cdot R_{\max} = \frac{R_{\max}}{1-\gamma}$$

  aufgrund der geometrischen Reihe:

$$\sum_{t=0}^{\infty} \gamma^t = \frac{1}{1-\gamma}$$

- Wenn die Umgebung Terminalzustände enthält und der Agent irgendwann garantiert einen Terminalzustand erreicht, kann $\gamma = 1$ gewählt werden.

Eine Strategie, die garantiert einen Terminalzustand erreicht, heißt *korrekte Strategie*.

Vergleiche die *erwarteten Nutzen*, die sich bei der Ausführung der Strategien ergeben. Der Nutzen einer Zustandsfolge $[s_0, s_1, s_2, \ldots]$ war definiert als:

$$\begin{aligned} U([s_0, s_1, s_2, \ldots]) &= R(s_0) + \gamma \cdot R(s_1) + \gamma^2 \cdot R(s_2) + \gamma^3 \cdot R(s_3) + \ldots \\ &= R(s_0) + \gamma \cdot (R(s_1) + \gamma \cdot R(s_2) + \gamma^2 \cdot R(s_3) + \ldots) \\ &= R(s_0) + \gamma \cdot U([s_1, s_2, \ldots]) \end{aligned}$$

Der *erwartete Nutzen* $U^\pi(s)$ eines Zustands s bei Ausführen von π beginnend in s ist daher der unmittelbare Gewinn für diesen Zustand plus dem erwarteten verminderten Gewinn des nächsten Zustandes.

$$U^\pi(s) = R(s) + \gamma \cdot \sum_{s'} P(s' \mid s, \pi(s)) \cdot U^\pi(s')$$

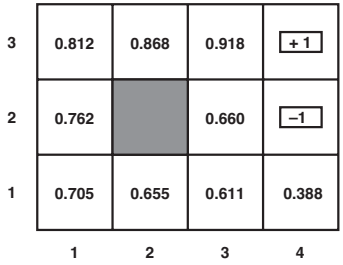
Achtung: $U^\pi(s)$ und $R(s)$ sind unterschiedliche Quantitäten: $R(s)$ ist der „kurzfristige“ Gewinn, sich in s zu befinden, während $U^\pi(s)$ der „langfristige“ Gesamtgewinn ab s ist.

Von allen Strategien, die ein Agent zur Ausführung beginnend in s wählen kann, hat eine den höchsten erwarteten Nutzen; diese nennen wir π_s^*, also $\pi_s^* = \arg\max_\pi U^\pi(s)$.

Die Strategie π_s^* ist insbesondere eine optimale Strategie für den Ausgangszustand s.

Eine optimale Strategie und daher optimale Nutzen erhalten wir, wenn der Agent immer die beste Aktion wählt:

$$U(s) = R(s) + \gamma \cdot \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a) \cdot U(s')$$



Links: Die Nutzen der Zustände für $\gamma = 1$ und $R(s) = -0.04$ für Nichtterminalzustände s.

Bei unendlichen Horizonten ist eine optimale Strategie sogar unabhängig vom Ausgangszustand.

Wir wollen nun eine optimale Strategie berechnen. Für eine optimale Strategie hatten wir bereits festgestellt:

$$U(s) = R(s) + \gamma \cdot \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a) \cdot U(s')$$

Diese Gleichung nennt man *Bellman-Gleichung*. Bei n möglichen Zuständen gibt es n Bellman-Gleichungen, eine für jeden Zustand.

Die n Gleichungen enthalten n Unbekannte, nämlich die Nutzen der Zustände.
Problem: Die Gleichungen sind nicht-linear, weil max kein linearer Operator ist, daher hilft uns die lineare Algebra hier nicht. Lösung: Nutze einen iterativen Ansatz.

$$U_{i+1}(s) = R(s) + \gamma \cdot \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a) \cdot U_i(s')$$

Beginne mit zufälligen Ausgangswerten $U_0(s)$ für die Nutzen der Zustände $s \in S$; die Aktualisierung wird bei jeder Iteration auf alle Zustände gleichzeitig angewendet.

function VALUEITERATION(mdp : MDP, ε : double)
  ▷ mdp contains a set of states S, a set of actions A, probabilities $P(s' \mid s, a)$,
  ▷ the reward function R and the discount factor γ
  initialize $U'(s) := 0$ for each state $s \in S$
  repeat
    $U := U'$, $\delta := 0$
    for all states $s \in S$ **do**
      $U'[s] := R[s] + \gamma \cdot \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a) \cdot U[s']$
      if $|U'[s] - U[s]| > \delta$ **then**
        $\delta := |U'[s] - U[s]|$
  until $\delta < \varepsilon(1 - \gamma)/\gamma$ (only valid for $0 < \gamma < 1$)
  return U

Hier bezeichnet ε den maximal erlaubten Fehler im Nutzen der Zustände.

Frage: Konvergieren die Werte bzw. terminiert der Algorithmus?

Sei (M, d) ein metrischer Raum. Eine Abbildung $\varphi : M \to M$ heißt *Kontraktion*, wenn es eine Zahl $\lambda \in [0, 1)$ gibt, mit der für alle $x, y \in M$ gilt:

$$d(\varphi(x), \varphi(y)) \leq \lambda \cdot d(x, y)$$

Bedeutung: Eine Funktion φ, die bei aufeinanderfolgender Anwendung auf zwei unterschiedlichen Eingaben zwei Ausgabewerte erzeugt, die „enger beisammen“ liegen als die ursprünglichen Eingaben, und zwar mindestens um einen konstanten Faktor.

Beispiel: Die Funktion $\varphi : x \mapsto x/2$ ist eine Kontraktion, denn wenn wir zwei Zahlen durch 2 dividieren, wird ihre Differenz halbiert. Die Funktion hat den *Fixpunkt* 0.

Kontraktionen haben zwei wichtige Eigenschaften:

- Eine Kontraktion φ hat nur einen Fixpunkt; gäbe es zwei Fixpunkte, würden sie sich bei Anwendung der Funktion nicht annähern und φ wäre keine Kontraktion.
- Wenn die Funktion auf ein Argument angewendet wird, muss sie dem Fixpunkt näher kommen (weil sich der Fixpunkt nicht bewegt); eine wiederholte Anwendung einer Kontraktion kommt also letztendlich dem Fixpunkt beliebig nahe.

Sei U_i der Nutzenvektor für alle Zustände in der i-ten Iteration. Einen Iterationsschritt können wir dann als $U_{i+1} := B\,U_i$ für einen geeigneten Operator B schreiben.

Um Distanzen zwischen Nutzenvektoren zu messen, verwenden wir die *Max-Norm*, die die „Länge“ eines Vektors als absoluten Wert seiner größten Komponente misst:

$$\|U\| = \max_s |U(s)|$$

Dann ist zu zeigen, dass $\|B\,U_i - B\,U_i'\| \leq \gamma \cdot \|U_i - U_i'\|$ gilt. Also: Die Aktualisierung in einer Iteration ist eine Kontraktion um einen Faktor γ im Raum der Nutzenvektoren.

Idee: Aus der Mathematik wissen wir, dass für Funktionen $f, g : M \to M$ gilt:

$$|\max_a f(a) - \max_a g(a)| \;\leq \max_a |f(a) - g(a)|$$

Wenn man dies auf $|(B\,U_i - B\,u')(s)|$ anwendet, sieht man, dass der Bellman-Operator eine Kontraktion ist.

Anmerkung: Wenn eine Aktion in einem Zustand deutlich besser als alle anderen Aktionen in dem Zustand ist, muss die genaue Größe der Nutzen der beteiligten Zustände nicht genau sein.

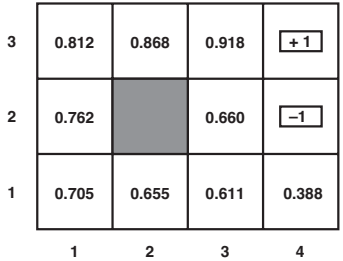
Eigentlich wollen wir eine optimale Strategie π^* berechnen und keine Nutzenfunktion. Aber die Nutzenfunktion U kann dabei helfen.

Die Nutzenfunktion U erlaubt es dem Agenten, Aktionen nach dem *Prinzip des maximalen erwarteten Nutzens* auszuwählen. Der Agent wählt die Aktion, die den erwarteten Nutzen des nachfolgenden Zustandes maximiert:

$$\pi^*(s) = \arg\max_{a \in A(s)} \sum_{s'} P(s' \mid s, a) \cdot U(s')$$

Über den Umweg der Nutzenfunktion haben wir eine optimale Strategie berechnet.

Wert-Iteration



Beispiel: Der Agent befinde sich auf dem Feld (2,1).

$$\pi^*(s) = \arg\max_{a \in A(s)} \sum_{s'} P(s' \mid s, a) \cdot U(s')$$

$$\max_{a \in A(s)} \begin{cases} 0.8 \cdot U((2,1)) + 0.1 \cdot U((1,1)) + 0.1 \cdot U((3,1)) & \text{hoch} \\ \quad = 0.8 \cdot 0.655 + 0.1 \cdot 0.705 + 0.1 \cdot 0.611 = 0.6556 & \\ 0.8 \cdot U((2,1)) + 0.1 \cdot U((3,1)) + 0.1 \cdot U((1,1)) & \text{runter} \\ \quad = 0.8 \cdot 0.655 + 0.1 \cdot 0.611 + 0.1 \cdot 0.705 = 0.6556 & \\ 0.8 \cdot U((1,1)) + 0.1 \cdot U((2,1)) + 0.1 \cdot U((2,1)) & \text{links} \\ \quad = 0.8 \cdot 0.705 + 0.2 \cdot 0.665 = 0.6950 & \\ 0.8 \cdot U((3,1)) + 0.1 \cdot U((2,1)) + 0.1 \cdot U((2,1)) & \text{rechts} \\ \quad = 0.8 \cdot 0.611 + 0.2 \cdot 0.665 = 0.6198 & \end{cases}$$

Eine optimale Strategie kann auch mittels der Strategie-Iteration ermittelt werden. Der Algorithmus beginnt mit einer Strategie π_0 und wechselt zwischen zwei Schritten:

- Auswertung: Für eine gegebene Strategie π_i wird $U_i = U^{\pi_i}$ berechnet, also der Nutzen jedes Zustandes, wenn π_i ausgeführt wird.
- Verbesserung: Es wird eine neue Strategie π_{i+1} berechnet, die basierend auf U_i immer einen Schritt vorausschaut.

Der Algorithmus terminiert, wenn die Verbesserung keine Änderung im Nutzen mehr erbringt. Dann ist die Nutzenfunktion U_i ein Fixpunkt und eine Lösung der Bellman-Gleichungen. Also muss π_i eine optimale Strategie sein.

endlich viele Strategien über einem endlichen Zustandsraum und jede Iteration liefert eine bessere Strategie → Strategie-Iteration terminiert

function PolicyIteration(mdp : MDP)
- ▷ mdp contains a set of states S, a set of actions A, probabilities $P(s' \mid s, a)$,
- ▷ the reward function R and the discount factor γ
- initialize $U'(s) := 0$ for each state $s \in S$
- **repeat**
  - $U :=$ PolicyEvaluation(π, U, mdp)
  - changed := false
  - **for all** states $s \in S$ **do**
    - **if** $\max_{a \in A(s)} \sum_{s'} P(s' \mid s, a) \cdot U[s'] > \sum_{s'} P(s' \mid s, \pi[s]) \cdot U[s']$ **then**
      - $\pi[s] := \arg\max_{a \in A(s)} \sum_{s'} P(s' \mid s, a) \cdot U[s']$
      - changed := true
- **until** not changed
- **return** π

Wie ist die Funktion PolicyEvaluation zu implementieren?

Die auszuführende Aktion ist in jedem Zustand durch die Strategie festgelegt. Bei der i-ten Iteration spezifiziert die Strategie π_i die Aktion $\pi_i(s)$ im Zustand s.

Wir müssen also eine vereinfachte Version der Bellman-Gleichung lösen:

$$U_i(s) = R(s) + \gamma \cdot \sum_{s'} P(s' \mid s, \pi_i(s)) \cdot U_i(s')$$

Diese Gleichungen sind linear. Für n Zustände haben wir n Gleichungen mit n Unbekannten und können mittels Standardmethoden der linearen Algebra in Zeit $\mathcal{O}(n^3)$ die Gleichungen lösen.

Für kleine Werte von n ist das vielleicht akzeptabel, nicht aber für größere Werte. Hier kann eine vereinfachte Auswertung helfen, die iterativ k-mal durchlaufen wird:

$$U_i^{t+1}(s) := R(s) + \gamma \cdot \sum_{s'} P(s' \mid s, \pi_i(s)) \cdot U_i^t(s')$$

Dadurch erhalten wir eine hinreichend genaue Abschätzung der Nutzen.

Übersicht

Im vorherigen Kapitel wurden Belohnungen eingeführt, um optimale Strategien in Markov-Entscheidungsprozessen zu definieren. Eine optimale Strategie maximiert den erwarteten Gesamtgewinn.

Die Aufgabe des verstärkenden Lernens, auch Lernen durch Belohnung genannt, ist es, anhand beobachteter, oft verzögerter Belohnungen eine (fast) optimale Strategie für die Umgebung zu lernen.

Während der Agent im letzten Kapitel ein vollständiges Übergangsmodell besitzt und die Belohnungsfunktion kennt, gehen wir hier von keinerlei Vorwissen aus. Der lernende Agent kennt

- das Übergangsmodell nicht, dass durch die W'keiten $P(s' \mid s, a)$ gegeben ist, nach Ausführen der Aktion a aus dem Zustand s in den Zustand s' zu gelangen.
- eventuell die möglichen Aktionen in einem Zustand nicht.
- auch nicht die Gewinnfunktion R, die die Belohnung $R(s)$ für jeden Zustand $s \in S$ angibt, es ist nur die letzte Belohnung r bekannt.

$\rightarrow$ unbekannter Markov-Entscheidungsprozess

- Ein nutzenbasierter Agent lernt eine Nutzenfunktion für Zustände und verwendet diese, um Aktionen auszuwählen, die den erwarteten Ergebnisnutzen maximieren.

  Dazu wird ein Übergangsmodell benötigt: Welche Auswirkung hat eine Aktion? Welcher Zustand s' wird erreicht, wenn im Zustand s die Aktion a ausgeführt wird? Welche Aktionen sind in einem Zustand erlaubt? Nur dann kann der Agent die Nutzenfunktion auf die Ergebniszustände anwenden.
- Ein Q-Learning-Agent lernt eine Aktion-Nutzen-Funktion $Q : A \times S \to \mathbb{R}$, oft Q-Funktion genannt, die den erwarteten Nutzen einer bestimmten Aktion in einem bestimmten Zustand angibt. Eine Aktion kann in verschiedenen Zuständen unterschiedlichen Nutzen haben.

  Ein Q-Learning-Agent benötigt kein Übergangsmodell. Er kann die erwarteten Nutzen für die ihm zur Verfügung stehenden Möglichkeiten vergleichen, ohne dass er die Ergebnisse kennen muss.
- Ein Reflex-Agent lernt eine Strategie, die direkt von Zuständen auf Aktionen abbildet.

Lernen durch Belohnung ist nicht durch Methoden oder Algorithmen charakterisiert, sondern durch eine Klasse von Lernproblemen. Die Interaktion eines lernenden Agenten mit seiner Umwelt ist als Markov-Entscheidungsproblem formuliert.

Man unterscheidet zwei Arten des verstärkenden Lernens:

- *passives verstärkendes Lernen:* Die Strategie des Agenten ist festgelegt, im Zustand s führt er die Aktion $\pi(s)$ aus. Er hat nur die Aufgabe, die Nutzen von Zuständen (oder von Zustand/Aktion-Paaren) zu lernen, also zu lernen, wie gut die Strategie ist.

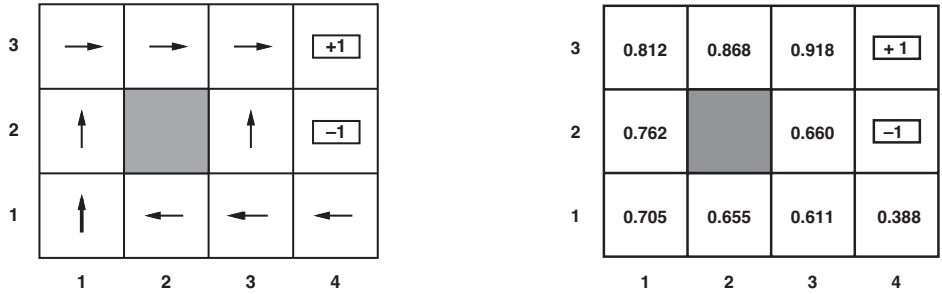
  Es kann erforderlich sein, auch ein Übergangsmodell zu lernen.
- *aktives verstärkendes Lernen:* Der Agent muss außerdem lernen, was zu tun ist, er muss also die Strategie lernen.

  Der wichtigste Aspekt ist die *Exploration*: Der Agent muss so viel wie möglich über seine Umgebung in Erfahrung bringen, um zu lernen, wie er sich darin verhalten soll.

Übersicht: Bei allen Problemen ist die Menge der Zustände S und die Menge der möglichen Aktionen A bekannt.

	außerdem gegeben	gesucht
MDP	Gewinnfunktion R und W'keiten $P(s' \mid s, a)$	optimale Strategie π^* und Nutzenfunktion U
passives Lernen	reward $r = R(s)$ der letzten Aktion, Strategie π	W'keiten $P(s' \mid s, a)$ und Nutzen U^π
aktives Lernen	$r = R(s)$ der letzten Aktion	W'keiten $P(s' \mid s, a)$, optimale Strategie π^* und U

Die W'keiten $P(s' \mid s, a)$ können durch relative Häufigkeiten (Gesetz der großen Zahlen) approximiert werden: Beobachte, wie oft jedes Aktionsergebnis auftritt und schätze die W'keit $P(s' \mid s, a)$ aus der Häufigkeit, wie oft s' erreicht wird, wenn wir Aktion a in Zustand s ausführen.



Adaptive dynamische Programmierung: Aus den beobachteten, relativen Häufigkeiten wird die W'keit $P(s' \mid s, a)$ geschätzt. Basierend auf diesen Schätzungen wird der Nutzen der Zustände berechnet, das Verfahren konvergiert relativ schnell.

$$U^{\pi}(s) = R(s) + \gamma \cdot \sum_{s'} P(s' \mid s, \pi(s)) \cdot U^{\pi}(s')$$

Ein Problem sind große Zustandsräume, weil hier die aus dem letzten Kapitel bekannte Funktion PolicyEvaluation mit Laufzeit $\mathcal{O}(n^3)$ genutzt wird.

function PASSIVE-ADP-AGENT(s' : state, r' : reward)
  ▷ s, a, π, mdp, U, N, M are global variables, and $s \times a \rightarrow s'$ gives reward r'
  if s' is new **then**
    $U[s'] := r'$, $R[s'] := r'$
  if s is not null **then**
    increment $N[s, a]$ and $M[s', s, a]$
    for all t such that $M[t, s, a] \neq 0$ **do**
      $P[t, s, a] := M[t, s, a]/N[s, a]$
  $U :=$ POLICYEVALUATION(π, U, mdp)
  if s' is terminal state **then**
    $s :=$ *null*, $a :=$ *null*
  else $s := s'$, $a := \pi[s']$
  return a

Der Wert $N[s, a]$ gibt an, wie oft die Aktion a in Zustand s ausgeführt wurde, und $M[s', s, a]$ gibt an, wie oft Zustand s' erreicht wurde, wenn die Aktion a im Zustand s angewendet wird. Der Wert $P[t, s, a]$ ist eine Schätzung für $P(t \mid s, a)$.

Idee: Vergleiche Soll- und Ist-Wert und nutze deren Differenz Δ, um den Soll-Wert etwas in Richtung Ist-Wert anzupassen. $\rightarrow$ *Temporal-Difference-Learning*

function PASSIVE-TD-AGENT(s' : state, r' : reward)
- $\triangleright$ s, a, r, π, U, N are global variables, and $s \times a \rightarrow s'$ gives reward r'
- **if** s' is new **then**
  - $U[s'] := r'$
- **if** s is not null **then**
  - increment $N[s]$
  - $\Delta := r + \gamma \cdot U[s'] - U[s]$
  - $U[s] := U[s] + \alpha(N[s]) \cdot \Delta$
- **if** s' is terminal state **then**
  - $s := null$, $a := null$, $r = null$
- **else** $s := s'$, $a := \pi[s']$, $r = r'$
- **return** a

Dabei ist α die *Lernrate*, $U[s]$ ist die Schätzung für $U(s)$ und daher der Soll-Wert, schließlich ist $r + \gamma \cdot U[s']$ der Ist-Wert.

Die *Lernrate* α wird oft als Funktion gewählt, die abfällt, je häufiger der Zustand besucht wurde, z.B. $\alpha(n) = c/c+n$ für eine geeignete Konstante c. Dann konvergiert $U(s)$ auf den korrekten Wert.

Ein passiv lernender Agent hat eine feste Strategie, die sein Verhalten bestimmt. Er berechnet nur den Nutzen der Zustände. Ein *aktiver Agent* muss entscheiden, welche Aktionen er ausführt, also eine optimale Strategie π^* lernen, er muss die Ergebnis-W'keiten für alle Aktionen lernen sowie die Nutzen der Zustände:

$$U(s) = R(s) + \gamma \cdot \sum_{s'} P(s' \mid s, \pi^*(s)) \cdot U(s')$$

Idee: Lerne iterativ wie beim ADP- oder TD-Agenten das Übergangsmodell, also die W'keiten $P(s' \mid s, a)$, und eine Schätzung für U.

Wir bezeichnen einen Agenten als *gierig*, wenn er im nächsten Schritt die Aktion wählt, die (vermeintlich) den erwarteten Nutzen maximiert:

$$\pi^*(s) := \arg\max_{a \in A(s)} \sum_{s'} P(s' \mid s, a) \cdot U(s') \qquad \text{„gierig“}$$

Problem: Solange nicht alle Wege genügend oft gegangen wurden, sind die Schätzungen für die W'keiten $P(s' \mid s, a)$, also das Übergangsmodell, zu ungenau und die gierige Auswahl führt zu suboptimalen Lösungen.

Dies lässt sich vermeiden, wenn der Agent durch *Exploration* die „wahre Umgebung" kennen lernt. Das führt zwar vielleicht zunächst in einen schlechteren Zustand, liefert langfristig aber vielleicht bessere Nutzen.

Eine reine Greedy-Strategie kann dazu führen, dass keine optimale Lösung gefunden wird. Eine reine Exploration zur Verbesserung des eigenen Wissens hat keinen Wert, wenn dieses Wissen niemals genutzt wird. Daher: Je mehr man weiß, desto weniger Exploration ist erforderlich. $\rightarrow$ Der Agent wählt manchmal eine zufällige Aktion aus, sonst folgt er der gierigen Strategie. Formal nutzt man eine Explorationsfunktion f:

$$f(u, n) = \begin{cases} R^+(s) & \text{falls } n < N \\ u & \text{sonst} \end{cases}$$

Dabei ist N ein fest gewählter Wert und R^+ eine optimistische Schätzung der höchsten Belohnung. $\rightarrow$ Jedes Aktion-Zustand-Paar wird mindestens N-mal ausprobiert.

Was erreicht man dadurch? Als Aktualisierungsgleichung erhalten wir:

$$U^+(s) := R(s) + \gamma \cdot \max_{a \in A(s)} f\left(\sum_{s'} P(s' \mid s, a) \cdot U^+(s'),\ N(s, a)\right)$$

Dabei bezeichnet $N(s, a)$ die Anzahl, wie oft die Aktion a im Zustand s ausprobiert wurde.

Literatur zu diesem Thema ist im Web frei verfügbar:

- L.P. Kaelbling, M.L. Littman, and A.W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research (JAIR)*, 4:237–285, 1996.
- R.S. Sutton and A.G. Barto. *Reinforcement learning - an introduction*. Adaptive computation and machine learning. MIT Press, 1998.
- C. Szepesvári. *Algorithms for Reinforcement Learning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2010.

Temporal-Difference-Learning am Beispiel Tic-Tac-Toe: Wie kann der Computer aus vielen Spielen selbständig lernen? Wir gehen hier davon aus, dass wir einen Agenten trainieren wollen, der Spieler X ersetzt.

Zustände stellen wir bspw. als 9-Tupel dar: $(\times, -, \circ, -, \times, -, \circ, -, \times)$

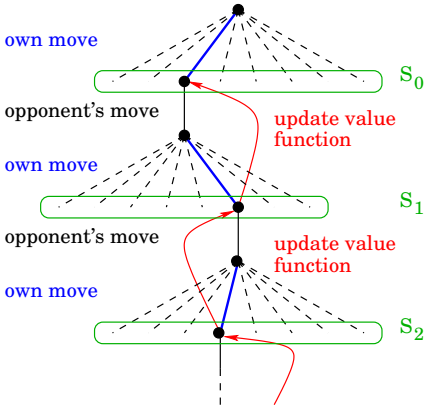
Jedem Zustand $s \in S$ wird eine Zahl $U(s) \in [0, 1]$ zugewiesen. Diese Zahl beschreibt die Chance des Gewinns aus der Sicht eines der Spieler für Zustand s. Implementiert ist U z.B. als Hash-Tabelle.

- Initial erhalten alle Zustände $s \in S$ einen Wert:
  - 3 $\times$'s in einer Zeile, einer Spalte oder einer Diagonalen erhalten den Wert $U(s) = 1$, da hier der Spielgewinn sicher ist.
  - 3 $\circ$'s in einer Zeile, einer Spalte oder einer Diagonalen erhalten den Wert $U(s) = 0$, da hier der Verlust sicher ist.
  - Alle anderen Zustände erhalten den Wert $U(s) = 1/2$, weil wir hier noch nichts über den Ausgang des Spiels wissen.
- Nun müssen sehr viele Spiele gegen einen oder mehrere Gegner gespielt werden.

- Um bei einem solchen Spiel einen Zug auszuwählen, betrachten wir die Werte der möglichen Nachfolgezustände.
  - Wir wählen dabei meistens den Zug, der in den Zustand s mit dem größten Wert $U(s)$ führt (greedy move).
  - Gelegentlich wählen wir allerdings auch einen Zug zufällig aus (Exploration).
  - Das Verhältnis von Greedy-Move und Exploration kann während des Trainings variieren.
- Nach einem Sieg werden die Werte $U(s)$ der Zustände s iterativ verbessert. Bei einem Greedy-Zug $s_t \to s_{t+1}$, nicht bei einem explorativen Zug, wird der Wert $U(s_t)$ in Richtung des Wertes $U(s_{t+1})$ angepasst:

$$U(s_t) \;:=\; U(s_t) + \alpha \cdot [U(s_{t+1}) - U(s_t)] \;=\; (1-\alpha) \cdot U(s_t) + \alpha \cdot U(s_{t+1})$$

Bei großen Zustandsräumen, also bei komplexen Spielen wie Schach oder Go, kann keine Hash-Tabelle für die Werte $U(s)$ genutzt werden, weil es zu viele Zustände gibt. Dann muss die Funktion U approximiert werden, wozu sich *Fourierreihen* oder *neuronale Netze* eignen.



Die Zustände in den grünen Bereichen, wo der Gegner am Zug ist, müssen bewertet werden.

- Diese Zustände müssen für den Agenten vielversprechend sein.
- Bewertungen dieser Zustände anpassen: $U(s_t) := U(s_t) + \alpha \cdot \big(U(s_{t+1}) - U(s_t)\big)$

Einschub: Programm vorführen und besprechen.

- erreichte Zustände werden in einem `std::vector<State>` gespeichert.
- gewählter Diskontierungsfaktor ist eins
- für jeden Spieler eigene Value-Funktion

Q-Lernen: Bezeichne $Q(s, a)$ den Wert der Ausführung von Aktion a in Zustand s. (Der Wert einer Aktion kann in unterschiedlichen Zuständen unterschiedlich sein.)

$$U(s) = \max_{a \in A(s)} Q(s, a)$$

Auf den ersten Blick ist das nur eine andere Form, den Nutzen eines Zustands zu berechnen bzw. zu definieren. Wenn die Q-Werte korrekt sind, muss gelten:

$$Q(s, a) = R(s) + \lambda \cdot \sum_{s'} P(s' \mid s, a) \cdot \max_{a' \in A(s')} Q(s', a')$$

Wie beim ADP-Agenten können wir diese Gleichung direkt als Aktualisierungsgleichung für einen Iterationsprozess verwenden, der genaue Q-Werte berechnet, wenn ein geschätztes Modell gegeben ist. Das bedingt jedoch, dass auch ein Modell gelernt wird, weil die Gleichung $P(s' \mid s, a)$ verwendet.

Aber: Ein TD-Agent, der eine Q-Funktion lernt, braucht weder für das Lernen noch für die Aktionsauswahl ein Modell der Form $P(s' \mid s, a)$, es sind nur Q-Werte notwendig.

$$Q(s,a) := Q(s,a) + \alpha \cdot (R(s) + \gamma \cdot \max_{a' \in A(s')} Q(s',a') - Q(s,a))$$

function Q-Learning-TD-Agent(s': state, r': double)
- $\triangleright$ Q, N, s, a, r are global variables, and $s \times a \to s'$ gives reward r'
- **if** s is terminal state **then**
  - $Q[s, none] := r'$
- **if** s is not null **then**
  - increment $N[s,a]$
  - $Q[s,a] := Q[s,a] + \alpha(N[s,a]) \cdot (r + \gamma \cdot \max_{a' \in A(s')} Q(s',a') - Q(s,a))$
- $s := s'$, $r := r'$
- $a := \arg\max_{a' \in A(s')} f(Q[s',a'], N[s',a'])$
- **return** a

Hier wird wieder die Explorationsfunktion f genutzt und α als abfallende Funktion gewählt.

Schon bei Spielen wie 4-Gewinnt und Othello sind die Suchräume so groß, dass die Nutzenfunktion U oder die Aktion-Nutzen-Funktion Q nicht mehr als Tabelle im Speicher gehalten werden kann und daher nicht mehr exakt ermittelt werden kann.

Wir haben jetzt mehrfach gehört: Für so große Suchräume können die Funktionen durch Fourierreihen oder neuronale Netze approximiert werden.

Stimmt das denn eigentlich? Können wir die Funktionen denn überhaupt berechnen? Haben wir so viel Zeit und Ressourcen?

Bei Tic-Tac-Toe können 5478 verschiedene Zustände[7] erreicht werden (ohne Rotation und Spiegelung), bei 4-Gewinnt etwa $4,5 \cdot 10^{12}$ Zustände[8].

[7] `https://de.wikipedia.org/wiki/Tic-Tac-Toe`

[8] Edelkamp, Stefan; Kissmann, Peter: Symbolic classification of general two-player games. In: Annual Conference on Artificial Intelligence, Springer, 2008

Tic-Tac-Toe: 5478 Zustände
bei 20% Exploration erreicht:

# Spiele	# Zustände
10.000	$\approx$ 4.400
100.000	$\approx$ 5.100
1.000.000	$\approx$ 5.400
5.000.000	5474

4-Gewinnt: $4,5 \cdot 10^{12}$ Zustände
bei 20% Exploration erreicht:

# Spiele	# Zustände
100.000	$\approx$ 1.200.000
1.000.000	$\approx$ 10.000.000
2.000.000	$\approx$ 19.000.000
10.000.000	$\approx$ 82.000.000

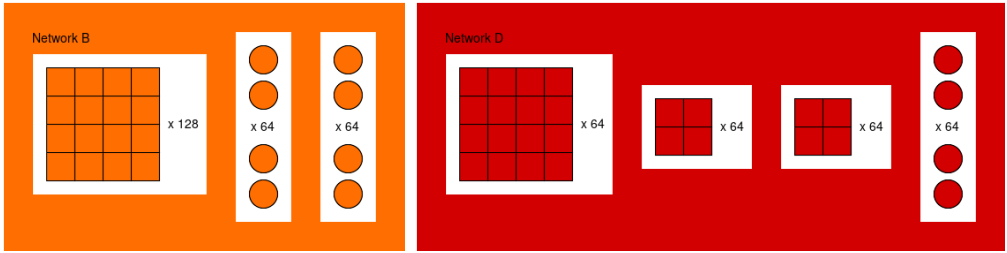
Annahme für 4-Gewinnt: 10^6 Spiele könnten in einer Sekunde gespielt und die Funktionswerte aller dabei erreichten Zustände berechnet werden, und es müssten $45 \cdot 10^{12}$ Spiele gespielt werden, damit jeder Zustand mindestens zehnmal bewertet wird. $\rightarrow$ 521 Tage Laufzeit

Wir benötigen die künstlichen neuronalen Netze also nicht nur wegen des begrenzten Speicherplatzes als Approximation der Value-Funktionen, sondern auch um die fehlenden Werte der Value-Funktionen zu „interpolieren“, also als Interpolation für die Werte der Funktion, die aus Zeitmangel nicht berechnet werden konnten.

Problem: Welche Netze sind geeignet?

4-Gewinnt mittels Convolutional Neural Network[9]:

- naheliegende Idee: Nutze 4×4-Filter, um Reihen von vier Feldern bewerten zu können.



- Netzwerk B: ein Convolutional-Layer und zwei vollständig vernetzte Layer.
- Netzwerk D: ein Convolutional-Layer mit 4×4-Grid, zwei Convolutional-Layer mit 2×2-Grid und ein vollständig vernetzter Layer.

[9] `https://codebox.net/pages/connect4`

Anstatt für große Suchräume die Funktionen durch Fourierreihen oder neuronale Netze zu approximieren, berechnen und speichern wir die Funktion nicht sondern ermitteln den Wert der Funktion durch eine Suche zur Laufzeit.

Time-Space Trade-Offs in a Pebble Game[10]

Let $S(k, n)$ be the set of all directed acyclic graphs with n nodes where each node has indegree at most k. There exists a family of graphs $G_n \in S(2, n)$, $n = 1, 2, \ldots$ and constants c_1, c_2, c_3, $c_2 < c_1$ such that for infinitely many n

- G_n can be pebbled with $c_1\sqrt{n}$ pebbles in n moves.
- G_n can also be pebbled with $c_2\sqrt{n}$ pebbles.
- Every strategy which pebbles G_n using only $c_2\sqrt{n}$ pebbles has at least $2^{c_3\sqrt{n}}$ moves.

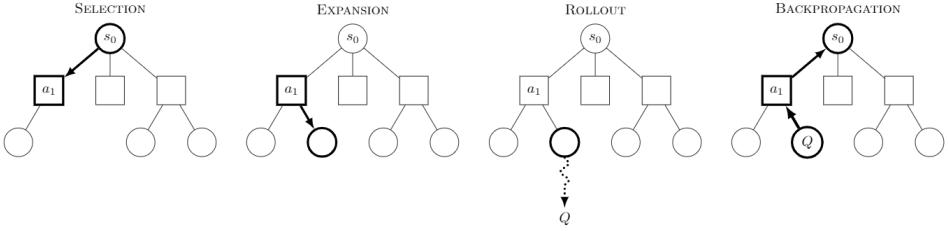
Thus even saving only a constant fraction of the pebbles already forces the time from linear to exponential.

Haben wir nicht genug Speicherplatz, um die Value-Funktion zu speichern, dann müssen wir Zeit investieren, eventuell sogar sehr viel Zeit.

[10] Wolfgang J. Paul, Robert E. Tarjan. Proceedings of the 4th International Colloquium on Automata, Languages and Programming (ICALP), LNCS, volume 52, pages 365-369, Springer, 1977.

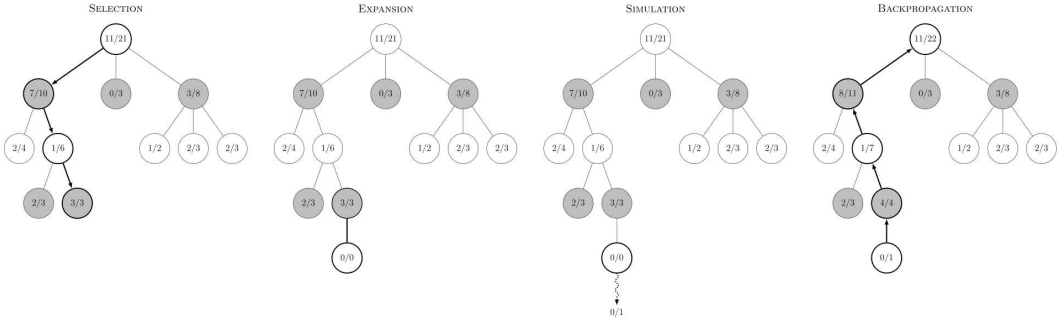
Idee der Monte-Carlo-Baumsuche:

- Wähle zunächst einen Knoten v im Suchbaum aus (SELECTION),
- führe einen noch nicht durchgeführten Zug auf diesen Zustand aus (EXPANSION),
- spiele das Spiel (z.B. zufällig) zu Ende (ROLLOUT oder SIMULATION)
- und bewerte anhand des Ausgangs des Spiels den Knoten v und seine Vorgängerknoten neu. (BACKPROPAGATION)



Quelle: `https://en.wikipedia.org/wiki/Monte_Carlo_tree_search`

Monte-Carlo-Tree-Search



Quelle: `https://en.wikipedia.org/wiki/Monte_Carlo_tree_search`

Beim Backpropagation muss darauf geachtet werden, dass der Gewinn bzw. der Verlust immer abwechselnd verrechnet wird.

Dieser Zyklus aus Selection, Expansion, Roll-Out und Backpropagation muss sehr häufig wiederholt werden. Einfluss auf die Güte haben die Auswahl der Knoten und die Art, ein Spiel zu Ende zu spielen.

Problem: Es müssen riesige Suchbäume analysiert bzw. traversiert werden, d.h. die Laufzeit ist sehr hoch und inakzeptabel für die Spieler. Hier können parallele Algorithmen helfen, die Laufzeiten wesentlich zu verkürzen bzw. eine deutlich bessere Spielstärke zu erreichen.

Das Programm AlphaGo[(11)], dass auf Monte-Carlo-Tree-Search basiert, schlug im Jahr 2016 den damaligen Go-Weltmeister, was bis dahin noch keine künstliche Intelligenz geschafft hatte. Allerdings wurden dazu 1.920 CPUs und 280 GPUs genutzt.

weiteres Problem: Welche Knoten sollen ausgesucht werden? Die, die den höchsten Gewinn versprechen? Vielleicht wurden andere Knoten nur noch nicht ausreichend untersucht. $\rightarrow$ Exploration vs. exploitation

[(11)] `https://en.wikipedia.org/wiki/AlphaGo`

Upper Confidence bound for Trees: Quelle: wikipedia

Choose in each node of the game tree the move for which the expression

$$\frac{w_i}{n_i} + c \cdot \sqrt{\frac{\ln N_i}{n_i}}$$

has the highest value.

- $w_i \rightarrow$ number of wins for the node considered after the i-th move
- $n_i \rightarrow$ number of simulations for the node considered after the i-th move
- $N_i \rightarrow$ total number of simulations after the i-th move run by the parent node of the one considered
- $c = \sqrt{2}$ is the exploration parameter; in practice usually chosen empirically

The first component of the formula above corresponds to exploitation; it is high for moves with high average win ratio. The second component corresponds to exploration; it is high for moves with few simulations.

Einschub: Programme Tic-Tac-Toe und 4-Gewinnt vorführen und besprechen.

Gegenwärtige Trends

- KI ist etabliert:
  - Die meisten Informatik-Abteilungen haben KI-Professur.
  - Wissensbasierte Techniken werden in der Industrie breit genutzt.
- Relative Selbständigkeit anwendungsorientierter Teilbereiche:
  - Wissenssysteme (Diagnostik, Konfigurierung; Scheduling, Planung, Simulation; Wissensportale)
  - Sprachverarbeitung (hohe Qualität bei gesprochener Sprache; einfaches Verstehen geschriebener Texte - shallowparsing)
  - Bildverarbeitung
  - Robotik (Robo-Cup, Aibo, Marsroboter)
- Betonung logischer Grundlagen:
  - Hidden Markov Models
  - Bayessche Netze
  - logikbasierte Wissensrepräsentationen

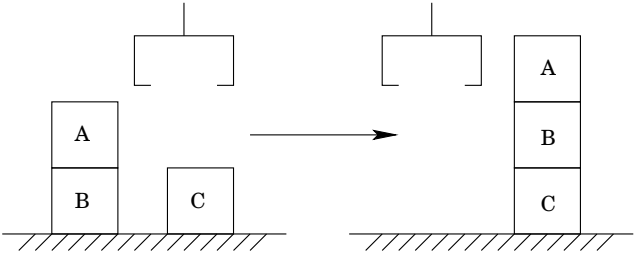
Aktuelle Erfolge und kommerzielle Einsatzgebiete

- Schachprogramme haben Weltmeisterniveau erreicht.
- Natural Language Processing (NLP):
  - natürlichsprachliche Auskunftssysteme wie z.B. Reisebuchung
  - Spracheingabe, automatische Übersetzer
- erfolgreiche Expertensysteme in vielen Bereichen
  - Selbstdiagnose bei großen Maschinen
  - XUMA: eXpertensystem UMweltgefährlichkeit von Altlasten
    Das System soll Fachleute in Behörden und Ingenieurbüros bei der Bewertung von Altlasten und bei der Beurteilung ihrer Umweltgefährdung unterstützen.
  - PROPLANT: Auf Grundlage regionaler Witterung erfolgt eine schlagbezogene Pflanzenschutzberatung. Mittlerweile auch als internetbasierte Version verfügbar.
- automatische Planung und Scheduling in der Raumfahrt
- automatischer Autofahrer, Rasenmäher, Staubsauger
- Dokumentenanalyse: Belegleser
- Fuzzy Control: Fotoapparate, Waschmaschinen, Bremsanlagen

Übersicht

Ziel ist das Aufstellen einer Sequenz von Aktionen, die ausgehend von einem Startzustand einen gewünschten Zielzustand erreichen, unter Berücksichtigung aller Laufzeitbedingungen.

Beispiel: Blocks World



Plan:

```
unstack(A,B)
putdown(A)
pickup(B)
stack(B,C)
pickup(A)
stack(A,B)
```

Zustandsbeschreibung mittels Prädikaten:

- vorher: `onTable(B)`, `onTable(C)`, `upon(A,B)`, `empty()`, `clear(A)`, `clear(C)`
- nachher: `onTable(C)`, `upon(B,C)`, `upon(A,B)`, `empty()`, `clear(A)`

Regeln zur Erzeugung neuer Zustände:

- `pickup(x)`
  - PRECOND: `empty()` $\wedge$ `clear(x)`
  - EFFECT: `hold(x)`
- `putdown(x)`
  - PRECOND: `hold(x)`
  - EFFECT: `empty()` $\wedge$ `onTable(x)` $\wedge$ `clear(x)`
- `stack(x,y)`
  - PRECOND: `hold(x)` $\wedge$ `clear(y)`
  - EFFECT: `upon(x,y)` $\wedge$ `empty()` $\wedge$ `clear(x)`
- `unstack(x,y)`
  - PRECOND: `upon(x,y)` $\wedge$ `clear(x)` $\wedge$ `empty()`
  - EFFECT: `clear(y)` $\wedge$ `hold(x)`

Hinweis: Kleine Buchstaben wie `x` und `y` bezeichnen Variablen, große Buchstaben wie `A` und `B` bezeichnen konkrete Objekte.

Was ist Planen?

- *Wissensbasis:* Definition von Operatoren.
- *Input:* Beschreibung von Start- und Zielsituation.
- *Output:* Der Plan, eine Sequenz von Aktionen.

Anwendungsbeispiele:

- Produktionsplanung
- Reiseplanung
- Spiele: Solitär, Rush Hour, 8-Puzzle usw.

Planen

Erzeugen eines Plans:

- Durchsuche einen Raum möglicher Aktionen so lange, bis eine Abfolge gefunden wird, die notwendig ist, um die betreffende Aufgabe zu bewältigen.
- Der Raum repräsentiert Zustände der Welt, die durch anwenden der Aktionen geändert werden.
- Planung verlässt sich auf Suchtechniken.
- Heuristiken können den Suchraum begrenzen.

Probleme:

- Die Beschreibung der Zustände der Welt ist sehr komplex.
- Wie können Pläne verallgemeinert werden?
- Systemwiederherstellung nach unerwarteten Planfehlschlägen?

Planen mit Situationslogik[12]

- Im Laufe der Zeit ändern die Aktionen `unstack`, `stack`, `pickup` und `putdown` die Welt.
- Der Wahrheitswert von Aussagen wie `onTable(A)` ändert sich und ist abhängig von der aktuellen Situation.
- Prädikate, die Merkmale des Problembereichs beschreiben, erhalten als letztes Argument die Situation, auf die sich die Aussage bezieht.
  - Die Konstante `S0` bezeichnet die initiale Situation.
  - Es gibt ein spezielles Funktionssymbol `do`, das aus einer Aktion `a` und einer Situation `s` eine neue Situation `do(a,s)` bildet.
  - `do(stack(B,A),do(stack(A,C),do(unstack(B,C),S0)))` bezeichnet die Situation, die aus der initialen Situation durch Anwenden der Aktionsfolge `[unstack(B,C), stack(A,C), stack(B,A)]` entsteht.

  Man nennt eine Situation auch Historie.

[12] Beierle, Kern-Isberner: Methoden wissensbasierter Systeme. Vieweg, 2003.

- Ausführungsbedingungen: Die Formel `Poss(a,s)` beschreibt, ob in Situation `s` die Aktion `a` ausgeführt werden kann.
  - $\texttt{Poss(stack(x,y),s)} \equiv \texttt{hold(x,s)} \wedge \texttt{clear(y,s)} \wedge \texttt{x} \neq \texttt{y}$
  - $\texttt{Poss(pickup(x),s)} \equiv \texttt{empty(s)} \wedge \texttt{clear(x,s)}$
- Effektaxiome spezifizieren die Veränderungen, die eine Aktion bewirkt.
  - $\texttt{Poss(stack(x,y),s)} \Rightarrow \texttt{upon(x,y,do(stack(x,y),s))}$
  - $\texttt{Poss(stack(x,y),s)} \Rightarrow \neg\, \texttt{clear(y,do(stack(x,y),s))}$
- Problem: In unserem Beispiel können wir nach der Ausführung von `unstack(A,B)` in der initialen Situation nicht auf den Wahrheitswert von `onTable(B,do(unstack(A,B),S0))` schließen!

  Die Operatoren beschreiben nur, was sich ändert, aber nicht das, was gleich bleibt! $\rightarrow$ Frame-Problem[13]

[13] analogy: Frames of a movie, in which normally very little changes from one frame to the next.

Rahmenproblem:

- Schwierigkeit, die Gültigkeit von Formeln auf die nachfolgende Situation zu übertragen.
- Lösung: Rahmenaxiome beschreiben genau, was durch eine Operation *nicht* verändert wird.
- Beispiel: `pickup(x)`
  - Zwei Blöcke, die durch die pickup-Operation nicht bewegt wurden, sind immer noch aufeinander:
    $\texttt{upon(x,y,s)} \wedge \texttt{x} \neq \texttt{u} \Rightarrow \texttt{upon(x,y,do(pickup(u),s))}$
  - Ein Block befindet sich immer noch auf dem Tisch, wenn er durch die pickup-Operation nicht bewegt wurde:
    $\texttt{onTable(x,s)} \wedge \texttt{x} \neq \texttt{u} \Rightarrow \texttt{onTable(x,do(pickup(u),s))}$
- Die Formulierung in Prädikatenlogik ist in der Praxis zu ineffizient: Sehr viele Rahmenaxiome beschreiben sehr viele (scheinbar redundante) Fakten, die erhalten bleiben.

Stanford **R**esearch **I**nstitute **P**roblem **S**olver:

- Anfang der 1970er Jahre entwickelt, um die Probleme des Situationskalküls zu vermeiden, ohne jedoch die Logik als Basis von Zustandsrepräsentationen aufzugeben.
- Beeinflusste viele andere Planungssysteme, die in der Folgezeit entstanden.
- Beschreibe Zustände durch Mengen von Formeln, die wir Datenbasis nennen.
- Unique-Names-Assumption: Verschiedene Namen repräsentieren verschiedene Objekte.
- Closed-World-Assumption: Es werden nur positive Fakten gespeichert und alles, was nicht in der Datenbasis vorhanden ist, wird als falsch angesehen.

[14] Beierle, Kern-Isberner: Methoden wissensbasierter Systeme. Vieweg, 2003.

Operatoren wie `unstack(x,y)` haben folgende Komponenten:

- Menge von Variablen: allquantifiziert
- Vorbedingungsteil: Konjunktion von positiven und negativen Literalen
  - PRECOND: `upon(x,y)` $\wedge$ `clear(x)` $\wedge$ `empty()`
- Effekte des Operators: positive und negative Fakten, die die entsprechenden Fakten in der Situation überschreiben.
  - ADD: `clear(y)` $\wedge$ `hold(x)`
  - DELETE: `upon(x,y)` $\wedge$ `clear(x)` $\wedge$ `empty()`
- Die Ausführung eines Operators besteht aus zwei Schritten:
  - lösche alle Literale der DELETE-Liste aus der Datenbasis
  - füge alle Literale der ADD-Liste zu der Datenbasis hinzu

  $\rightarrow$ alle anderen Literale der Datenbasis bleiben erhalten
- Eine konkrete Anwendung eines Operators muss voll instanziiert sein: Alle Variablen sind durch Konstanten substituiert.

Vorwärtssuche: (faktenbasiert)

- Gehe von der initialen Datenbasis aus und
- ändere die Datenbasis durch Operatoranwendungen,
- bis die Datenbasis die Zielbeschreibung erfüllt.

$\rightarrow$ Ist in realistischen Anwendungsszenarien nicht praktikabel!

Rückwärtssuche: (zielorientiert)

- Gehe von dem zu erreichenden Ziel aus.
- Um einen Operator anwenden zu dürfen, müssen die Vorbedingungen erfüllt sein.
- Stelle die Vorbedingungen als Teilziel auf.

Mischung aus beiden Verfahren ist wünschenswert, um den Suchraum zu beschneiden $\rightarrow$ means-end-analysis

Mittel-Ziel-Analyse beim General Problem Solver[15]:

- Ermittle syntaktische Unterschiede zwischen dem aktuellen Zustand und dem Zielzustand.
- Wähle einen dieser Unterschiede aus und suche den Operator, der diesen Unterschied reduzieren kann. Probleme:
  - Operator kann im aktuellen Zustand nicht angewendet werden. → erzeuge Teilziel, für den Operator angewendet werden kann
  - Operator erzeugt nicht den exakten Zielzustand. → Zwischenzustand wird neuer Startzustand für zweites Problem
- Hoffnung: GPS eignet sich als allgemeine, von Wissensdomäne unabhängige Architektur zum intelligenten Problemlösen, da nur die syntaktische Form von Zuständen untersucht wird.
- Leider: Es scheint nicht nur eine Heuristik zu geben, die sich erfolgreich auf alle Problemdomänen anwenden lässt.

[15] Newell, Shaw, Simon: Report on a general problem-solving program. Proceedings of the International Conference on Information Processing, 1959

Anmerkungen:

- Linearität:

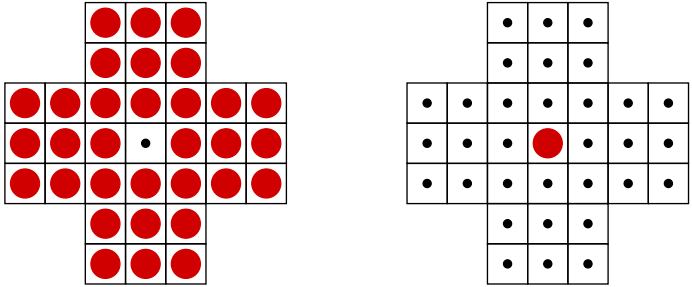
  Einem Plan liegt eine totale lineare zeitliche Ordnung zugrunde. Der Planungsalgorithmus nutzt diese Ordnung.
- Lösung des Rahmenproblems:

  Jeder Operator ändert genau das, was als sein Effekt angegeben ist. Alle nicht angegebenen Fakten bleiben erhalten.

  lässt sich nicht in Prädikatenlogik erster Stufe ausdrücken

Solitär: Gegeben ist ein (englisches) Spielbrett in Kreuzform mit 33 Löchern, in denen 32 Stäbchen stecken. Das Loch in der Mitte ist leer. Man muss nacheinander die Stäbchen entfernen, indem man sie in waagerechter oder senkrechter Richtung überspringt.

Am Ende muss ein Stäbchen in der Mitte übrigbleiben.



Übung 4. Schreiben Sie ein Programm, dass zu einem gegebenen Spielbrett eine Zugreihenfolge ermittelt. Welche Laufzeit ergibt sich?

Erkenne aussichtslose Spielsituation mittels Pagoda-Funktion pg:

- Sei P die Menge der zulässigen Positionen beim Solitär. Dann sei $\sigma : P \to \mathbb{R}$ eine Funktion, so dass für drei benachbarte Positionen a, b, c gilt: $\sigma(a) + \sigma(b) \geq \sigma(c)$
- Sei $B \subset P$ die Menge der aktuell belegten Positionen. Dann ist der Wert der Pagoda-Funktion definiert als

$$pg(B) := \sum_{b \in B} \sigma(b).$$

- Die Werte der Pagoda-Funktion sind monoton fallend im Laufe eines Spiels.

Die Suche kann abgebrochen werden, wenn der Wert der Pagoda-Funktion der aktuellen Spielsituation kleiner ist als der Wert für die Zielsituation.

Übung 5. Implementieren Sie folgende Pagoda-Funktion und vergleichen Sie die Laufzeiten Ihrer Implementierungen.

```
        -1  0 -1
         1  1  1
-1  1  0  1  0  1 -1
 0  1  1  2  1  1  0
-1  1  0  1  0  1 -1
         1  1  1
        -1  0 -1
```

Beispiel: Produktionsplanung[16] für zwei Produkte, zu deren Herstellung drei Maschinen A, B und C benötigt werden:

- maximale monatliche Laufzeiten der Maschinen:

  A: 170 Stunden, B: 150 Stunden, C: 180 Stunden
- Produkt 1: eine Stunde Maschine A, eine Stunde Maschine B
- Produkt 2: zwei Stunden Maschine A, eine Stunde Maschine B und drei Stunden Maschine C

→ Maximiere Profit, wenn 300 Euro mit Produkt 1 und 500 Euro mit Produkt 2 verdient wird,

$$300 \cdot x + 500 \cdot y \to \max$$

ohne dabei die Maschinenlaufzeiten zu überschreiten:

$$\begin{array}{rcrcl} x & + & 2 \cdot y & \leq & 170 \\ x & + & y & \leq & 150 \\ & & 3 \cdot y & \leq & 180 \end{array} \quad \Longleftrightarrow \quad \begin{array}{rcl} y & \leq & \frac{170-x}{2} = 85 - {}^{x}/_{2} \\ y & \leq & 150 - x \\ y & \leq & \frac{180}{3} = 60 \end{array}$$

[16] `http://de.wikipedia.org/wiki/Lineare_Optimierung`

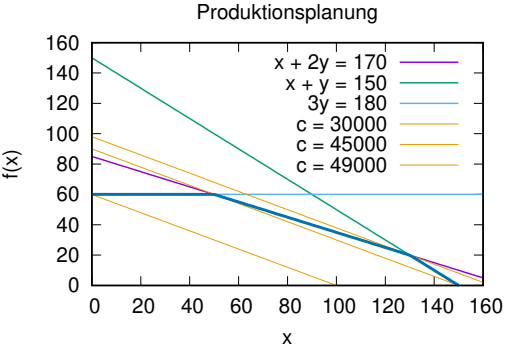
Wir ersetzen in den Ungleichungen das $\leq$ durch ein $=$ und erhalten:

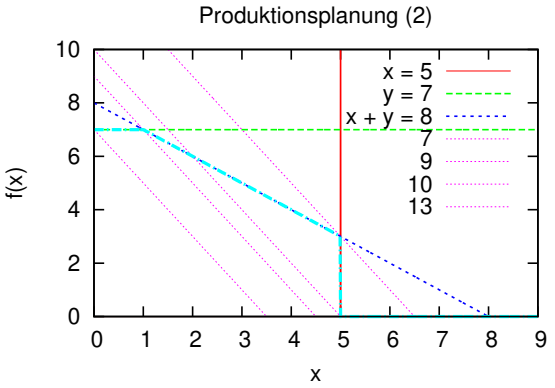
$$y = 85 - x/2 \qquad y = 150 - x \qquad y = 60$$

Betrachte zu maximierende Funktion $300 \cdot x + 500 \cdot y$ für konstante Zielwerte c:

$$300 \cdot x + 500 \cdot y = c \iff y = c/500 - 300/500 \cdot x$$

Wenn wir alle obigen Gleichungen in ein Diagramm eintragen, stellen wir fest, dass der optimale Wert auf einem Eckpunkt des resultierenden Polygons liegt:





Maximiere $2x_1 + x_2$

Randbedingungen:

$0 \leq x_1 \leq 5$

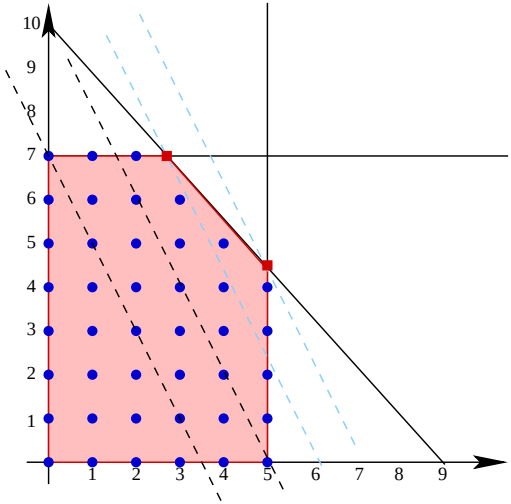
$0 \leq x_2 \leq 7$

$0 \leq x_1 + x_2 \leq 8$

Der optimale Wert liegt auf einem Eckpunkt des Polygons:

$$
\begin{array}{rcrcrcr}
2x_1 + x_2 & = & 7 & \iff & x_2 & = & 7 - 2x_1 \\
2x_1 + x_2 & = & 9 & \iff & x_2 & = & 9 - 2x_1 \\
2x_1 + x_2 & = & 10 & \iff & x_2 & = & 10 - 2x_1 \\
2x_1 + x_2 & = & 13 & \iff & x_2 & = & 13 - 2x_1
\end{array}
$$

Verlangen wir, dass die Lösungen ganzzahlig sind, dann muss der optimale Wert nicht mehr auf einem Eckpunkt des Polygons liegen.



Maximiere $2x_1 + x_2$

Randbedingungen:
$x_1 \leq 5$ und $x_1 \in \mathbb{N}$
$x_2 \leq 7$ und $x_2 \in \mathbb{N}$
$10x_1 + 9x_2 \leq 90$

Viele kombinatorische Probleme können als Integer Programm formuliert werden.

Beispiel: 0/1-Rucksackproblem mit Profiten $p_1, \ldots, p_n$ und den Gewichten $w_1, \ldots, w_n$ der n Objekte sowie dem nicht zu überschreitenden Gesamtgewicht G:

$$x_j = \begin{cases} 1 & \text{das Objekt } j \text{ ist im Rucksack} \\ 0 & \text{sonst} \end{cases}$$

Binary Integer Program: $x_j \in \{0, 1\}\ \forall j$

$$\begin{array}{lll} \max & \sum_{j=1}^{n} p_j x_j & \rightarrow \text{Profit über alle Objekte im Rucksack} \\ s.t. & \sum_{j=1}^{n} w_j x_j \leq G & \rightarrow \text{Gesamtgewicht nicht überschreiten} \end{array}$$

Beispiel: Vertex Cover für Graph $G = (V, E)$ berechnen:

$$x_v = \begin{cases} 1 & \text{der Knoten } v \text{ ist im Vertex Cover} \\ 0 & \text{sonst} \end{cases}$$

Binary Integer Program: $x_v \in \{0, 1\}\ \forall v \in V$

- Minimiere die Anzahl der Knoten im Vertex Cover:

$$\min \sum_{v \in V} x_v$$

- Randbedingung: $x_u + x_v \geq 1 \quad \forall \{u, v\} \in E$

Beispiel: Tourenplanung – Traveling Salesperson Problem für den Graph $G = (V, E, c)$ mit Kostenfunktion c:

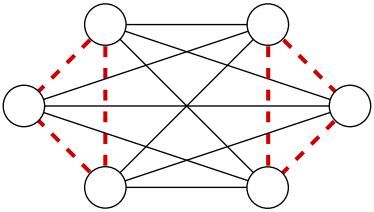
$$x_{ij} = \begin{cases} 1 & \text{die Kante } (i,j) \text{ ist in der Tour} \\ 0 & \text{sonst} \end{cases}$$

Binary Integer Program: $x_{ij} \in \{0,1\}\ \forall i,j$

$$\begin{array}{lll} \min & \sum_{i,j} c_{ij} x_{ij} & \rightarrow \text{Kosten über alle benutzten Kanten} \\ s.t. & \sum_j x_{ij} = 1 \quad \forall i & \rightarrow \text{Knoten } i \text{ genau einmal verlassen} \\ & \sum_i x_{ij} = 1 \quad \forall j & \rightarrow \text{Knoten } j \text{ genau einmal besuchen} \end{array}$$

Mit diesen Beschränkungen sind aber noch Teiltouren möglich, die keine Lösung des Problems beschreiben.

Wir fügen noch die Kurzzyklus-Ungleichungen oder auch Subtour-Eliminationsbedingungen ein.



Sei S eine nicht-leere echte Teilmenge von $\{1, \ldots, n\}$.

- Jede Tour muss S mindestens einmal verlassen:

$$\sum_{i \in S} \sum_{j \notin S} x_{ij} \geq 1$$

- Oder: In jeder Tour kann es nicht mehr als $|S| - 1$ Kanten geben, die Knoten in S verbinden, sonst gibt es eine Teiltour:

$$\sum_{i \in S} \sum_{j \in S} x_{ij} \leq |S| - 1$$

Lineare Programmierung:

gegeben: Matrix $A \in \mathbb{R}^{m,n}$ und zwei Vektoren $b \in \mathbb{R}^m$ und $c \in \mathbb{R}^n$

zulässige Lösung: Vektor $x \in \mathbb{R}^n$, der lineare Bedingungen erfüllt:

$$\begin{array}{ccccccccc} a_{11}x_1 & + & a_{12}x_2 & + & \ldots & + & a_{1n}x_n & \leq & b_1 \\ a_{21}x_1 & + & a_{22}x_2 & + & \ldots & + & a_{2n}x_n & \leq & b_2 \\ \vdots & & \vdots & & & & \vdots & & \vdots \\ a_{m1}x_1 & + & a_{m2}x_1 & + & \ldots & + & a_{mn}x_n & \leq & b_m \end{array}$$

Ziel: finde zulässigen Vektor x, der das Skalarprodukt

$$c^T x = c_1 x_1 + \ldots + c_n x_n$$

maximiert. Dieses Optimierungsproblem wird oft abkürzend als

$$\max\{c^T x \mid Ax \leq b,\ x \geq 0\}$$

geschrieben, wobei $Ax \leq b$ und $x \geq 0$ komponentenweise zu verstehen sind.

Überführen in Standardform:

- Minimieren statt Maximieren:

$$\min\{c^T x \mid Ax \leq b, x \geq 0\} \iff \max\{-c^T x \mid Ax \leq b, x \geq 0\}$$

- Größer-als statt Kleiner-als:

$$a_{i1}x_1 + \ldots + a_{in}x_n \geq b_i \iff -a_{i1}x_1 - \ldots - a_{in}x_n \leq -b_i$$

- Gleichheit statt Kleiner-gleich:

$$a_{i1}x_1 + \ldots + a_{in}x_n = b_i \iff a_{i1}x_1 + \ldots + a_{in}x_n \geq b_i \ \wedge \ a_{i1}x_1 + \ldots + a_{in}x_n \leq b_i$$

- Nichtnegativität der Variablen: ersetze x_i durch $x_i' - x_i''$ mit $x_i', x_i'' \geq 0$

F. Gurski: Efficient Binary Linear Programming Formulations for Boolean Functions. Statistics, Optimization and Information Computing, Vol 2, 274 – 279, 2014.

- Every conjunction $x_1 \wedge x_2$ can be realized by introducing a new variable x_3 and three conditions

$$x_3 - x_1 \leq 0, \quad x_3 - x_2 \leq 0, \quad x_1 + x_2 - x_3 \leq 1$$

such that finally $x_3 = x_1 \wedge x_2$.

- Every n-ary disjunction $x_1 \vee x_2 \vee \ldots \vee x_n$ can be realized by introducing a new variable x_{n+1} and $n+1$ conditions

$$x_1 - x_{n+1} \leq 0 \quad \ldots \quad x_n - x_{n+1} \leq 0 \quad \text{and} \quad x_1 + x_2 + \ldots + x_n - x_{n+1} \geq 0$$

such that finally $x_{n+1} = x_1 \vee x_2 \vee \ldots \vee x_n$.

Die innere Punkte-Methode[17] oder die Ellipsoid-Methode liefern eine Lösung der Linearen Programmierung in polynomieller Zeit.

Probleme:

- Oft müssen die Variablen ganzzahlig sein.
  - Es können nicht 3.7 Flugzeuge gebaut werden.
  - Eine Maschine wird eingesetzt oder nicht.
- TSP: $O(2^{|E|})$ viele Subtouren müssen durch Eliminationsbedingungen ausgeschlossen werden. (sind diese Subtouren überhaupt sinnvoll?)

Relaxation (Aufheben gewisser Einschränkungen):

- Lasse Ganzzahligkeitsbedingungen weg und berechne eine Lösung. Diese Lösung ist Schranke für eigentliches Problem.
- Nutze diese Schranke bei Branch-and-Bound-Verfahren, um den Suchraum zu begrenzen.

[17] N.K. Karmarkar: A new polynomial-time algorithm for linear programming. Combinatorica, 1984.

Mittels GLPK[18] (GNU Linear Programming Kit) können wir bspw. das n-Damen-Problem[19] lösen:

```
# size of the chess board
param n, integer, > 0, default 8;
# x[i,j] = 1 means a queen is placed in square [i,j]
var x{1..n, 1..n}, binary;

# at most one queen in each row, col, diagonal
subject to r{i in 1..n}: sum{j in 1..n} x[i,j] <= 1;
subject to c{j in 1..n}: sum{i in 1..n} x[i,j] <= 1;
s.t. diagl{k in 2-n..n-2}:
    sum{i in 1..n, j in 1..n: i-j == k} x[i,j] <= 1;
s.t. diagr{k in 3..n+n-1}:
    sum{i in 1..n, j in 1..n: i+j == k} x[i,j] <= 1;
```

[18] http://www.gnu.org/software/glpk/

[19] file:///usr/share/doc/glpk-utils/examples/queens.mod

```
# objective is to place as many queens as possible
maximize obj: sum{i in 1..n, j in 1..n} x[i,j];

# solve the problem
solve;

# print optimal solution
for {i in 1..n} {
   for {j in 1..n}
      printf " %s", if x[i,j] then "Q" else ".";
   printf("\n");
}

end;
# Written in GNU MathProg by Andrew Makhorin
```

Übung 6. Starten Sie obiges Programm mittels `glpsol --math queens.mod` für verschiedene Größen des Schachbretts.

GLPK enthält unter `/usr/share/doc/glpk-utils/examples/` Beispielprogramme für viele weitere Probleme:

- `sudoku.mod` ist ein Sudoku-Solver für das klassische Spiel der Größe 9×9.
- `tsp.mod.gz` enthält Code zum Lösen des Travelling Salesperson Problems und einige kleinere Beispielgraphen.

Außerdem finden Sie unter `file:///usr/share/doc/glpk-doc/glpk.pdf` das Reference Manual zum GNU Linear Programming Kit sowie die Beschreibung `file:///usr/share/doc/glpk-doc/gmpl.pdf` der Modeling Language GNU MathProg.

Applegate, Bixby, Chvátal und Cook (1990): Beginn der Entwicklung des Programms Concorde, das an sämtlichen Lösungsrekorden der letzten Jahre beteiligt war.

Reinelt (1991): TSPLIB, Sammlung standardisierter Testinstanzen mit unterschiedlichem Schwierigkeitsgrad, anhand derer verschiedene Forschergruppen ihre Resultate vergleichen konnten.

Cook et al. (2006): beweisbar kürzeste Tour durch 85.900 Knoten eines Layoutproblems für integrierte Schaltkreise, was die bislang größte optimal gelöste TSPLIB-Instanz ist.[20]

Für Instanzen mit mehreren Millionen Städten konnten sie mit Hilfe zusätzlicher Dekompositionstechniken Touren bestimmen, deren Länge beweisbar weniger als 1% vom Optimum entfernt liegt.

[20] Applegate, Bixby, Chvátal, Cook: The Traveling Salesman Problem. A Computational Study. Princeton University Press, 2007.

[21] `http://de.wikipedia.org/wiki/Problem_des_Handlungsreisenden`

Ist es Zufall, dass der optimale Zielwert bei Linearer Programmierung auf einem Eckpunkt des Polygons liegt?

Beispiel:

$$
\begin{array}{rrcrcll}
\textit{maximiere} & 300 \cdot x_1 & + & 500 \cdot x_2 & & & \\
s.t. & x_1 & + & 2 \cdot x_2 & \leq & 170 & (1) \\
 & x_1 & + & x_2 & \leq & 150 & (2) \\
 & & & 3 \cdot x_2 & \leq & 180 & (3)
\end{array}
$$

Wie sieht eine obere Schranke für den Zielwert aus?

$$
\begin{aligned}
500 \cdot (2) &\rightarrow 500x_1 + 500x_2 \leq 75000 \\
300 \cdot (1) &\rightarrow 300x_1 + 600x_2 \leq 51000 \\
200 \cdot (1) + 100 \cdot (2) &\rightarrow 300x_1 + 500x_2 \leq 49000
\end{aligned}
$$

Die *Zielfunktion* kann also maximal den Wert 49000 annehmen!

Wir haben eine Linearkombination der Ungleichungen $Ax \leq b$ gebildet, die gerade der ursprünglichen Zielfunktion entspricht, oder zumindest (komponentenweise) größer als die ursprüngliche Zielfunktion ist, also:

$$\lambda^T \cdot A \geq c^T$$

Da wir eine möglichst kleine obere Schranke für das ursprüngliche Problem haben möchten, müssen wir die neue Zielfunktion $\lambda^T \cdot b$ minimieren!

ursprüngliches Problem:

$$\begin{array}{rrcl} \textit{maximiere} & c^T \cdot x & & \\ s.t. & A \cdot x & \leq & b \\ & x & \geq & 0 \end{array}$$

duales Problem:

$$\begin{array}{rrcl} \textit{minimiere} & \lambda^T \cdot b & & \\ s.t. & \lambda^T \cdot A & \geq & c^T \\ & \lambda & \geq & 0 \end{array}$$

Die beiden Zielwerte sind gleich, wenn beide LP's eine Lösung haben.

Beispiel: `https://de.wikipedia.org/wiki/Simplex-Verfahren`

$$
\begin{array}{rrcrcl}
\textit{maximiere} & z = 300 \cdot x_1 & + & 500 \cdot x_2 & & \\
\textit{s.t.} & x_1 & + & 2 \cdot x_2 & \leq & 170 \\
 & x_1 & + & x_2 & \leq & 150 \\
 & & & 3 \cdot x_2 & \leq & 180 \\
 & x_1 \geq 0, x_2 \geq 0 & & & &
\end{array}
$$

Zunächst überführen wir das gegebene lineare Programm durch Einführen von Schlupf-Variablen (engl. slack variables) in ein Gleichungssystem.

$$
\begin{array}{rrcrcrcl}
\textit{maximiere} & z = 300 \cdot x_1 & + & 500 \cdot x_2 & & & & \\
\textit{s.t.} & x_1 & + & 2 \cdot x_2 & + & y_A & = & 170 \\
 & x_1 & + & x_2 & + & y_B & = & 150 \\
 & & & 3 \cdot x_2 & + & y_C & = & 180 \\
 & x_1 \geq 0, x_2 \geq 0, y_A \geq 0, y_B \geq 0, y_C \geq 0 & & & & & &
\end{array}
$$

Das Gleichungssystem ist unterbestimmt, da wir zusätzlich zu den Variablen noch so viele Schlupfvariablen wie Gleichungen haben:

$$\begin{array}{rrcrcrcl} \textit{maximiere} & z = 300 \cdot x_1 & + & 500 \cdot x_2 & & & & \\ s.t. & x_1 & + & 2 \cdot x_2 & + & y_A & = & 170 \\ & x_1 & + & x_2 & + & y_B & = & 150 \\ & & & 3 \cdot x_2 & + & y_C & = & 180 \end{array}$$

Wir schreiben dieses Gleichungssystem so um, dass die Schlupf-Variablen auf der linken Seite stehen.

$$\begin{array}{rcrcrcr} z & = & 0 & + & 300 \cdot x_1 & + & 500 \cdot x_2 \\ y_A & = & 170 & - & x_1 & - & 2 \cdot x_2 \\ y_B & = & 150 & - & x_1 & - & x_2 \\ y_C & = & 180 & - & & - & 3 \cdot x_2 \end{array}$$

Eine initiale Lösung des Gleichungssystems kann man direkt ablesen: $x_1 = 0$, $x_2 = 0$, $y_A = 170$, $y_B = 150$ und $y_C = 180$.

Diese gefundene Lösung versuchen wir schrittweise zu verbessern, indem die Werte der Variablen x_1 und x_2 erhöht werden.

zunächst: Wie groß darf x_1 bzw. x_2 werden, ohne dass eine Restriktion verletzt wird?

- x_1: da $x_2 = 0$ ist, erhalten wir folgende Gleichungen

$$\begin{aligned} 0 \leq y_A &= 170 - 1 \cdot x_1 \Rightarrow x_1 \leq 170 \\ 0 \leq y_B &= 150 - 1 \cdot x_1 \Rightarrow x_1 \leq 150 \end{aligned}$$

- x_2: da $x_1 = 0$ ist, erhalten wir folgende Gleichungen

$$\begin{aligned} 0 \leq y_A &= 170 - 2 \cdot x_2 \Rightarrow x_2 \leq 170/2 \\ 0 \leq y_B &= 150 - 1 \cdot x_2 \Rightarrow x_2 \leq 150 \\ 0 \leq y_C &= 180 - 3 \cdot x_2 \Rightarrow x_2 \leq 180/3 \end{aligned}$$

Wert der Zielfunktion $z = 300 \cdot x_1 + 500 \cdot x_2$ ist 45.000 für $x_1 = 150$ bzw. 30.000 für $x_2 = 60$. Also tauschen wir x_1 und y_B.

Da der Wert y_B auf 0 sinkt, wenn wir $x_1 = 150$ setzen, schreiben wir die entsprechende Zeile um, so dass x_1 links steht

$$y_B = 150 - x_1 - x_2 \iff x_1 = 150 - y_B - x_2$$

und ersetzen x_1 in obigen Gleichungen:

$$\begin{array}{rcrcrcr} z & = & 0 & + & 300 \cdot (150 - y_B - x_2) & + & 500 \cdot x_2 \\ y_A & = & 170 & - & 1 \cdot (150 - y_B - x_2) & - & 2 \cdot x_2 \\ y_C & = & 180 & - & 0 \cdot (150 - y_B - x_2) & - & 3 \cdot x_2 \end{array}$$

Als neues Gleichungssystem erhalten wir:

$$\begin{array}{rcrcrcr} z & = & 45.000 & - & 300 \cdot y_B & + & 200 \cdot x_2 \\ y_A & = & 20 & + & 1 \cdot y_B & - & 1 \cdot x_2 \\ x_1 & = & 150 & - & 1 \cdot y_B & - & 1 \cdot x_2 \\ y_C & = & 180 & + & 0 \cdot y_B & - & 3 \cdot x_2 \end{array}$$

Links stehen die Variablen mit Wert ungleich 0.

Da die jetzige Zielfunktion noch einen positiven Koeffizienten enthält, kann die Lösung noch verbessert werden. Wir führen also einen weiteren Schritt durch, bei dem jetzt nur noch der Wert der Variablen x_2 geändert werden kann. Da $y_B = 0$ ist, gilt:

$$\begin{array}{rcll} 0 \leq y_A & = & 20 - 1 \cdot x_2 & \Rightarrow \quad x_2 \leq 20 \\ 0 \leq x_1 & = & 150 - 1 \cdot x_2 & \Rightarrow \quad x_2 \leq 150 \\ 0 \leq y_C & = & 180 - 3 \cdot x_2 & \Rightarrow \quad x_2 \leq 180/3 \end{array}$$

Wir setzen $x_2 = 20$ und y_A wird zu 0, daher schreiben wir die entsprechende Zeile um

$$y_A = 20 + 1 \cdot y_B - 1 \cdot x_2 \iff x_2 = 20 - y_A + y_B$$

und setzen dann x_2 in die Gleichungen ein:

$$\begin{array}{rcrcrcl} z & = & 45.000 & - & 300 \cdot y_B & + & 200 \cdot (20 - y_A + y_B) \\ x_1 & = & 150 & - & 1 \cdot y_B & - & 1 \cdot (20 - y_A + y_B) \\ y_C & = & 180 & + & 0 \cdot y_B & - & 3 \cdot (20 - y_A + y_B) \end{array}$$

Wir erhalten so schließlich:

$$\begin{aligned} z &= 49.000 - 100 \cdot y_B - 200 \cdot y_A \\ x_2 &= 20 + 1 \cdot y_B - 1 \cdot y_A \\ x_1 &= 130 - 2 \cdot y_B + 1 \cdot y_A \\ y_C &= 120 - 3 \cdot y_B + 3 \cdot y_A \end{aligned}$$

Auch hier stehen links die Variablen mit Wert ungleich 0. Da die Zielfunktion keine positiven Koeffizienten mehr enthält, ist eine optimale Lösung erreicht.

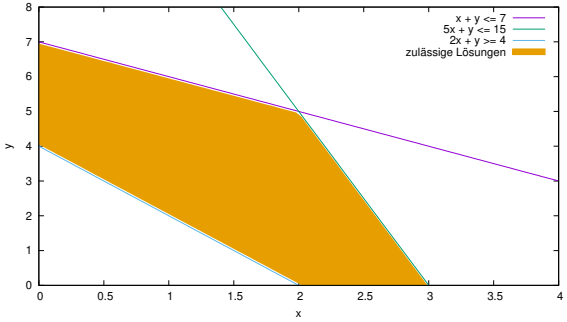
Als Ergebnis stellen wir fest:

- Die Maschinen A und B sind voll ausgelastet ($y_A = y_B = 0$),
- der Profit beträgt 49.000,
- Maschine C könnte noch 120 Stunden genutzt werden.

Initiale Lösung für Simplex-Methode

Bisher sind wir davon ausgegangen, dass $x = y = 0$ eine gültige, initiale Lösung des Modells ist, die Basisvariablen also alle auf Null gesetzt werden können. Wenn das aber nicht möglich ist, dann muss zunächst eine initiale Lösung bestimmt werden.

$$
\begin{array}{rcrcr}
3x & + & 2y & \rightarrow & \max \\
x & + & y & \leq & 7 \\
5x & + & y & \leq & 15 \\
2x & + & y & \geq & 4
\end{array}
$$



Überführe das Modell in die Standardform, füge eine neue Variable z ein und ändere die Zielfunktion.

$$
\begin{array}{rcrcrcr}
 & & & - & z & \rightarrow & \max \\
x & + & y & - & z & \leq & 7 \\
5x & + & y & - & z & \leq & 15 \\
-2x & - & y & - & z & \leq & -4
\end{array}
$$

Das modifizierte Modell hat immer eine Lösung, da z immer so groß gewählt werden kann, dass die linken Seiten der Ungleichungen beliebig klein werden und daher die Ungleichungen erfüllt sind. Hier erhalten wir als Lösung $x = 2$, $y = 0$ und $z = 0$.

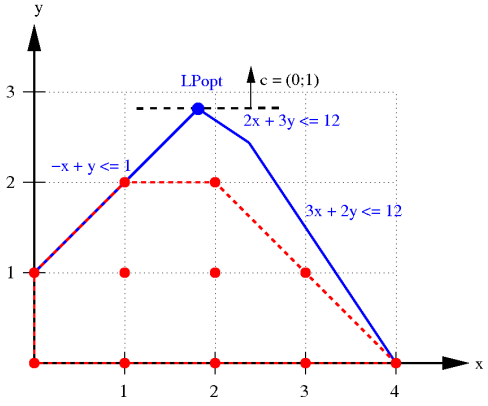
$$\begin{array}{rcrcrcr} & & & & -z & \to & \max \\ x & + & y & - & z & \leq & 7 \\ 5x & + & y & - & z & \leq & 15 \\ -2x & - & y & - & z & \leq & -4 \end{array}$$

Das ursprüngliche Modell hat eine gültige Lösung genau dann wenn das modifizierte Modell eine Lösung mit $z = 0$ hat. Die Lösung des modifizierten Modells kann als initiale Lösung des ursprünglichen Modells genutzt werden.

$$\begin{array}{rcrcr} x & + & y & \leq & 7 \\ 5x & + & y & \leq & 15 \\ 2x & + & y & \geq & 4 \end{array} \qquad \begin{array}{rcrcr} 2 & + & 0 & \leq & 7 \\ 10 & + & 0 & \leq & 15 \\ 4 & + & 0 & \geq & 4 \end{array}$$

Anschließend arbeitet das Simplex-Verfahren so, wie es oben besprochen wurde.

Wir lösen zunächst das gegebene Problem mittels Simplex-Verfahren, lassen also die Ganzzahligkeitsbedingungen weg.



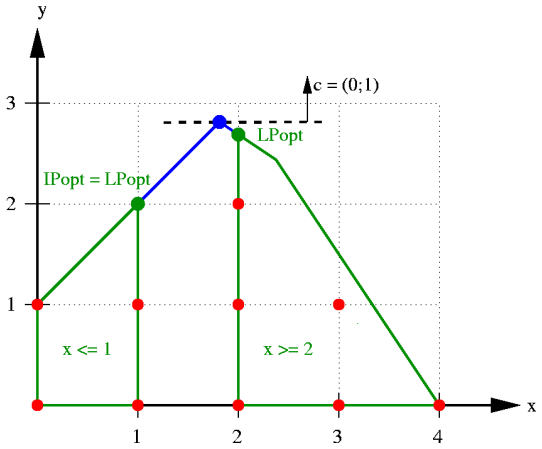
Die Lösung des relaxierten Problems liefert eine obere Schranke für den optimalen Wert der ganzzahligen Lösung, denn:

> Jede zulässige Lösung des ganzzahligen Problems ist auch eine zulässige Lösung für das relaxierte Problem.

Der Wert der Lösung des relaxierten Problems wird *duale Schranke* genannt.

Quelle: `https://de.wikipedia.org/wiki/Ganzzahlige_lineare_Optimierung`

Für eine Basis-Variable x_i, die die Ganzzahligkeitsbedingung verletzt, fügen wir eine obere bzw. untere Schranke als Bedingung hinzu. Wir erhalten so zwei Teilprobleme, die wir wieder mit dem Simplex-Verfahren lösen.



Quelle: `https://de.wikipedia.org/wiki/Ganzzahlige_lineare_Optimierung`

Bei jeder Verzweigung (branch) wird also der Wertebereich einer Variablen eingeschränkt. Dies wird solange wiederholt, bis eine ganzzahlige Lösung für das Teilproblem gefunden wurde. Dann endet in diesem Zweig die Suche.

Der optimale Wert ergibt sich aus dem Maximum der gefundenen, ganzzahligen Lösungen der Teilprobleme.

Eine bereits gefundene ganzzahlige Lösung eines der bearbeiteten Zweige ist eine untere Schranke des Problems und wird *primale Schranke* genannt.

Ist die Lösung des relaxierten Teilproblems schlechter als die primale Schranke, dann wird die Bearbeitung des aktuellen Zweigs abgebrochen (bound).

Gap (absolut): Differenz zwischen dualer und primaler Schranke.
Gap (relativ): Gap (absolut) dividiert durch primale Schranke.

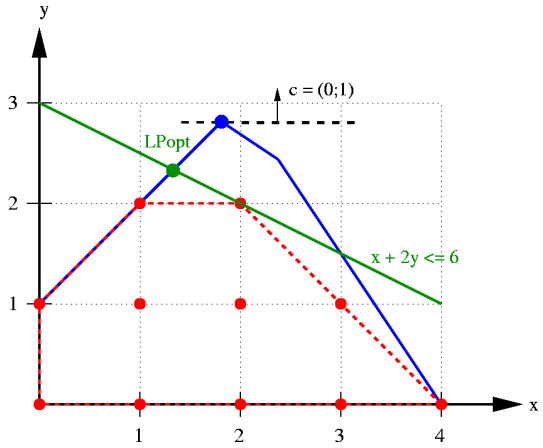
Die Laufzeit des Branch-and-Bound-Verfahrens ist in der Regel zu groß, weil zu wenig vom Suchbaum abgeschnitten wird.

Gute IP-Löser verbessern daher die duale Schranke mittels Schnittebenenverfahren.

Eine *Schnittebene* ist eine zusätzliche Ungleichung, die von allen zulässigen Punkten des IPs erfüllt wird, aber nicht von der aktuellen LP-Lösung. (siehe nächste Folie)

Wird die Ungleichung dem LP hinzugefügt, muss beim erneuten Lösen eine andere Lösung herauskommen.

Der Gap verkleinert sich, wenn eine kleinere, duale Schranke durch die Schnittebene (in der Grafik grün gezeichnet) gefunden wird.



Der Gap verkleinert sich auch dann, wenn eine bessere primale Schranke durch Lösen eines Teilproblems gefunden wird.

Lösen wir mittels Simplex-Verfahren das relaxierte lineare Programm aus dem obigen Bild.

$$\begin{aligned} \textit{maximiere} \quad & z = 1 \cdot x + 5 \cdot y \\ \textit{s.t.} \quad & -1 \cdot x + 1 \cdot y \leq 1 \\ & 3 \cdot x + 2 \cdot y \leq 12 \\ & 2 \cdot x + 3 \cdot y \leq 12 \\ & x \geq 0,\ y \geq 0 \end{aligned}$$

Nach Einfügen der Schlupfvariablen erhalten wir:

$$\begin{aligned} \textit{maximiere} \quad & z = 1 \cdot x + 5 \cdot y \\ \textit{s.t.} \quad & -1 \cdot x + 1 \cdot y + a = 1 \\ & 3 \cdot x + 2 \cdot y + b = 12 \\ & 2 \cdot x + 3 \cdot y + c = 12 \\ & x \geq 0,\ y \geq 0,\ a \geq 0,\ b \geq 0,\ c \geq 0 \end{aligned}$$

Als initiale Simplex-Tabelle ergibt sich:

$$\begin{array}{rcrcrcrl} z & = & 0 & + & 1 \cdot x & + & 5 \cdot y & (0) \\ a & = & 1 & + & 1 \cdot x & - & 1 \cdot y & (1) \\ b & = & 12 & - & 3 \cdot x & - & 2 \cdot y & (2) \\ c & = & 12 & - & 2 \cdot x & - & 3 \cdot y & (3) \end{array}$$

Eine initiale Lösung ergibt sich für $x = 0$ und $y = 0$. Anhand der Nebenbedingungen ermitteln wir, wie groß y werden darf:

$$\begin{array}{rcrcl} 0 \leq a & = & 1 - 1 \cdot y & \Rightarrow & y \leq 1 \\ 0 \leq b & = & 12 - 2 \cdot y & \Rightarrow & y \leq 6 \\ 0 \leq c & = & 12 - 3 \cdot y & \Rightarrow & y \leq 4 \end{array}$$

Wir setzen $y = 1$, formen Gleichung (1) nach y um, setzen $y = 1 + x - a$ in (0), (2) und (3) ein und erhalten:

$$
\begin{array}{rcrcrcrl}
z & = & 0 & + & 1 \cdot x & + & 5 \cdot (1 + x - a) & \\
 & = & 5 & + & 6x & - & 5a & (0') \\
y & = & 1 & + & 1 \cdot x & - & 1 \cdot a & \\
 & = & 1 & + & x & - & a & (1') \\
b & = & 12 & - & 3 \cdot x & - & 2 \cdot (1 + x - a) & \\
 & = & 10 & - & 5x & + & 2a & (2') \\
c & = & 12 & - & 2 \cdot x & - & 3 \cdot (1 + x - a) & \\
 & = & 9 & - & 5x & + & 3a & (3')
\end{array}
$$

Wir ermitteln nun, wie groß x werden darf:

$$
\begin{array}{rcrcl}
0 \leq y & = & 1 + 1 \cdot x & \Rightarrow & x \leq \infty \\
0 \leq b & = & 10 - 5 \cdot x & \Rightarrow & x \leq 2 \\
0 \leq c & = & 9 - 5 \cdot x & \Rightarrow & x \leq \frac{9}{5}
\end{array}
$$

Wir setzen $x = 9/5$, formen Gleichung (3') nach x um, setzen $x = 9/5 + 3/5a - 1/5c$ in (0'), (1') und (2') ein und erhalten:

$$
\begin{array}{rcrcrcr}
z &=& 5 &+& 6\cdot(9/5+3/5a-1/5c) &-& 5\cdot a \\
&=& 15\,4/5 &-& 1\,2/5a &-& 1\,1/5c \\
y &=& 1 &+& 1\cdot(9/5+3/5a-1/5c) &-& 1\cdot a \\
&=& 2\,4/5 &-& 2/5a &-& 1/5c \\
b &=& 10 &-& 5\cdot(9/5+3/5a-1/5c) &+& 2\cdot a \\
&=& 1 &-& 3a &+& c \\
x &=& 9/5 &+& 3/5\cdot a &-& 1/5\cdot c \\
&=& 1\,4/5 &+& 3/5a &-& 1/5c
\end{array}
$$

Um Gomory-Cuts zu berechnen, trennen wir den ganzzahligen Teil vom nicht-ganzzahligen Teil einer der obigen Gleichungen:

$$y = 2\,4/5 - 2/5a - 1/5c \iff y + 2/5a + 1/5c = 2 + 4/5$$

Da y ganzzahlig ist und $y \geq 0$, $a \geq 0$, $b \geq 0$ gilt, erhalten wir:

$$y \leq 2 \quad \text{und} \quad 2/5a + 1/5c \geq 4/5$$

Die letzte Zeile ist ein Gomory-Cut.

$$2/5a + 1/5c \geq 4/5 \iff 2a + 1c \geq 4$$

Wollen wir diesen Cut in unser Bild einzeichnen, müssen wir die zusätzliche Nebenbedingung bezüglich der x, y-Werte umrechnen:

$$\begin{aligned} -x + 1y + a &= 1 \quad \iff \quad -2x + 2y + 2a = 2 \\ 2x + 3y + c &= 12 \end{aligned}$$

Wenn wir diese Gleichungen addieren und den Gomory-Cut einsetzen, erhalten wir:

$$5y + 2a + c = 14 \iff 5y \leq 10 \iff y \leq 2$$

Durch diesen Cut schneiden wir die aktuelle Lösung des LP-Problems ab, behalten aber alle ganzzahligen Lösungen.

Übung 7. Lösen Sie das folgende lineare Programm mittels des Simplex-Verfahrens und bestimmen Sie einen Gomory-Cut.

$$\begin{array}{lrcl} \textit{maximiere} & y & & \\ s.t. & -x+y & \leq & 1 \\ & x+2y & \leq & 10 \end{array}$$

Zeichnen Sie den Gomory-Cut sowie die obigen Geraden in ein Koordinatensystem ein.

Anmerkung: Bei ganzzahliger Programmierung können die Schlupfvariablen auch ganzzahlig gewählt werden. Betrachten wir dazu rationale Koeffizienten wie im folgenden Beispiel:

$$\begin{aligned} 1/3x + 2/5y &\leq 5/2 \qquad (1) \\ 1/4x - 2/3y &\leq 6/5 \qquad (2) \end{aligned}$$

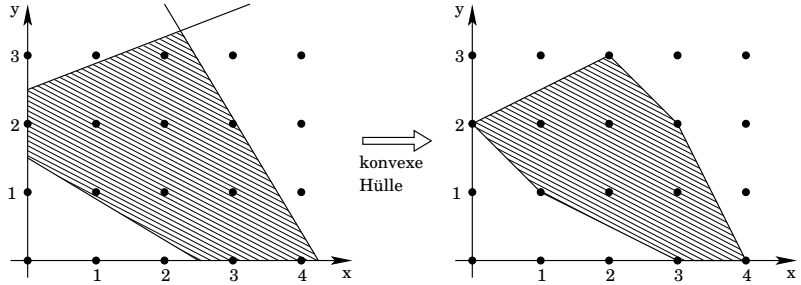
Diese Gleichungen können äquivalent umformuliert werden:

$$\begin{aligned} 5x + 6y &\leq 75/2 = 37,5 \qquad (1) \cdot 3 \cdot 5 \\ 3x - 8y &\leq 72/5 = 14,4 \qquad (2) \cdot 4 \cdot 3 \end{aligned}$$

Weil x und y ganzzahlig sind und $x \geq 0$, $y \geq 0$ gilt, können wir weiter vereinfachen und erhalten ganzzahlige Schlupfvariablen:

$$\begin{aligned} 5x + 6y &\leq \lfloor 37,5 \rfloor = 37 \quad \Rightarrow \quad 5x + 6y + a = 37 \\ 3x - 8y &\leq \lfloor 14,4 \rfloor = 14 \quad \Rightarrow \quad 3x - 8y + b = 14 \end{aligned}$$

Werden geeignete Schnittebenen gefunden, dann kann ein Integer-Programm auch ohne Branch-and-Bound gelöst werden:



Besonders effektiv sind daher Schnittebenen, die möglichst weit oben im Baum, also möglichst nahe bei der Wurzel angewendet werden können.

$\rightarrow$ Alle MIP-Solver nutzen eine Vorverarbeitung (presolve).

Cover-Cuts werden auch Knapsack-Cuts genannt. Sei $N \subseteq \{1, \ldots, n\}$ und

$$\sum_{j \in N} a_j \cdot x_j \leq b; \qquad x_i \in \{0, 1\}; \qquad a_i > 0; \qquad b > 0$$

eine Randbedingung des Modells. Dann ist $C \subseteq N$ ein *Cover*, wenn $\sum_{j \in C} a_j > b$ gilt.

$\rightarrow$ Füge die Schnittebene $\sum_{j \in C} x_j \leq |C| - 1$ dem Modell hinzu.

Beispiel: Für die Randbedingung $4x_1 + 2x_2 + 3x_3 + 6x_4 \leq 8$ mit binären Variablen $x_1, x_2, x_3, x_4 \in \{0, 1\}$ erhalten wir unter anderem folgende Cover-Cuts:

$$\begin{aligned} 3 + 6 > 8 \quad &\rightarrow \quad x_3 + x_4 \leq 1 \\ 4 + 6 > 8 \quad &\rightarrow \quad x_1 + x_4 \leq 1 \\ 4 + 2 + 3 > 8 \quad &\rightarrow \quad x_1 + x_2 + x_3 \leq 2 \end{aligned}$$

Zwei binäre Variablen sind *inkompatibel*, wenn nicht beide gleichzeitig den Wert eins einnehmen können.

$$x + y \leq 1 \quad \rightarrow \quad x \text{ und } y \text{ sind inkompatibel}$$

Eine *Clique* C ist eine Menge von paarweise inkompatiblen Variablen.

$\rightarrow$ Füge die Schnittebene $\sum_{j \in C} x_j \leq 1$ dem Modell hinzu.

Beispiel: Die binären Variablen der folgenden Randbedingungen bilden eine Clique.

$$x + y \leq 1 \qquad x + z \leq 1 \qquad y + z \leq 1$$

Füge die Schnittebene $x + y + z \leq 1$ dem Modell hinzu.

Bei großen oder generierten Modellen können unnötige bzw. redundante Bedingungen vorkommen, die zunächst entfernt werden:

- Eine Bedingung wie $x + y \leq 2$ für zwei binäre Variablen stellt keine Einschränkung dar und kann entfernt werden.
- Beschreiben Randbedingungen das Gleiche wie bei $x + y \geq 7$ und $2x + 2y \geq 14$, dann können alle bis auf eine aus dem Modell entfernt werden.
- Bei einer Randbedingung wie $7x_1 + 3x_2 + 5x_3 \geq 13$ und binären Variablen $x_1, x_2, x_3 \in \{0, 1\}$ muss für eine gültige Lösung $x_1 = x_2 = x_3 = 1$ gelten.

$\rightarrow$ Weniger Randbedingungen und Variablen bedeuten weniger Rechenaufwand und Speicherbedarf zur Lösung des Modells.

Wird eine Randbedingung durch eine andere *dominiert*, dann kann die dominierte Bedingung aus dem Modell entfernt werden. Von den Bedingungen

$$\begin{array}{rcrcrcll} 3x_1 & + & 2x_2 & + & 5x_3 & \leq & 14 & \text{(Zeile } i\text{)} \\ 4x_1 & + & 2x_2 & + & 7x_3 & \leq & 13 & \text{(Zeile } j\text{)} \end{array}$$

mit $b_i \geq b_j$ und $a_{ik} \leq a_{jk}$ für alle $1 \leq k \leq n$ kann Zeile i entfernt werden.

Weitere Schritte der Vorverarbeitung ergeben sich oft durch Auf- bzw. Abrunden von nicht ganzzahligen Werten, denn ganzzahlige Vielfache von ganzzahligen Werten sind ebenfalls ganzzahlig:

- Eine nicht-ganzzahlige Grenze einer ganzzahligen Variablen kann abgeschnitten werden: Für $x \in \mathbb{N}_0$ kann $x \leq 2,5$ durch $x \leq \lfloor 2,5 \rfloor = 2$ ersetzt werden.
- Randbedingung $3x_1 + 6x_2 + 3x_3 \leq 8$ mit ganzzahligen Variablen $x_1, x_2, x_3 \in \mathbb{N}_0$ kann zu $3x_1 + 6x_2 + 3x_3 \leq 6$ modifiziert werden, da 8 kein Vielfaches von 3 ist.

Da große numerische Werte beim Rechnen oft Probleme machen, kann der letzte Schritt verbessert werden, indem durch den gemeinsamen Teiler der Koeffizienten der linken Seite geteilt und dann die rechte Seite abgerundet wird: Wegen

$$3x_1 + 6x_2 + 3x_3 \leq 8 \iff \frac{3}{3}x_1 + \frac{6}{3}x_2 + \frac{3}{3}x_3 \leq \frac{8}{3}$$

können wir die Randbedingung durch $x_1 + 2x_2 + x_3 \leq \lfloor 8/3 \rfloor = 2$ ersetzen.

Durch den letzten obigen Schritt können sich Randbedingungen ergeben, durch die einige Variablen eindeutig bestimmt sind:

> Für ganzzahlige Variablen $x_1, x_2 \in \mathbb{N}_0$ ist die Randbedingung $2x_1 + 2x_2 \leq 1$ äquivalent zu $x_1 + x_2 \leq 1/2$ und daher gilt $x_1 + x_2 \leq \lfloor 1/2 \rfloor = 0$.
>
> Also gilt $x_1 = x_2 = 0$ für eine zulässige Lösung.

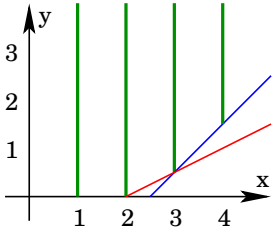
Sinn der „Verbesserungen“ des Modells durch Schnittebenen: Je weniger sich die Lösungsräume der relaxierten und der ganzzahligen Modelle unterscheiden, umso weniger Verzweigungen sind beim Branch-and-Bound nötig.

Der GLPK-Solver kennt außer den Cover- und Clique-Cuts außerdem noch Gomory-Mixed-Integer- und Mixed-Integer-Rounding-Cuts. Der CBC-Solver kennt noch weitere Cuts wie Flow-Cover-, Split- und Capacity-Cuts.

Für eine Randbedingung wie $x - y \leq b$ mit $x \in \mathbb{N}_0$ und $y \in \mathbb{R}_0^+$ fügen wir die Schnittebene $x - \frac{1}{1-f} \cdot y \leq \lfloor b \rfloor$ in das Modell ein, wobei $f := b - \lfloor b \rfloor$ der Nachkommateil von b ist.

Beispiel: Gegeben sei die Randbedingung $x - y \leq 2,5$ mit $x \in \mathbb{N}_0$, $y \in \mathbb{R}_0^+$.

Dann gilt $f = 2,5 - \lfloor 2,5 \rfloor = 0,5$ sowie $\frac{1}{1-f} = \frac{1}{0,5} = 2$ und wir erhalten die Schnittebene $x - 2y \leq \lfloor 2,5 \rfloor = 2$.



Um zu sehen, dass eine solche Schnittebene gültig ist, betrachten wir die zwei Fälle $x \leq \lfloor b \rfloor$ und $x \geq \lfloor b \rfloor + 1$.

1. Fall: $x \leq \lfloor b \rfloor$

Offensichtlich gilt $-\frac{1}{1-f} \cdot y \leq 0$. Addieren wir die beiden Ungleichungen, so erhalten wir die zu zeigende Aussage:

$$x - \frac{1}{1-f} \cdot y \leq \lfloor b \rfloor + 0 = \lfloor b \rfloor$$

Fortsetzung: $x - y \leq b$ mit $x \in \mathbb{N}_0$ und $y \in \mathbb{R}_0^+$ ergibt Schnittebene $x - \frac{1}{1-f} \cdot y \leq \lfloor b \rfloor$.

2. Fall: $x \geq \lfloor b \rfloor + 1 \iff -x \leq -\lfloor b \rfloor - 1$

Multiplizieren wir diese Ungleichung mit $\frac{f}{1-f}$, so erhalten wir:

$$-\frac{f}{1-f} \cdot x \leq \frac{f}{1-f} \cdot (-\lfloor b \rfloor - 1)$$

Außerdem multiplizieren wir die ursprüngliche Randbedingung mit $\frac{1}{1-f}$ und erhalten:

$$\frac{1}{1-f} \cdot (x - y) \leq \frac{1}{1-f} \cdot b$$

Addieren wir beide Ungleichungen, so erhalten wir:

$$-\frac{f}{1-f} \cdot x + \frac{1}{1-f} \cdot (x - y) \leq \frac{f}{1-f} \cdot (-\lfloor b \rfloor - 1) + \frac{1}{1-f} \cdot b$$

Wir erhalten die zu zeigende Aussage, wenn wir $b = f + \lfloor b \rfloor$ einsetzen.

```
#include <glpk.h>
.....
glp_prob *lp = glp_create_prob();
glp_read_mps(lp, GLP_MPS_FILE, 0, "filename");
glp_term_out(GLP_ON);

glp_adv_basis(lp, 0);
int rc = glp_simplex(lp, 0);
if (rc != 0 || glp_get_status(lp) != GLP_OPT) {
   cout << "error" << endl; return 1;
}
cout << "lp: " << glp_get_obj_val(lp) << endl;

rc = glp_intopt(lp, 0);
if (rc == 0)
   cout<< "opt: "<< glp_mip_obj_val(lp)<< endl;
else cout "error" << endl;
glp_delete_prob(lp);
```

MIPLIB 2003	`http://miplib2010.zib.de/miplib2003/index.php`
MIPLIB 2010	`http://miplib2010.zib.de`
MIPLIB 2017	`http://miplib.zib.de`

- electronically available library of both pure and mixed integer programs
- has become a standard test set used to compare the performance of mixed integer optimizers

Übung 8. Starten Sie `glpsol` für verschiedene Instanzen der MIPLIB. Testen Sie verschiedene Heuristiken für Node-Selection und Branching.

Auszug aus `glpsol --help`:

```
--first   branch: first integer variable
--last    branch: last integer variable
--mostf   branch: most fractional variable
--drtom   branch: heuristic by Driebeck and Tomlin(*)
--pcost   branch: hybrid pseudocost heuristic
```

Auszug aus `glpsol --help`: (Fortsetzung)

```
--dfs       backtrack: depth first search
--bfs       backtrack: breadth first search
--bestp     backtrack: best projection heuristic
--bestb     backtrack: best local bound(*)

--gomory    generate Gomory's mixed integer cuts
--mir       generate Mixed Integer Rounding cuts
--cover     generate mixed cover cuts
--clique    generate clique cuts
--cuts      generate all cuts above
```

Branching Rules: Let x_i be a variable with fractional LP value.

- *Most Fractional Branching* choose the variable with fractional part closest to 0.5
  - $score(x_i) := 0.5 - \mid \tilde{x}_i - \lfloor \tilde{x}_i \rfloor - 0.5 \mid$
  - Beispiel: $score(7.9) = 0.1$, $score(3.6) = 0.4$
  - Wähle x_j mit $score(x_j)$ maximal.
- *Least Fractional Branching* choose the variable that is almost an integer
  - $score(x_i) := \mid \tilde{x}_i - \lfloor \tilde{x}_i \rfloor - 0.5 \mid$
  - Beispiel: $score(7.9) = 0.4$, $score(3.6) = 0.1$
  - Wähle x_j mit $score(x_j)$ maximal.
- *Full Strong Branching* test which of the fractional candidates gives the best progress
  - Δ_l change in objective value of left branch on x_i
  - Δ_r change in objective value of right branch on x_i
  - $score(x_i) := (1 - \mu) \cdot \min\{\Delta_l, \Delta_r\} + \mu \cdot \max\{\Delta_l, \Delta_r\}$
- *Pseudo-Cost Branching* und weitere ...

```
#include <iostream>
#include <list>
#include <cmath>
#include <cfloat>
#include <glpk.h>
#include <unistd.h>

using namespace std;

const double _eps = 1e-8;

list<int> getFractionalVariables(glp_prob *p);
int getBranchVariable(glp_prob *p);
double getScore(glp_prob *p, int var);
pair<glp_prob *, glp_prob *> getBranches(glp_prob *job, int i);
```

```
int getBranchVariable(glp_prob *p) {
   double val = 0.0;
   int var = -1;
   list<int>::iterator it;
   list<int> vars = getFractionalVariables(p);

   for (it = vars.begin(); it != vars.end(); it++) {
      double score = getScore(p, *it);

      if (score > val) {
          val = score;
          var = *it;
      }
   }
   return var;
}
```

```
list<int> getFractionalVariables(glp_prob *p) {
   list<int> vars;
   int cols = glp_get_num_cols(p);

   for (int i = 1; i <= cols; i++) {
      int knd = glp_get_col_kind(p, i);

      if (glp_get_col_stat(p, i) == GLP_BS &&
                (knd == GLP_IV || knd == GLP_BV)) {
         double xi = glp_get_col_prim(p, i);
         double rVal = round(xi);
         double diff = fabs(xi - rVal);

         if (diff >= _eps) vars.push_back(i);
      }
   }
   return vars;
}
```

Sei x_i eine ganzzahlige Variable, die nach der LP-Relaxierung einen nicht-ganzzahligen Wert $\hat{x}_i$ hat. Dann fügen wir entweder neue Schranken hinzu oder wir ändern bereits hinzugefügte Schranken in der folgenden Weise:

- Falls x_i nicht begrenzt war, fügen wir $x_i \leq \lfloor \hat{x}_i \rfloor$ bzw. $\lceil \hat{x}_i \rceil \leq x_i$ als neue Schranke hinzu.
- Falls es bereits eine obere Schranke $x_i \leq c_2$ gibt, dann ändern wir diese Schranke in $x_i \leq \lfloor \hat{x}_i \rfloor$ bzw. $\lceil \hat{x}_i \rceil \leq x_i \leq c_2$.
- Falls es eine untere Schranke $c_1 \leq x_i$ gibt, dann ändern wir diese Schranke in $c_1 \leq x_i \leq \lfloor \hat{x}_i \rfloor$ bzw. $\lceil \hat{x}_i \rceil \leq x_i$.
- Falls die Variable bereits nach beiden Seiten $c_1 \leq x_i \leq c_2$ beschränkt ist, ändern wir die Schranke in $c_1 \leq x_i \leq \lfloor \hat{x}_i \rfloor$ bzw. $\lceil \hat{x}_i \rceil \leq x_i \leq c_2$.

In allen obigen Fällen erhalten wir zwei neue Teilprobleme.

```
double getScore(glp_prob *p, int var) {
   // Most Fractional Branch
   double val = glp_get_col_prim(p, var);
   return 0.5 - fabs(val - floor(val) - 0.5);
}

pair<glp_prob *, glp_prob *> getBranches(glp_prob *job, int i){
   double val = glp_get_col_prim(job, i);
   double ub = floor(val);
   double lb = ub + 1.0;

   glp_prob *lp = glp_create_prob();
   glp_copy_prob(lp, job, GLP_OFF);
   glp_prob *rp = glp_create_prob();
   glp_copy_prob(rp, job, GLP_OFF);

   int type = glp_get_col_type(job, i);
   double lower = glp_get_col_lb(job, i);
   double upper = glp_get_col_ub(job, i);
```

```
int lType, rType;

if (type == GLP_DB) {
   lType = GLP_DB;
   rType = GLP_DB;
} else if (type == GLP_LO) {
   lType = GLP_DB;
   rType = GLP_LO;
} else if (type == GLP_UP) {
   lType = GLP_UP;
   rType = GLP_DB;
} else {
   lType = GLP_UP;
   rType = GLP_LO;
}
```

```
    if (lower >= ub) {
        ub = lower;
        lType = GLP_FX;
    }
    glp_set_col_bnds(lp, i, lType, lower, ub);

    if (lb >= upper) {
        lb = upper;
        rType = GLP_FX;
    }
    glp_set_col_bnds(rp, i, rType, lb, upper);

    return pair<glp_prob *, glp_prob *>(lp, rp);
}
```

```
int main(int argc, char *argv[]) {
   if (argc != 2) {
      cout << "usage: " << argv[0]
           << " filename" << endl;
      return 1;
   }

   double opt;
   char *file = argv[1];

   glp_prob *lp = glp_create_prob();
   if (glp_read_mps(lp, GLP_MPS_FILE, 0, file)){
      cout << file << " not found" << endl;
      return 1;
   }

   // disable terminal output
   glp_term_out(GLP_OFF);
```

```
int objDir = glp_get_obj_dir(lp);

if (objDir == GLP_MAX)
   opt = -DBL_MAX;
else opt = DBL_MAX;

glp_adv_basis(lp, 0);
int rc = glp_simplex(lp, 0);  // lp relaxation

if (rc != 0 || glp_get_status(lp) != GLP_OPT){
   cout << "error" << endl;
   return 1;
}
cout << "==> lp-relax: ";
if (objDir == GLP_MIN)
   cout << "lower bound ";
else cout << "upper bound ";
cout << glp_get_obj_val(lp) << " <==" << endl;
```

```
list<glp_prob *> pool;

pool.push_back(lp);
while ( ! pool.empty() ) {
   glp_prob *job = pool.front();
   pool.pop_front();

   glp_adv_basis(job, 0);
   rc = glp_simplex(job, 0);  // lp relaxation

   if (0 != rc || glp_get_status(job) != GLP_OPT) {
      // no valid solution found ==> continue with next job
      glp_delete_prob(job);
      continue;
   }
```

```
double val = glp_get_obj_val(job);
if ((objDir == GLP_MAX && val <= opt)
      || (objDir == GLP_MIN && val >= opt)) {
   // current solution is bad ===> bound
   glp_delete_prob(job);
   continue;
}

int i = getBranchVariable(job);

if (i == -1) {
   if ((objDir == GLP_MAX && val > opt)
         || (objDir == GLP_MIN && val < opt)){
      opt = val;
      cout << "solution: " << val << endl;
   }
   glp_delete_prob(job);
   continue;
}
```

```
        pair<glp_prob *, glp_prob *> probs = getBranches(job, i);

        // Depth First Search
        pool.push_front(probs.first);
        pool.push_front(probs.second);

        glp_delete_prob(job);
    }
    cout << ">>> value: " << opt << " <<<" << endl;

    return 0;
}
```

$y = \neg x$ wird zu $y = 1 - x$

x	$\neg x$	y	$y = \neg x$
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0

$\rightsquigarrow$

x	$1 - x$	y	$y = 1 - x$
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0

$x \Rightarrow y$ wird zu $x \leq y$

x	y	$x \Rightarrow y$
0	0	1
0	1	1
1	0	0
1	1	1

$\rightsquigarrow$

x	y	$x \leq y$
0	0	1
0	1	1
1	0	0
1	1	1

Darstellen boolescher Logik mittels Integer Programmen

$z = x \vee y$ wird zu $0 \leq 2 \cdot z - x - y \leq 1$

x	y	z	$z = x \vee y$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

$\rightsquigarrow$

x	y	z	$0 \leq 2 \cdot z - x - y \leq 1$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

allgemein: $y = x_1 \vee x_2 \vee \ldots \vee x_n$ wird zu $0 \leq n \cdot y - \sum_i x_i \leq n - 1$

$z = x \wedge y$ wird zu $0 \leq x + y - 2 \cdot z \leq 1$

x	y	z	$z = x \wedge y$
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

$\rightsquigarrow$

x	y	z	$0 \leq x + y - 2 \cdot z \leq 1$
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

allgemein: $y = x_1 \wedge x_2 \wedge \ldots \wedge x_n$ wird zu $0 \leq \sum_i x_i - n \cdot y \leq n - 1$

Der Ausdruck

$$(x \wedge y) \Rightarrow z \;\equiv\; \neg(x \wedge y) \vee z \;\equiv\; \neg x \vee \neg y \vee z$$

wird zu $(1 - x) + (1 - y) + z \geq 1$

x	y	z	$(x \wedge y) \Rightarrow z$
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

$\rightsquigarrow$

x	y	z	$(1-x)+(1-y)+z \geq 1$
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

$$\underbrace{\neg A \wedge \neg B \Rightarrow C = D}_{=:F} \rightsquigarrow \underbrace{C - D \leq A + B}_{=:X} \text{ und } \underbrace{-C + D \leq A + B}_{=:Y}$$

A	B	C	D	F	X	Y
0	0	0	0	1	1	1
0	0	0	1	0	1	0
0	0	1	0	0	0	1
0	0	1	1	1	1	1
0	1	0	0	1	1	1
0	1	0	1	1	1	1
0	1	1	0	1	1	1
0	1	1	1	1	1	1
1	0	0	0	1	1	1
1	0	0	1	1	1	1
⋮				⋮		

Use binary decision variables:

- $X_{i,t} = 1$ iff position i is empty after move t
- $Y_{i,j,t} = 1$ iff peg is moved from position i to j at time t

The objective is to maximize the number of empty holes in the final state:

$$\sum_i X_{i,T} \longrightarrow \max$$

T denotes the number of moves which depends on the number of pegs in the initial state and on the number of pegs in the final state, since after each move exactly one peg has been removed from the board.

We have to consider the following constraints:

- One jump per turn:
$$\sum_i \sum_{j \in D_i} Y_{i,j,t} = 1 \quad \forall t$$
- Jump to empty hole: $\bigvee_{j \in D_i} Y_{j,i,t} \Rightarrow X_{i,t}$
$$\sum_{j \in D_i} Y_{j,i,t} \leq X_{i,t} \quad \forall i, t$$
- Jump from occupied hole: $\bigvee_{j \in D_i} Y_{i,j,t} \Rightarrow \neg X_{i,t}$
$$\sum_{j \in D_i} Y_{i,j,t} \leq 1 - X_{i,t} \quad \forall i, t$$

D_i denotes the set of destination positions for a peg in position i.

further constraints:

- Jump over peg: $\bigvee_{(j,k)\in E_i} Y_{j,k,t} \Rightarrow \neg X_{i,t}$

$$\sum_{(j,k)\in E_i} Y_{j,k,t} \leq 1 - X_{i,t} \quad \forall i, t$$

- constraints to describe the initial and final state

E_i denotes the position pairs (j, k) such that a jump from position j over position i to position k is possible.

Further we have to ensure the balance equations:

$$X_{i,t} + \sum_{(j,k)\in E_i} Y_{j,k,t} + \sum_{j\in D_i} Y_{i,j,t} - \sum_{j\in D_i} Y_{j,i,t} = X_{i,t+1} \quad \forall i, t$$

The balance equation ensures:

- $X_{i,t} = X_{i,t+1}$ if position i is not involved in a move
- $X_{i,t} = 1$ and $X_{i,t+1} = 0$ if a move is made to i
- $X_{i,t} = 0$ and $X_{i,t+1} = 1$ if a move is made from i or over i

This IP model comes from:

G.W. DePuy and G.D. Taylor: Using board puzzles to teach operations research. INFORMS Transactions on Education 7(2), 160 – 171 (2007)

1				
	2			

1				
	2			
3				

...

1	24	13	18	7
14	19	8	23	12
9	2	xx	6	17
20	15	4	11	22
3	10	21	16	5

Use binary decision variables:

- $X_{i,t} = 1$ iff square i contains the knight at time t
- $Y_{i,j,t} = 1$ iff at time t the knight is moved from square i to j

The objective is to maximize the number of moves:

$$\sum_i \sum_{j \in D_i} \sum_t Y_{i,j,t} \longrightarrow \max$$

D_i denotes the set of feasible positions that can be reached by the knight from square i.

We have to consider the following constraints:

- the knight stands on one square per turn:

$$\sum_i X_{i,t} = 1 \quad \forall t$$

- the knight visits each square once over time:

$$\sum_t X_{i,t} = 1 \quad \forall i$$

- exactly one move is made per turn:

$$\sum_i \sum_{j \in D_i} Y_{i,j,t} = 1 \quad \forall t$$

- a move from square i requires the knight to be there:

$$\bigvee_{j \in D_i} Y_{i,j,t} \Rightarrow X_{i,t} \qquad \rightsquigarrow \qquad \sum_{j \in D_i} Y_{i,j,t} \leq X_{i,t} \quad \forall i, t$$

Further we have to ensure the balance equation:

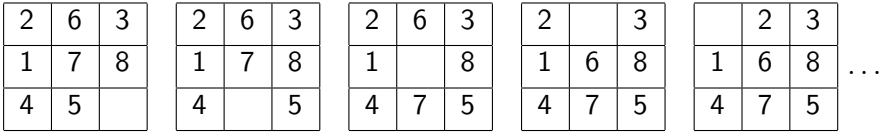
$$X_{i,t} - \sum_{j \in D_i} Y_{i,j,t} + \sum_{j \in D_i} Y_{j,i,t} = X_{i,t+1} \quad \forall i, t$$

The balance equation ensures:

- $X_{i,t} = X_{i,t+1}$ if square i is not involved in a move
- $X_{i,t} = 1$ and $X_{i,t+1} = 0$ if a move is made from i
- $X_{i,t} = 0$ and $X_{i,t+1} = 1$ if a move is made to i

Knight-Tour problem is efficiently solvable, see

I. Parberry: An efficient algorithm for the knight's tour problem. Discrete Applied Mathematics 73(3), 251 – 260 (1997)



Use binary decision variables:

- $X_{i,k,t} = 1$ iff square i contains number k at time t
  number 0 denotes the missing tile
- $Y_{i,j,t} = 1$ iff at time t the numbers of the squares i and j exchange places

The objective is to minimize the number of moves:

$$\sum_i \sum_{j \in D_i} \sum_t Y_{i,j,t} \longrightarrow \min$$

D_i denotes the set of destination positions if the empty tile is in square i. Value T denotes the number of moves which must be chosen large enough.

We have to consider the following constraints:

- exchange at most one pair of squares per turn:

$$\sum_i \sum_{j \in D_i} Y_{i,j,t} \leq 1 \quad \forall t$$

- each square contains exactly one number at each time:

$$\sum_k X_{i,k,t} = 1 \quad \forall i, t$$

- the puzzle contains each number exactly once per time:

$$\sum_i X_{i,k,t} = 1 \quad \forall k, t$$

- if empty tile is moved from square i to j
  - then square i holds 0 at time t:

$$Y_{i,j,t} \Rightarrow X_{i,0,t} \quad \rightsquigarrow \quad Y_{i,j,t} \leq X_{i,0,t} \quad \forall i,j \in D_i, t$$

  - then square j holds 0 at time $t+1$:

$$Y_{i,j,t} \Rightarrow X_{j,0,t+1} \quad \rightsquigarrow \quad Y_{i,j,t} \leq X_{j,0,t+1} \quad \forall i,j \in D_i, t$$

  - and square j holds number k, then square i holds k afterwards:

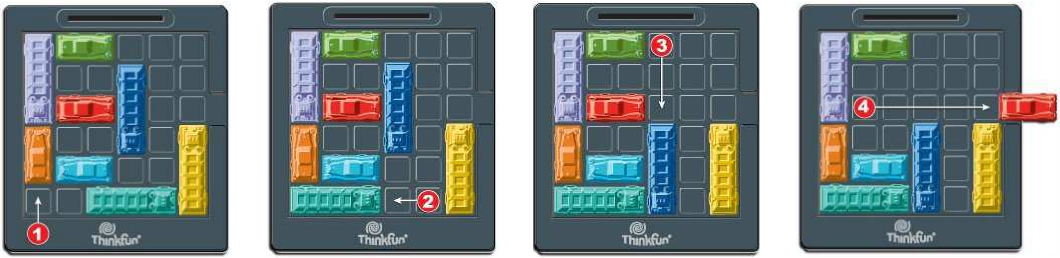
$$Y_{i,j,t} \wedge X_{j,k,t} \Rightarrow X_{i,k,t+1} \quad \rightsquigarrow \quad 1 - Y_{i,j,t} + 1 - X_{j,k,t} + X_{i,k,t+1} \geq 1 \quad \forall i,j \in D_i, t, k > 0$$

- if no exchange between squares i and j has been made, then $X_{i,k,t} = X_{i,k,t+1}$.

$$X_{i,k,t} - X_{i,k,t+1} \leq \sum_{j \in D_i} Y_{i,j,t} + \sum_{j \in D_i} Y_{j,i,t} \quad \forall i,k,t$$

$$-X_{i,k,t} + X_{i,k,t+1} \leq \sum_{j \in D_i} Y_{i,j,t} + \sum_{j \in D_i} Y_{j,i,t} \quad \forall i,k,t$$

Planen mittels Integer Programming: Rush-Hour



Use binary decision variables, where the marker denotes the upper left position of a vehicle:

- $X_{v,j,t} = 1$ iff marker of vehicle v is at position j at time t
- $Z_{v,j,t} = 1$ iff vehicle v occupies position j at time t.
- $Y_{v,i,j,t} = 1$ iff vehicle v moves marker from position i to position j at time t.

The objective is to minimize the number of moves:

$$\sum_{v} \sum_{i \in B_v} \sum_{j \in B_v} \sum_{t} Y_{v,i,j,t} \longrightarrow \min$$

B_v denotes the set of possible positions of marker of vehicle v.

The number of moves T must be chosen large enough to be able to solve the problem.

We have to consider the following constraints:

- at most one vehicle in a position:

$$\sum_{v} Z_{v,i,t} \leq 1 \quad \forall i, t$$

- move at most one vehicle per turn:

$$\sum_{v} \sum_{i \in B_v} \sum_{j \in B_v} Y_{v,i,j,t} \leq 1 \quad \forall t$$

- define positions occupied by vehicles:

$$L_v \cdot X_{v,i,t} \leq \sum_{m \in M_{v,i}} Z_{v,m,t} \quad \forall v, i, t$$

L_v denotes the length of vehicle v, and $M_{v,j}$ denotes the set of positions occupied by vehicle v if its marker is in position j.

Further constraints:

- take note of occupied positions:

$$Y_{v,i,j,t} \leq 1 - \sum_{w \neq v} Z_{w,p,t-1} \quad \forall v, i, j, t, p \in P_{v,i,j}$$

  $P_{v,i,j}$ denotes the set of positions between i and j that vehicle v may reach.

- update marker for vehicles:

$$X_{v,i,t-1} - \sum_{j \in B_v} Y_{v,i,j,t} + \sum_{j \in B_v} Y_{v,j,i,t} = X_{v,i,t} \quad \forall v, i, t$$

Vorgehen bei gegebener Modellierung mittels STRIPS:

- Wird eine Aktion a durchgeführt, dann müssen vorher die PRE-Conditions und hinterher die ADD-Effects gelten, die DEL-Effects dürfen nicht mehr gelten:

$$\begin{array}{lcll} a_t & \rightarrow & p_t & \forall p \in pre(a) \\ a_t & \rightarrow & p_{t+1} & \forall p \in add(a) \\ a_t & \rightarrow & \neg p_{t+1} & \forall p \in del(a) \end{array}$$

- Parallele, widersprüchliche Aktionen werden ausgeschlossen:

$$a_t \rightarrow \neg b_t \quad \forall a, b : del(a) \cap (pre(b) \cup add(b)) \neq \emptyset$$

- Bei einem Wechsel des Zustands muss eine entsprechende Aktion durchgeführt worden sein:

$$\begin{array}{lcl} (\neg p_t \wedge p_{t+1}) & \rightarrow & \bigvee_{p \in add(a)} a_t \\ (p_t \wedge \neg p_{t+1}) & \rightarrow & \bigvee_{p \in del(a)} a_t \end{array}$$

`unstack(X,Y)`:

- PRE: `upon(X,Y)`, `clear(X)`, `empty()`
- ADD: `clear(Y)`, `hold(X)`
- DEL: `upon(X,Y)`, `clear(X)`, `empty()`

Constraints im linearen Programm:

$$
\begin{aligned}
\mathit{unstack}[x,y,t] &\leq \mathit{upon}[x,y,t] \\
\mathit{unstack}[x,y,t] &\leq \mathit{clear}[x,t] \\
\mathit{unstack}[x,y,t] &\leq \mathit{empty}[t] \\
\mathit{unstack}[x,y,t] &\leq \mathit{clear}[y,t+1] \\
\mathit{unstack}[x,y,t] &\leq \mathit{hold}[x,t+1] \\
\mathit{unstack}[x,y,t] &\leq 1 - \mathit{upon}[x,y,t+1] \\
\mathit{unstack}[x,y,t] &\leq 1 - \mathit{clear}[x,t+1] \\
\mathit{unstack}[x,y,t] &\leq 1 - \mathit{empty}[t+1]
\end{aligned}
$$

`stack(X,Y)`:

- PRE: `clear(Y)`, `hold(X)`
- ADD: `upon(X,Y)`, `clear(X)`, `empty()`
- DEL: `clear(Y)`, `hold(X)`

Constraints im linearen Programm:

$$
\begin{aligned}
stack[x,y,t] &\leq clear[y,t] \\
stack[x,y,t] &\leq hold[x,t] \\
stack[x,y,t] &\leq upon[x,y,t+1] \\
stack[x,y,t] &\leq clear[x,t+1] \\
stack[x,y,t] &\leq empty[t+1] \\
stack[x,y,t] &\leq 1 - clear[y,t+1] \\
stack[x,y,t] &\leq 1 - hold[x,t+1]
\end{aligned}
$$

`pickup(X)`:

- PRE: `onTable(X)`, `clear(X)`, `empty()`
- ADD: `hold(X)`
- DEL: `onTable(X)`, `clear(X)`, `empty()`

Constraints im linearen Programm:

$$
\begin{aligned}
pickup[x,t] &\leq onTable[x,t] \\
pickup[x,t] &\leq clear[x,t] \\
pickup[x,t] &\leq empty[t] \\
pickup[x,t] &\leq hold[x,t+1] \\
pickup[x,t] &\leq 1 - onTable[x,t+1] \\
pickup[x,t] &\leq 1 - clear[x,t+1] \\
pickup[x,t] &\leq 1 - empty[t+1]
\end{aligned}
$$

`putdown(X)`:

- PRE: `hold(X)`
- ADD: `onTable(X)`, `clear(X)`, `empty()`
- DEL: `hold(X)`

Constraints im linearen Programm:

$$
\begin{aligned}
\mathit{putdown}[x,t] &\leq \mathit{hold}[x,t] \\
\mathit{putdown}[x,t] &\leq \mathit{onTable}[x,t+1] \\
\mathit{putdown}[x,t] &\leq \mathit{clear}[x,t+1] \\
\mathit{putdown}[x,t] &\leq \mathit{empty}[t+1] \\
\mathit{putdown}[x,t] &\leq 1 - \mathit{hold}[x,t+1]
\end{aligned}
$$

Falls sich `upon(x,y)` geändert hat:

$$\begin{aligned} 1 - upon[x,y,t] + upon[x,y,t+1] &\leq 1 + stack[x,y,t] \\ upon[x,y,t] + 1 - upon[x,y,t+1] &\leq 1 + unstack[x,y,t] \end{aligned}$$

Falls sich `clear(x)` geändert hat:

$$\begin{aligned} & 1 - clear[x,t] + clear[x,t+1] \\ & \quad \leq 1 + \sum_y unstack[y,x,t] + \sum_y stack[x,y,t] + putdown[x,t] \\ & clear[x,t] + 1 - clear[x,t+1] \\ & \quad \leq 1 + \sum_y stack[y,x,t] + \sum_y unstack[x,y,t] + pickup[x,t] \end{aligned}$$

Falls sich `onTable(x)` geändert hat:

$$\begin{aligned} 1 - onTable[x,t] + onTable[x,t+1] &\leq 1 + putdown[x,t] \\ onTable[x,t] + 1 - onTable[x,t+1] &\leq 1 + pickup[x,t] \end{aligned}$$

Falls sich `empty()` geändert hat:

$$1 - empty[t] + empty[t+1] \leq 1 + \sum_x putdown[x,t] + \sum_x \sum_y stack[x,y,t]$$

$$empty[t] + 1 - empty[t+1] \leq 1 + \sum_x pickup[x,t] + \sum_x \sum_y unstack[x,y,t]$$

Falls sich `hold(x)` geändert hat:

$$1 - hold[x,t] + hold[x,t+1] \leq 1 + pickup[x,t] + \sum_y unstack[x,y,t]$$

$$hold[x,t] + 1 - hold[x,t+1] \leq 1 + putdown[x,t] + \sum_y stack[x,y,t]$$

Es darf nur eine Operation zur Zeit t ausgeführt werden:

$$\sum_x \sum_y (stack[x,y,t] + unstack[x,y,t]) + \sum_x (pickup[x,t] + putdown[x,t]) \leq 1$$

Minimiere die Anzahl der ausgeführten Aktionen:

$$\sum_x \sum_y \sum_t (stack[x,y,t] + unstack[x,y,t]) + \sum_x \sum_t (pickup[x,t] + putdown[x,t]) \longrightarrow \min$$

Übung 9. Implementieren Sie das Integer-Programm zur Lösung des Brett-Solitär-Problems mittels GLPK und vergleichen Sie dessen Laufzeit mit der Laufzeit eines Programms in C++ für das Problem.

Übung 10. Erweitern Sie das Integer-Programm zur Lösung des Blocks-World-Problems, so dass mehrere Greifarme gleichzeitig Aktionen durchführen können. Implementieren Sie beide Programme mittels GLPK und vergleichen Sie deren Laufzeiten sowie die berechneten Lösungen.

Wie ist das Integer-Programm zu ändern, wenn der Tisch nur eine begrenzte Anzahl von Blöcken aufnehmen kann?

Übersicht

Übersicht

Semantische Netze[22]

Domänenwissen: Wissen über

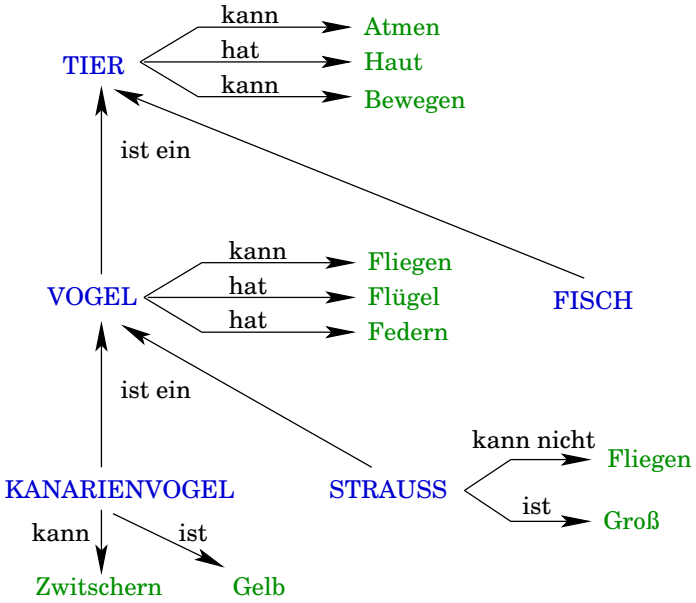
- Objekte und
- ihre Beziehungen.

Semantische Netze:

- Stelle Wissen in einem Graphen dar:
  - Knoten entsprechen Fakten bzw. Konzepten und
  - Kanten entsprechen Relationen bzw. Assoziationen zwischen Konzepten.
- Sowohl Knoten als auch Kanten sind in der Regel mit Beschriftungen versehen.

Beziehungen zwischen Objekten können teilweise automatisch erkannt und für die Verarbeitung genutzt werden.

[22] aus George F. Luger: Künstliche Intelligenz


TIER
kann
Atmen
hat
Haut
kann
Bewegen
ist ein
VOGEL
kann
Fliegen
hat
Flügel
hat
Federn
FISCH
ist ein
KANARIENVOGEL
kann
Zwitschern
ist
Gelb
STRAUSS
kann nicht
Fliegen
ist
Groß

Semantische Netze entsprechen vielleicht der Speicherung von Informationen beim Menschen. (Harmon und King, 1985)

Satz	Antwortzeit (s)
Ein Kanarienvogel ist ein Kanarienvogel.	1.0
Ein Kanarienvogel ist ein Vogel.	1.18
Ein Kanarienvogel ist ein Tier.	1.26
Ein Kanarienvogel kann zwitschern.	1.32
Ein Kanarienvogel kann fliegen.	1.38
Ein Kanarienvogel hat Haut.	1.48

Am schnellsten konnten die Informationen abgerufen werden, die spezifisch für den jeweiligen Vogel sind.

Auch auf spezifischer Ebene: Verarbeitung von Ausnahmen. Auf die Frage, ob Strauße fliegen können, wurde schneller geantwortet als auf die Frage, ob Strauße atmen können.

Relationen, also Kanten im Netz:

- Hierarchische Relationen:
  - Vererbung
  - Beispiel (Instanz)
  - Teile-Ganzes-Beziehung
- Synonymie (Bedeutungsgleichheit von Ausdrücken)
- Antonymie (Bedeutungsgegensatz von Ausdrücken)
- Kausation: ein Ereignis verursacht ein anderes
- Eigenschaft

Aktuelle Systeme:

- Shapiro: Semantic Network Processing System (SNePS)
- Helbig: MultiNet[23], mehrschichtiges erweitertes semantisches Netz zur Repräsentation natürlichsprachlichen Wissens.

[23] `https://de.wikipedia.org/wiki/MultiNet`

Semantic Web: WWW als semantisches Netz

- bisher: WWW verknüpft Daten miteinander → Volltextsuche
- neu: Semantic Web verknüpft Informationen anhand ihrer Bedeutung miteinander → Ontologien

Beispiel:

- Page 1 enthält: Krefeld (Stadt) liegt am Rhein (Fluss).
- Page 2 enthält: Am Rhein (Fluss) ist es schön (Attribut).

⇒ In Krefeld (Stadt) ist es schön (Attribut).

- Dieser Schluss ist möglich, obwohl Informationen in verschiedenen Web-Pages liegen.

Um Krefeld als Ort von Krefeld als Nachname oder Unternehmen zu unterscheiden, müssen Daten mit zusätzlichen Informationen, also Metadaten, versehen werden.

Computer sind mit automatischer Beurteilung oft überfordert.

Beispiel: *Manager* und *Handy* als Suchbegriffe

- Ein Geschäftsführer *ist-ein* *Manager*. (Vererbung)
- Ein Vorstandsvorsitzender *ist-ein* *Manager*. (Vererbung)
- Walter Zabel *ist-ein* Geschäftsführer. (Instanz)
- Ein *Handy* *ist-ein* Mobiltelefon (Synonym)
- Ein *Handy* *ist-ein* mobiles Endgerät (Synonym)

$\rightarrow$ Dokumente, in denen Walter Zabel und Mobiltelefon vorkommen, werden gefunden, auch wenn die Suchbegriffe *Manager* und *Handy* nicht im Text vorkommen.

Idee des Semantic Web:

- Inhalte des Hypertexts mit Metadaten beschreiben
- Ressourcenbeschreibung mittels Resource Description Framework (RDF)
- Modellierungssprache: Ontology Web Language (OWL)
- Ziel: Ausdrücke mit wohldefinierten und somit maschinell interpretierbaren Bedeutungen versehen

$\rightarrow$ inhaltsbezogene Informationssuche

Suchtechnologien im Vergleich:

- Volltextsuche
  - Liefert alle Medieninhalte, die einen oder mehrere der Suchbegriffe beinhalten.
  - Erstellen des Index vollautomatisch → geringe Kosten
  - Darstellen der Ergebnisse in Listenform, schneller Zugriff auf ein Dokument.
- Thesaurus
  - Liefert alle Medieninhalte, die einen oder mehrere der Suchbegriffe oder verwandte, über- oder untergeordnete Begriffe beinhalten.
  - Beispiel: früher `www.dr-antonius.de`, heute `https://www.symptoma.de/`
- Semantische Netze
  - Orientiert sich im Gegensatz zu Thesaurus nicht nur an verwandten Begriffen, sondern an allen Assoziationen.
  - Zusammenhänge werden erfasst.

Anfrage:

- Kinderfreundliches Hotel mit Bademöglichkeit am Strand in Norddeutschland.

Linguistische Analyse:

- Hotel: Nomen
- kinderfreundlich: Adjektiv
- mit Bademöglichkeit am Strand: adverbiale Bestimmung
- in Norddeutschland: adverbiale Bestimmung

Synonymersetzung:

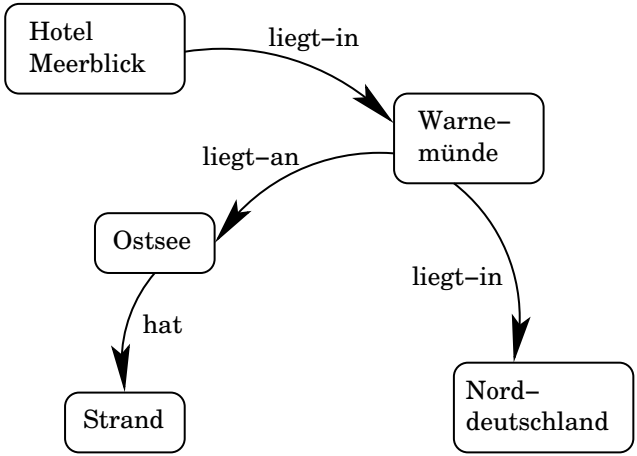
- familienfreundlich entspricht kinderfreundlich

Schlussfolgern:

- Norddeutschland ist-ein Teil von Deutschland
- Deutschland liegt-an Nordsee
- Deutschland liegt-an Ostsee
- Nordsee ist-ein Meer
- Ostsee ist-ein Meer
- Meer hat-ein Strand usw.

Abgleichen:

- Finde Hotels, die der Beschreibung nahe kommen.



→ Ontologie Web Language (OWL)

Ontologie-Schema:

- :Hotel rdf:type rdfs:Class.
  → Hotel ist vom Typ „Klasse“, also ein Konzept mit Instanzen
- :Lokation rdf:type rdfs:Class.
- :liegtIn rdf:type owl:TransitiveProperty
  → „liegtIn“ ist eine Relation (transitiv)
- usw.

Beispiele für Aussagen:

- :HotelMeerblick rdf:type :Hotel.
  → „Hotel Meerblick“ ist eine Instanz der Klasse „Hotel“
- :Warnemünde rdf:type :Lokation.
  → „Warnemünde“ ist eine Instanz von „Lokation“
- :Warnemünde :liegtIn :MecklenburgVorpommern.
- :MecklenburgVorpommern :liegtIn :Norddeutschland.
- :HotelMeerblick :liegtIn :Warnemünde.
- usw.

Abfragesprache SPARQL:

select ?hotel where {?hotel :liegtIn :Warnemünde}

Übersicht

Regeln sind formalisierte Konditionalsätze der Form „Wenn A dann B.“ mit der Bedeutung:

Wenn	A wahr (erfüllt, bewiesen) ist,
dann	schließe, dass auch B wahr ist.

Dabei wird A als *Prämisse* oder *Antezedenz* bezeichnet, B wird *Konklusion* oder *Konsequenz* genannt. Ist die Prämisse einer Regel erfüllt, so sagt man, dass die Regel *feuert*.

Regeln sind eine klassische, weit verbreitete Programmierform für Expertensysteme:

WENN	(Batterie ok)
und	(Wert Tankuhr > 0)
und	(Benzinfilter sauber)
DANN	(Problem = Zündanlage)

Wissensdarstellung durch Regeln

Regeln werden auch gerne in Produktionssystemen zur Steuerung eingesetzt, wobei die Konklusion oft mit einer Aktion verbunden ist:

- *Wenn* der Druck zu hoch ist, *dann* öffne das Ventil.
- *Wenn* die Motortemperatur zu hoch ist, *dann* prüfe die Kühlflüssigkeit.

Solche Regeln werden daher Produktionsregeln genannt.

Modellierung menschlichen Problemlöseverhaltens:

- Erfahrungen basieren auf Problemen, die der Experte früher gelöst hat.
- Diese Erfahrungen hat der Experte in Faustregeln abstrahiert. Wenn ein Experte mit einem neuen Problem konfrontiert wird, wendet er die passende Faustregel an.

Wissensdarstellung durch Regeln

Vorteile der Wissensdarstellung in Form von Regeln:

- Regeln stellen einen guten Kompromiss zwischen Verständlichkeit der Wissensdarstellung und formalen Ansprüchen dar.
- Konditionalsätze werden von Menschen seit langer Zeit benutzt, um Handlungsanweisungen oder Prognosen auszudrücken.
- Regeln sind dem Benutzer daher hinreichend vertraut.
- Ein großer Teil von Expertenwissen lässt sich üblicherweise in Regelform ausdrücken.
- Regeln haben eine große Nähe zum menschlichen Denken.

Es gibt verschiedene Arten von Regeln.

- *Produktionsregeln:* Falls in der aktuellen Datenbasis die Bedingung erfüllt ist, führe die Aktion aus.
- *Logische Regeln:* Falls in der aktuellen Datenbasis die Bedingung wahr ist, dann ist auch die Folgerung wahr.

Unterschied zwischen Regeln und IF-THEN-Anweisung in Programmiersprachen:

- Konventionelle Programme haben einen Kontrollfluss: Eine IF-THEN-Anweisung wird nur ausgeführt, wenn das Programm an die entsprechende Stelle kommt.
- In regelbasierten Systemen überprüft die Inferenzkomponente zu jedem Zeitpunkt alle Regeln, ob ihre Bedingung erfüllt ist. Regeln verhalten sich wie WHENEVER-THEN.

Beispiel für eine Regelbasis:

R1	WENN	(Anlasser arbeitet normal)
	DANN	(Batterie ok)
R2	WENN	(Batterie ok)
	und	(Wert Tankuhr > 0)
	und	(Benzinfilter sauber)
	DANN	(Problem = Zündanlage)
R3	WENN	(Batterie ok)
	und	(Wert Tankuhr > 0)
	und	(NICHT Benzinfilter sauber)
	DANN	(Defekt = Benzinzuleitung)
R4	WENN	(NICHT Scheibenwischer ok)
	und	(NICHT Licht ok)
	DANN	(Defekt = Batterie leer)

Fortsetzung:

R5	WENN	(NICHT Wert Tankuhr > 0)
	DANN	(Defekt = Tank leer)
R6	WENN	(Problem = Zündanlage)
	und	(Verteilerdose ok)
	DANN	(Defekt = Zündspule)

Fakten des aktuellen Falls:

F1 Anlasser arbeitet normal

F2 Scheibenwischer ok

F3 Licht ok

F4 Wert Tankuhr > 0

F5 Benzinfilter sauber

F6 Verteilerdose ok

Abgeleitete Fakten:

A1 Batterie ok (R1 mit F1)

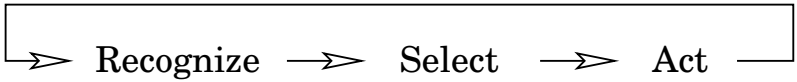
A2 Problem = Zündanlage (R2 mit A1, F4 und F5)

A3 Defekt = Zündspule (R6 mit A2 und F6)

Wenn alle Bedingungen einer Regel durch Fakten der Wissensbasis erfüllt sind, dann führe die Aktion aus.

Recognize-Select-Act Zyklus:

- *Recognize:* Finde alle Regeln, deren Bedingungsteil erfüllt sind.
- *Select:* Wenn mehr als eine Regel anwendbar ist, wird eine Regel ausgewählt (Konfliktlösung).
- *Act:* Führe die Aktion der selektierten Regel aus.



Terminierungsbedingungen: Stop, wenn

- keine Regel mehr anwendbar ist oder
- wenn die Aktion „halt“ ausgeführt wird.

Oft sind wir nicht an allen ableitbaren Fakten interessiert. Daher geht man bei der *Rückwärtsverkettung* von einem Zielobjekt aus, über dessen Zustand der Benutzer Informationen wünscht.

- Das System durchsucht die Regelbasis nach geeigneten Regeln, die das Zielobjekt q in der Konklusion enthalten.
- Die Objekte der Prämisse werden zu Zwischenzielen. Bei einer Regel der Form $p_1 \wedge p_2 \wedge \ldots \wedge p_n \rightarrow q$ werden also die Objekte $p_1, p_2, \ldots, p_n$ zu Zwischenzielen.
- Die Auswertung der Zwischenziele erfolgt rekursiv bis zu den Fakten. $\rightarrow$ Tiefensuche
- Die Regeln werden also vom Folgerungsteil zum Bedingungsteil hin ausgewertet.
- Dies entspricht der Vorgehensweise eines Prolog-Interpreters.

Widersprüchlichkeit

In der Praxis treten häufig Widersprüchlichkeiten in den Wissensbasen auf:

- Experten benutzen zur Ableitung ihrer Schlüsse oft unausgesprochene Annahmen
- oder übersehen, dass Regeln auch Ausnahmen haben können.

Beispiel:

if V then F	Regel	Alle Vögel können fliegen.
if $V \wedge P$ then $\neg F$	Ausnahme	Pinguine können nicht fliegen.

Wenn in diesem Beispiel sowohl V als auch P wahr sind, erhalten wir zwei Schlüsse: F und $\neg F$

Eine Konsistenzprüfung sollte den Benutzer bereits beim Aufbau einer Regelbasis auf solche Unstimmigkeiten hinweisen.

Syntaktischer Aufbau von Regeln

- Man legt i.d.R. Wert auf eine möglichst einfache syntaktische Form der Regeln.
- Dies ermöglicht eine effizientere Abarbeitung der Regeln und
- es verbessert die Übersichtlichkeit der Wirkung einzelner Regeln.

Üblicherweise stellt man folgende Bedingungen an die Form der Regeln (Prämisse $\Rightarrow$ Konklusion):

- $\vee$ (logisches Oder) darf nicht in der Prämisse einer Regel auftreten.
- Die Konklusion einer Regel sollte nur aus einem Literal bestehen.

Regeln, die diesen Bedingungen nicht genügen, müssen nach den Regeln der klassischen Logik umgeformt werden.

Wenn wir Regeln der Form

$$A \vee B \to C \qquad A \to B \wedge C \qquad A \to B \vee C$$

verbieten, dann müssen wir angeben, wie wir solche Regeln umformen:

$$\begin{array}{lcll} A \vee B \to C & \iff & \neg(A \vee B) \vee C & \text{Def. der Implikation} \\ & \iff & (\neg A \wedge \neg B) \vee C & \text{de Morgan} \\ & \iff & (\neg A \vee C) \wedge (\neg B \vee C) & \text{distributiv} \\ & \iff & (A \to C) \wedge (B \to C) & \text{Def. der Implikation} \end{array}$$

Ebenso kann man herleiten:

$$\begin{array}{lcll} A \to B \wedge C & \iff & \neg A \vee (B \wedge C) & \text{Def. der Implikation} \\ & \iff & (\neg A \vee B) \wedge (\neg A \vee C) & \text{distributiv} \\ & \iff & (A \to B) \wedge (A \to C) & \text{Def. der Implikation} \end{array}$$

Nun müssen wir noch herleiten:

$$\begin{array}{lll} A \to B \vee C & \iff & \neg A \vee (B \vee C) \quad \text{Def. der Implikation} \\ & \iff & (\neg A \vee B) \vee C \quad \text{assoziativ} \\ & \iff & \neg(A \wedge \neg B) \vee C \quad \text{de Morgan} \\ & \iff & A \wedge \neg B \to C \quad \text{Def. der Implikation} \end{array}$$

Fassen wir oben bei der Anwendung des Assoziativgesetzes anders zusammen, nämlich $(\neg A \vee C) \vee B$, dann erhalten wir:

$$A \to B \vee C \iff A \wedge \neg C \to B$$

Fassen wir nun obige Umformungen zusammen und wenden sie auf ein Beispiel an.

Aus der Regel

$$A \vee B \rightarrow C \vee D$$

erhalten wir dann die vier Regeln:

$$A \wedge \neg C \rightarrow D \qquad A \wedge \neg D \rightarrow C \qquad B \wedge \neg C \rightarrow D \qquad B \wedge \neg D \rightarrow C$$

Das ist auch intuitiv klar:

- Die linke Seite ist erfüllt, wenn A oder B wahr ist.
- In diesem Fall muss auch die rechte Seite wahr sein.
- Ist ein Teil der rechten Seite nicht wahr, dann muss der andere rechte Teil erfüllt sein.

Schauen wir uns das an einem konkreten Beispiel aus dem Alltag an.

Beispiel: Wenn es morgen regnet oder schneit, gehen wir ins Kino oder bleiben zu Hause.

Diese Regel lässt sich äquivalent durch die folgenden vier Regeln ausdrücken:

- Wenn es morgen regnet und wir nicht ins Kino gehen, dann bleiben wir zu Hause.
- Wenn es morgen regnet und wir nicht zu Hause bleiben, dann gehen wir ins Kino.
- Wenn es morgen schneit und wir nicht ins Kino gehen, dann bleiben wir zu Hause.
- Wenn es morgen schneit und wir nicht zu Hause bleiben, dann gehen wir ins Kino.

Nun erfüllt jede Regel die gestellten Anforderungen. Die Zahl der Regeln hat sich allerdings erhöht. Der Sinn dieser Umformungen wird im Kapitel Prädikatenlogik erklärt.

Regelbasierte Programmierung: Diskussion

Deklarative Programmierung:

- Direkte Repräsentation von Expertenwissen.

Kein expliziter Kontrollfluss:

- Reihenfolge der Regeln spielt keine Rolle.
- Regeln sind aktiv sobald ihre Bedingung erfüllt ist.
- Konfliktlösung wählt eine der aktiven Regeln aus.

Problem:

- Wartung großer Regelsysteme ist aufwändig.
- Auswirkungen von Regeländerungen sind nicht direkt erkennbar.

Regeln sind nicht für alle Arten von Wissen geeignet.

Übersicht

Unsicheres Wissen

Leider sind Fakten und Regeln nicht immer so eindeutig, wie wir sie bisher dargestellt haben:

- unsichere Fakten:
  - Der Temperatursensor hat eine Toleranz von $\pm 1°C$.
  - Morgen wird es regnen.
- unscharfe Fakten:
  - Ab welcher Temperatur hat man hohes Fieber?
  - Ab wann ist der Druck im Kessel zu hoch?
- unvollständige Fakten: Fehlende Informationen werden durch Annahmen ersetzt.
- unsichere Regeln: „Wenn der Patient Fieber hat, dann hat er sehr wahrscheinlich eine bakterielle Infektion.“

Alle Schlussfolgerungen, die auf solchen Fakten oder Regeln basieren, sind unsicher.

Die klassische zweiwertige Logik mit den Wahrheitswerten 0 und 1 kann hier nicht angewendet werden.

Stattdessen drückt man durch Wahrscheinlichkeiten aus, mit welchem Maß man davon ausgehen kann, dass ein Ereignis eintritt.

Wahrscheinlichkeitsfunktion: Sei Ω ein endlicher Ereignisraum. Eine Funktion $p : 2^{\Omega} \to [0, 1]$ heißt Wahrscheinlichkeitsfunktion (oder Wahrscheinlichkeit), wenn gilt:

- $p(\Omega) = 1$
- $p(X \cup Y) = p(X) + p(Y)$ für $X, Y \subseteq \Omega$ und $X \cap Y = \emptyset$

[24] aus U. Lämmel, J. Cleve: Lehr- und Übungsbuch Künstliche Intelligenz.

Für solche Wahrscheinlichkeitsfunktionen gilt mit $X, Y \subseteq \Omega$:

- *unmögliches Ereignis:* $p(\emptyset) = 0$
- *Monotonie:* $X \subseteq Y \Rightarrow p(X) \leq p(Y)$
- *Subtraktivität:* für $X \subseteq Y$ gilt: $p(Y - X) = p(Y \cap X^c) = p(Y) - p(X)$

  denn $X \subseteq Y \Rightarrow Y = X \cup (Y \cap X^c)$, wobei X und $Y \cap X^c$ disjunkt sind, also gilt: $P(Y) = P(X) + P(Y \cap X^c) \iff P(Y \cap X^c) = P(Y) - P(X)$
- *Subadditivität:* $p(X \cup Y) \leq p(X) + p(Y)$
- *Additivität:* $p(X \cup Y) = p(X) + p(Y) - p(X \cap Y)$
- *Komplement:* $p(X^c) = 1 - p(X)$

Bedingte Wahrscheinlichkeit von X unter der Bedingung B:

$$p(X \mid B) = \frac{p(X \cap B)}{p(B)}$$

Beispiel: Welchen Wahrscheinlichkeitswert muss die Aussage „Wenn eine gerade Zahl gewürfelt wurde, dann ist es eine 2.“ bekommen?

- $X = \{2\}$: Es wurde eine 2 gewürfelt.
- $B = \{2, 4, 6\}$: Es wurde eine gerade Zahl gewürfelt.
- $p(X) = p(\{2\}) = \frac{1}{6}$
- $p(B) = p(\{2, 4, 6\}) = \frac{3}{6} = 0,5$
- $p(X \mid B) = p(\{2\} \mid \{2, 4, 6\}) = \frac{p(\{2\})}{p(\{2,4,6\})} = \frac{1}{3}$

Bayessche Formel:

$$p(X \mid B) = \frac{p(X \cap B)}{p(B)} = \frac{\frac{p(B \cap X)}{p(X)} \cdot p(X)}{p(B)} = \frac{p(B \mid X) \cdot p(X)}{p(B)}$$

Wenn die Berechnung von $p(X \mid B)$ verlangt ist, und $p(B \mid X)$ bekannt ist, dann kann die Schlussfolgerung „umgedreht“ werden.

Beispiel:

- X: Der Motor macht seltsame Geräusche. $p(X) = 0.08$
- Y: Der Motor braucht einen Ölwechsel. $p(Y) = 0.06$
- Ein dringend erforderlicher Ölwechsel sorgt in 60% der Fälle für seltsame Motorengeräusche. $p(X|Y) = 0.6$
- Mit welcher W.-keit deutet ein seltsames Motorgeräusch auf einen dringend erforderlichen Ölwechsel hin?

$$p(Y \mid X) = \frac{p(X \mid Y) \cdot p(Y)}{p(X)} = \frac{0.6 \cdot 0.06}{0.08} = 0.45$$

Anmerkung: In praktischen Anwendungen werden oft die Wahrscheinlichkeitstheoretischen Voraussetzungen nicht erfüllt.

- Analytische Wahrscheinlichkeiten oder auf gesicherten statistischen Erhebungen basierende Wahrscheinlichkeiten sind problemlos.
- Rein subjektive Einschätzungen von Experten führen schnell zu Inkonsistenzen.

  Wären im obigen KFZ-Beispiel die Wahrscheinlichkeiten subjektiv ermittelt, und nimmt man $p(X) = 0.03$ an, dann ergäbe sich $p(Y|X) = 1.2$ als Wert der bedingten Wahrscheinlichkeit.

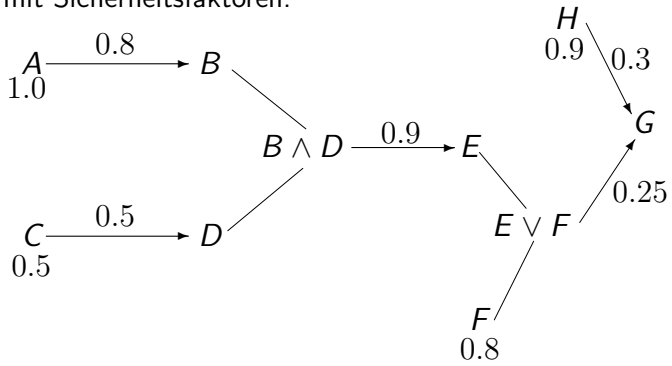
Bei subjektiven Einschätzungen ist der Begriff *Glaube* eher angebracht als der Begriff Wahrscheinlichkeit.

Bereits bei einem der ersten wissensbasierten Systeme überhaupt, dem medizinischen Diagnosesystem MYCIN, musste das Problem der Repräsentation und der Verarbeitung unsicheren Wissens gelöst werden.

- Regeln wurden mit einem Sicherheitsfaktor (certainty factor) versehen, einer reellen Zahl zwischen -1 und $+1$.

  Sicherheitsfaktoren drücken den Grad des Glaubens an eine Hypothese aus. $CF(A \to B)$ gibt an, wie sicher die Konklusion B ist, wenn die Prämisse A wahr ist.
  - $CF(A \to B) = \pm 1$: B ist definitiv wahr/falsch
  - $CF(A \to B) = 0$: indifferente Haltung
- Auch die Sicherheit eines Fakts F wird mit einem solchen Sicherheitsfaktor $CF(F)$ quantifiziert.
- $CF(B \mid A)$ beschreibt den Modus Ponens: Aus ($A \to B$ und A) folgt B, aber mit welcher Sicherheit gilt B?

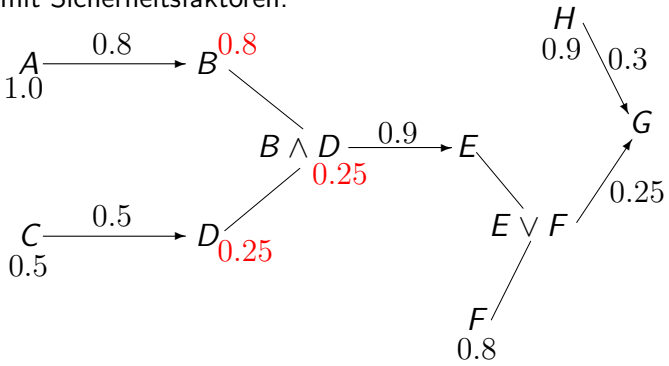
Inferenznetzwerk mit Sicherheitsfaktoren:



In MYCIN wurden folgende Propagationsregeln implementiert:

- Konjunktion: $CF(A \wedge B) = \min\{CF(A), CF(B)\}$
- Disjunktion: $CF(A \vee B) = \max\{CF(A), CF(B)\}$
- Kombination: $CF(B \mid A) = CF(A \rightarrow B) \cdot \max\{0, CF(A)\}$

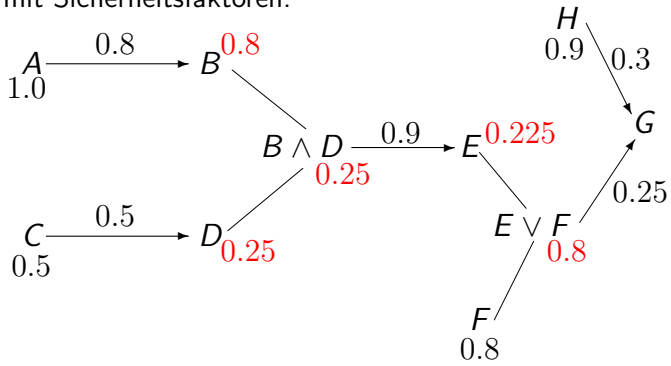
Inferenznetzwerk mit Sicherheitsfaktoren:



Beispiele:

- $CF(B \mid A) = CF(A \to B) \cdot CF(A) = 0.8 \cdot 1.0 = 0.8$
- $CF(D \mid C) = CF(C \to D) \cdot CF(C) = 0.5 \cdot 0.5 = 0.25$
- $CF(B \wedge D) = \min\{CF(B), CF(D)\} = 0.25$

Inferenznetzwerk mit Sicherheitsfaktoren:



Für G liegen zwei bestärkende Aussagen vor:

- $CF(G \mid H) = CF(H \rightarrow G) \cdot CH(H) = 0.3 \cdot 0.9 = 0.27$
- $CF(G \mid E \vee F) = CF(E \vee F \rightarrow G) \cdot CH(E \vee F) = 0.25 \cdot 0.8 = 0.2$

$\Rightarrow$ $CF(G \mid H, E \vee F)$ ist größer als $CF(G \mid H)$ und $CF(G \mid E \vee F)$.

parallele Kombination:

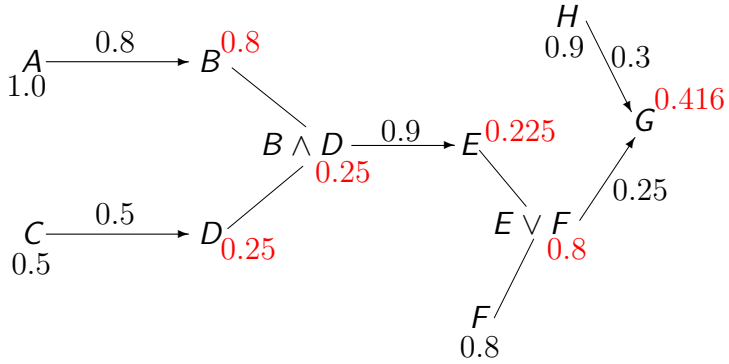
$$CF(B \mid A_1, A_2, \ldots, A_n) \;=\; f(CF(B \mid A_1, A_2, \ldots, A_{n-1}),\; CF(B \mid A_n))$$

Dabei ist $f : [0,1] \times [0,1] \to [0,1]$ definiert als

$$f(x,y) := \begin{cases} x + y - xy & \text{wenn } x, y > 0 \\ x + y + xy & \text{wenn } x, y < 0 \\ \textit{undefiniert} & \text{wenn } x \cdot y = -1 \\ \frac{x+y}{1-\min\{|x|,|y|\}} & \text{sonst} \end{cases}$$

In unserem Beispiel:

$$\begin{aligned} CF(G \mid H, E \vee F) &= f(CF(G \mid H), CF(G \mid E \vee F)) \\ &= CF(G \mid H) + CF(G \mid E \vee F) - CF(G \mid H) \cdot CF(G \mid E \vee F) \\ &= 0.27 + 0.2 - 0.054 = 0.416 \end{aligned}$$

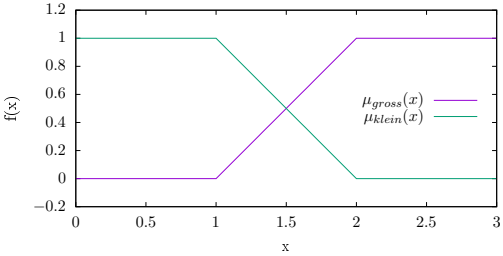


Die Verwendung von Sicherheitsfaktoren ist nur eine heuristische Methode zur Repräsentation und Verarbeitung quantifizierter Unsicherheiten, die sich theoretisch nicht fundieren ließ. Dennoch erwies sie sich als recht leistungsstark und erfolgreich.

Übersicht

Wie können unscharfe Aussagen wie groß/klein, jung/alt oder kalt/heiß modelliert werden?

Charakteristische Funktion: Beschreibt die Zugehörigkeit zu einer Klasse und lässt Werte zwischen 0 und 1 zu.



- Eine 2,10 m große Person, wird von allen als groß angesehen.
- Ein Säugling von 80 cm wird von allen als klein angesehen.

- Ein Volleyballspieler von 2,10 m wird eine 1,80 m große Person nicht als groß bezeichnen, für Kinder wird eine 1,80 m große Person sehr wohl groß sein.

Eine *Fuzzy-Menge* Z über einer Referenzmenge X ist eine Teilmenge aus $X \times [0,1]$. Dabei wird Z durch die Zugehörigkeitsfunktion $\mu_Z : X \to [0,1]$ beschrieben, wobei gilt:

$$z = (x,s) \in Z \iff \mu_Z(x) = s$$

Das Fuzzy-Konzept ist nur eine Verallgemeinerung der üblichen Mengenlehre, bei der wir auch Werte zwischen 0 und 1 erlauben. Klassisch:

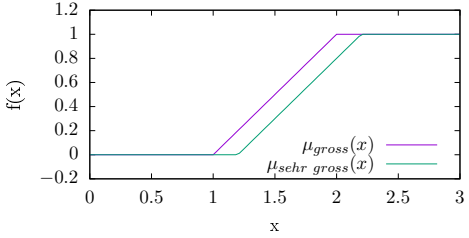
$$\mu_A : X \to \{0,1\} \qquad \text{mit} \qquad \mu_A(x) = \begin{cases} 1, & \text{falls } x \in A \\ 0, & \text{sonst} \end{cases}$$

Um die Zugehörigkeit von x zur Fuzzy-Menge $M_{gross} \cap M_{klein}$ zur berechnen, müssen die Zugehörigkeitswerte von $\mu_{gross}(x)$ und $\mu_{klein}(x)$ geeignet verknüpft werden.

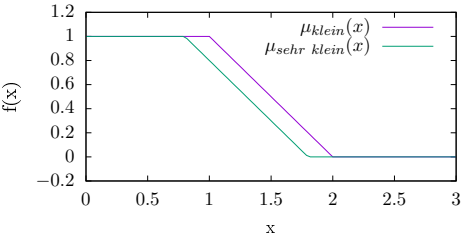
Teilmengenbeziehung: Seien Z_1 und Z_2 zwei Fuzzy-Mengen über X. Z_1 heißt Teilmenge von Z_2, geschrieben $Z_1 \subset Z_2$, genau dann wenn für alle $x \in X$ gilt:

$$\mu_{Z_1}(x) \leq \mu_{Z_2}(x)$$

Beispiel: Die Menge $M_{sehr\ gross}$ der sehr großen Menschen ist eine Teilmenge der Menge M_{gross} der großen Menschen.



$M_{sehr\ gross} \subseteq M_{gross}$



$M_{sehr\ klein} \subseteq M_{klein}$

Durchschnitt zweier Fuzzy-Mengen:

Sei $S : [0,1] \times [0,1] \to [0,1]$ eine Funktion mit den Eigenschaften:

- neutrales Element: $S(a,1) = a$ für $a \in [0,1]$.

  Der Zugehörigkeitswert 1 steht dafür, dass das Element mit Sicherheit zur Fuzzy-Menge Z_2 gehört. Dann ist es plausibel, dass der Zugehörigkeitswert des Durchschnitts dem Zugehörigkeitswert zur Menge Z_1 entspricht.
- Monotonie: $a \leq b \Rightarrow S(a,c) \leq S(b,c)$ für $a, b, c \in [0,1]$.
- Kommutativität: $S(a,b) = S(b,a)$ für $a, b \in [0,1]$.
- Assoziativität: $S(a, S(b,c)) = S(S(a,b), c)$ für $a, b, c \in [0,1]$.

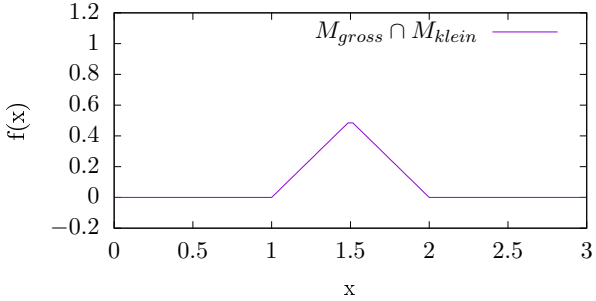
Eine solche Funktion S nennt man *t-Norm*.

Der Durchschnitt zweier Fuzzy-Mengen Z_1 und Z_2 wird definiert durch die Zugehörigkeitsfunktion $\mu_{Z_1 \cap Z_2} = S(\mu_{Z_1}(x), \mu_{Z_2}(x))$.

Mögliche Funktionen:

- $S(a, b) = \min(a, b)$
- $S(a, b) = a \cdot b$

Wählt man das Minimum, erhält man für den Schnitt von M_{gross} und M_{klein}:



Vereinigung zweier Fuzzy-Mengen:

Sei $V : [0,1] \times [0,1] \to [0,1]$ eine Funktion mit den Eigenschaften:

- neutrales Element: $V(a,0) = a$ für $a \in [0,1]$.

  Der Zugehörigkeitswert 0 steht dafür, dass das Element mit Sicherheit nicht zur Fuzzy-Menge Z_2 gehört. Dann ist es plausibel, dass der Zugehörigkeitswert der Vereinigung dem Zugehörigkeitswert zur Menge Z_1 entspricht.
- Monotonie: $a \leq b \Rightarrow V(a,c) \leq V(b,c)$ für $a, b, c \in [0,1]$.
- Kommutativität: $V(a,b) = V(b,a)$ für $a, b \in [0,1]$.
- Assoziativität: $V(a, V(b,c)) = V(V(a,b), c)$ für $a, b, c \in [0,1]$.

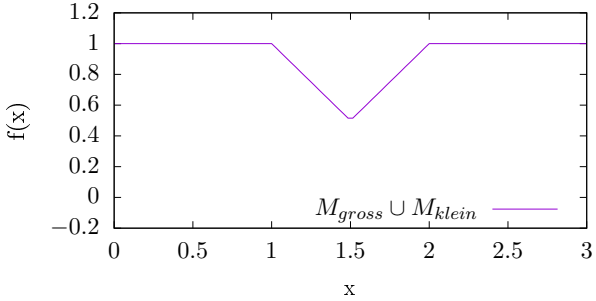
Eine solche Funktion V nennt man *t-CoNorm*.

Die Vereinigung zweier Fuzzy-Mengen Z_1 und Z_2 wird definiert durch die Zugehörigkeitsfunktion $\mu_{Z_1 \cup Z_2} = V(\mu_{Z_1}(x), \mu_{Z_2}(x))$.

Mögliche Funktionen:

- $V(a, b) = \max(a, b)$
- $V(a, b) = a + b - a \cdot b$

Wählt man das Maximum, erhält man für die Vereinigung von M_{gross} und M_{klein}:



Die gewählte t-Norm und die t-CoNorm müssen zueinander passen, um die in der klassischen Logik geltenden Aussagen auch bei der Fuzzy-Logik zu erfüllen.

Beispiel: Minimum als t-Norm, Maximum als t-CoNorm. Dann gilt

$$A \vee (A \wedge B) = A$$

auch in der Fuzzy-Logik, denn es gilt:

$$\mu_{A\vee(A\wedge B)}(x) = \max\left\{\mu_A(x), \min\{\mu_A(x), \mu_B(x)\}\right\} = \mu_A(x)$$

Wie wir aus der Aussagenlogik wissen, sind die Operatoren $\vee$, $\wedge$ und $\neg$ eine Basis. Daher können wir die Implikation $A \to B$ auch darstellen als $\neg A \vee B$. Die so definierte Zugehörigkeitsfunktion

$$\mu_{A \to B}(x) := \max\{1 - \mu_A(x), \mu_B(x)\}$$

geht auf Kleene und Dienes zurück. Es gibt weitere Definitionen für die Implikation:

Name	$\mu_{A \to B}(x)$
Lukasiewicz	$\min\{1,\ 1 - \mu_A(x) + \mu_B(x)\}$
Kleene-Dienes	$\max\{1 - \mu_A(x),\ \mu_B(x)\}$
Zadeh	$\max\left\{1 - \mu_A(x),\ \min\{\mu_A(x),\ \mu_B(x)\}\right\}$
Reichenbach	$1 - \mu_A(x) + \mu_A(x) \cdot \mu_B(x)$

Lotfi A. Zadeh gilt als Erfinder der Fuzzy-Logik (1965), University of California, Berkeley.

Machen wir uns zunächst klar, dass die Implikation $A \to B$ in der klassischen Aussagenlogik auf zwei verschiedene Arten vorkommt:

- Sind die Wahrheitswerte von A und B bekannt, dann kann auf den Wahrheitswert von $A \to B$ geschlossen werden.
- Modus Ponens: Wenn bekannt ist, dass die Regel $A \to B$ gilt, und wir den Wahrheitswert von A kennen, dann können wir auf den Wahrheitswert von B schließen.

In der Fuzzy-Logik ist es genauso: Sind die Zugehörigkeitswerte $\mu_A(x)$ und $\mu_B(x)$ von x bekannt, dann können wir nach einer der obigen Formeln den Zugehörigkeitswert $\mu_{A \to B}(x)$ von x berechnen.

Beim Schließen mittels Modus Ponens setzen wir die Gültigkeit der Regel voraus, und ermitteln aus dem Zugehörigkeitswert $\mu_A(x)$ den Zugehörigkeitswert $\mu_B(x)$.

Beispiel: Es gelte die Regel „große Menschen sind schwer“, und außerdem gelte „Hans ist groß“. Klassisch:

große Menschen sind schwer	vorhandenes	$A \to B$
Hans ist groß	Wissen	A
Hans ist schwer	Schlussfolgerung	B

In der Fuzzy-Logik sind unscharfe Werte gegeben. Wenn Hans $1,80$ Meter groß ist, dann lesen wir den Zugehörigkeitswert $\mu_{gross}(1,8) = 0,8$ aus unserer Grafik ab: Hans ist ziemlich groß.

große Menschen sind schwer	vorhandenes
Hans ist *ziemlich* groß	Wissen
Hans ist *ziemlich* schwer	Schlussfolgerung

Wir übertragen die Werte der Fakten auf die Werte der Folgerung.

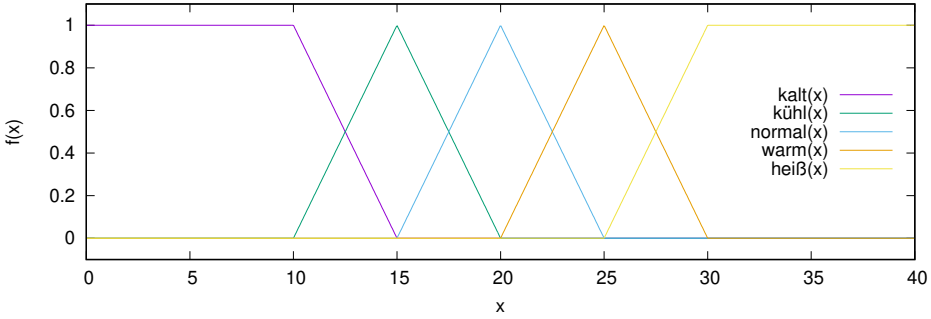
Werte einer *Linguistischen Variablen* sind Worte einer natürlichen Sprache. Bei der Temperatur unterscheiden wir oft die linguistischen Werte *kalt*, *kühl*, *normal*, *warm* und *heiß*.

Diese Werte werden durch unscharfe Mengen A_i bzw. deren Zugehörigkeitsfunktionen $\mu_{A_i}(x)$ repräsentiert.

Die Zugehörigkeitsfunktionen bilden eine linguistische auf eine numerische Werteskala ab.

In der Praxis hat es sich als zweckmäßig erwiesen, die Zugehörigkeitsfunktionen durch stückweisen linearen Verlauf zu definieren. Es sind auch andere Verläufe möglich, wichtig ist nur, dass das Problem adäquat modelliert wird.

Wir weisen den Werten der linguistischen Variablen *kalt*, *kühl*, *normal*, *warm* und *heiß* Fuzzy-Mengen zu:



Wollen wir nun eine Heizungssteuerung implementieren, benötigen wir einige Regeln der Form:

(1) *Wenn* $T_{innen} =$ kühl *Und* $T_{aussen} =$ kalt *Dann* Heizung = stark
(2) *Wenn* $T_{innen} =$ kühl *Und* $T_{aussen} =$ kühl *Dann* Heizung = normal
(3) *Wenn* $T_{innen} =$ normal *Und* $T_{aussen} =$ kalt *Dann* Heizung = normal
(4) *Wenn* $T_{innen} =$ normal *Und* $T_{aussen} =$ kühl *Dann* Heizung = schwach

Ebenso wie für die Temperatur legen wir linguistische Werte *aus*, *schwach*, *normal*, *stark* und *voll* für die Leistung der Heizung fest.

Zu einer vollständigen Regelung benötigen wir noch weitere Regeln, aber die angegebenen reichen für unser Beispiel.

Wir gehen davon aus, dass wir innen und außen folgende Temperaturen gemessen haben.

$$T_{innen} = 17 \qquad \text{und} \qquad T_{aussen} = 11$$

Diese Werte sehen wir als scharfe Werte an, so wie die Größe von Hans im vorigen Beispiel.

Wir bestimmen nun die Zugehörigkeitswerte zu den einzelnen Fuzzy-Mengen:

T	$\mu_{\text{kalt}}(T)$	$\mu_{\text{kühl}}(T)$	$\mu_{\text{normal}}(T)$	$\mu_{\text{warm}}(T)$	$\mu_{\text{heiß}}(T)$
innen $= 17$	0	0,6	0,4	0	0
außen $= 11$	0,8	0,2	0	0	0

Da wir Regeln der Form $A \wedge B \rightarrow C$ haben, müssen wir zunächst den Zugehörigkeitswert der linken Seite bestimmen. Dazu nutzen wir die Formel $\mu_{A \wedge B}(x) = \min\{\mu_A(x), \mu_B(x)\}$.

Wir haben bereits gesehen, dass wir den Zugehörigkeitsgrad der Prämisse für die Schlussfolgerung übernehmen. Welche Werte der linguistischen Variablen *Heizung* in der Tabelle stehen, wird durch die Regeln festgelegt:

T_{innen}			$\wedge$	T_{aussen}			$\Rightarrow$	Heizung			Regel
kühl	=	0,6	$\wedge$	kalt	=	0,8	$\Rightarrow$	stark	=	0,6	(1)
kühl	=	0,6	$\wedge$	kühl	=	0,2	$\Rightarrow$	normal	=	0,2	(2)
normal	=	0,4	$\wedge$	kalt	=	0,8	$\Rightarrow$	normal	=	0,4	(3)
normal	=	0,4	$\wedge$	kühl	=	0,2	$\Rightarrow$	schwach	=	0,2	(4)

Probleme:

- Zwei Regeln führen zur gleichen Aussage mit verschiedenen Zugehörigkeitsgraden.
- Die Heizung soll gleichzeitig stark, normal und schwach heizen.

Wir benötigen eine Verknüpfungsstrategie für die unscharfen Mengen, denen die Regeln in der Konklusion unterschiedliche Zugehörigkeitsgrade zuweisen. Es wurden verschiedene Strategien untersucht:

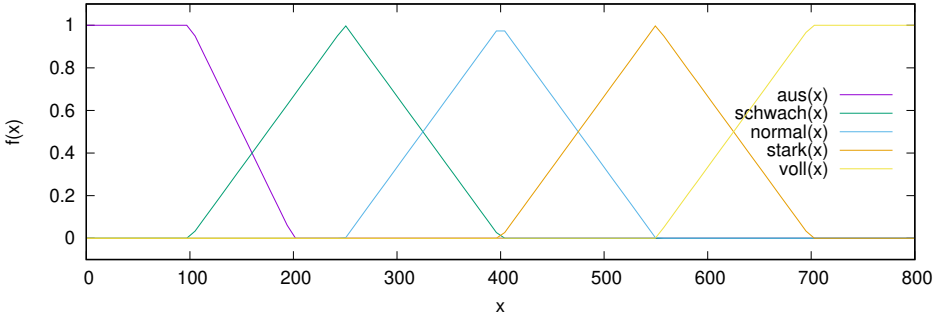
- Wähle das Maximum der Zugehörigkeitsgrade.
- Wähle den arithmetischen Mittelwert.
- Bilde die Summe der unscharfen Mengen: $\mu_{A+B} = \mu_A + \mu_B - \mu_A \cdot \mu_B$

Welche Verknüpfungsstrategie gewählt wird hängt davon ab, welche die Prozesscharacteristik am besten beschreibt.

Wir wählen das Maximum der Werte und erhalten so:

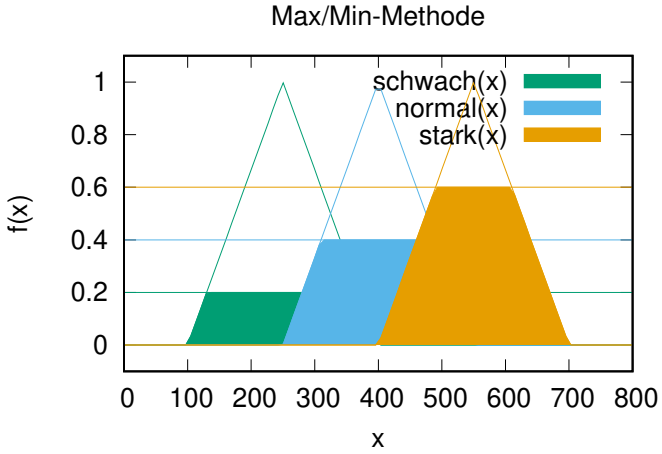
$$\mu_{\text{stark}} = 0,6 \qquad \mu_{\text{normal}} = 0,4 \qquad \mu_{\text{schwach}} = 0,2$$

Wenn wir nun die Heizung regeln wollen, müssen wir die Zugehörigkeitswerte de-fuzzyfizieren:

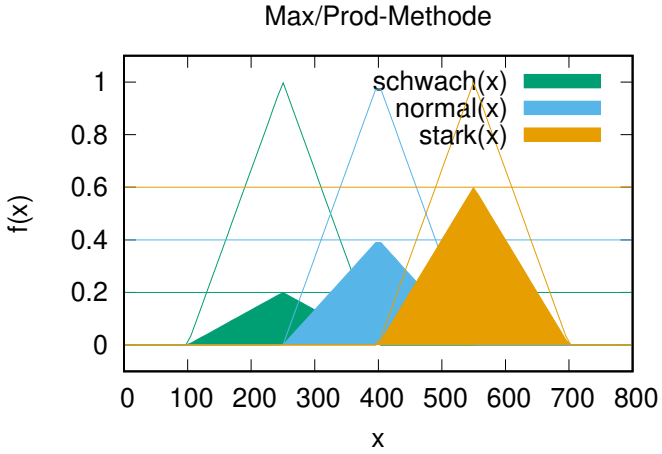


Für das De-Fuzzifizieren wurden ebenfalls mehrere Möglichkeiten untersucht. Betrachten wir zunächst die Max/Min-Methode.

Bei der Max/Min-Methode werden die Zugehörigkeitsfunktionen der einzelnen unscharfen Mengen in Höhe des jeweils ermittelten Zugehörigkeitsgrades abgeschnitten.



Bei der Max/Prod-Methode werden die Werte mit dem Zugehörigkeitsgrad multipliziert. Das bewirkt eine Stauchung der Zugehörigkeitsfunktionen.



Schließlich muss aus diesen Flächen ein einziger Wert ermittelt werden, der dann zur Steuerung genutzt werden kann. Oft wird dafür der Flächenschwerpunkt ermittelt:

$$x_s = \frac{\int_{x_A}^{x_E} x \cdot f(x) dx}{\int_{x_A}^{x_E} f(x) dx}$$

Dabei bedeutet:

- x_s ist die x-Koordinate des Flächenschwerpunktes
- x_A ist die x-Koordinate des Anfangswertes der Fläche
- x_E ist die x-Koordinate des Endwertes der Fläche
- $f(x)$ ist die Funktion, die die Berandung des Flächenstücks beschreibt

Der so ermittelte Wert x_s wird als Steuergröße für den Regelprozess verwendet.

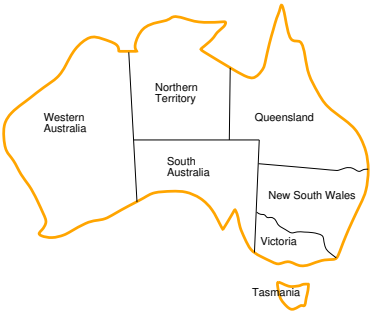
Übersicht

Übersicht

Motivation

- Sehr viele Probleme der Wissensverarbeitung lassen sich auf ein Suchproblem zurückführen.
- Die Eigenschaften und Lösungsverfahren von Suchproblemen sind daher von grundlegender Bedeutung für die Wissensverarbeitung.
- Suchverfahren sind ein klassisches Kapitel innerhalb der Wissensverarbeitung.
- Wir werden die Zustandsraumsuche im wesentlichen bei Spielen kennen lernen.

Färbeproblem



Beispiel: Die angegebene Landkarte ist so mit den Farben rot, blau, gelb und grün zu färben, dass keine zwei benachbarten Länder die gleiche Farbe haben.

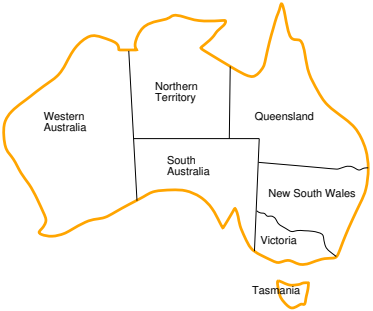
- Ein naives generate-and-test Verfahren würde 4^7 mögliche Farbkonstellationen prüfen.
- Allgemein sind m^n Farbkonstellationen zu prüfen,
  mit $m :=$ Anzahl der Farben und $n :=$ Anzahl der Länder.

$\Rightarrow$ Ineffizient!

Naives generate-and-test Verfahren:

WA	NT	SA	QL	NSW	Vic	Tas
rot	rot	rot	rot	rot	rot	rot
rot	rot	rot	rot	rot	rot	blau
rot	rot	rot	rot	rot	rot	grün
rot	rot	rot	rot	rot	rot	gelb
rot	rot	rot	rot	rot	blau	rot
rot	rot	rot	rot	rot	blau	blau
rot	rot	rot	rot	rot	blau	grün
rot	rot	rot	rot	rot	blau	gelb
rot	rot	rot	rot	rot	grün	rot
rot	rot	rot	rot	rot	grün	blau
rot	rot	rot	rot	rot	grün	grün
rot	rot	rot	rot	rot	grün	gelb
...	...	...	...	...	...	...

Es scheint sinnvoller zu sein, die Länder der Reihe nach zu färben.

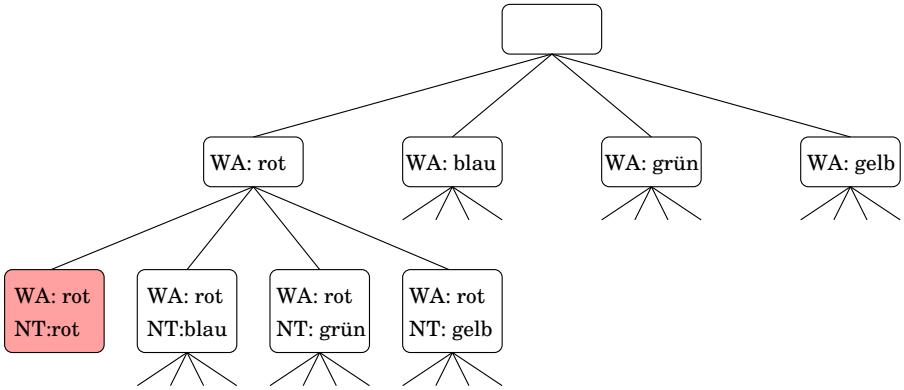


- Beschreibe Zwischenzustände durch Teilfärbungen, etwa (WA:rot, NT:blau, SA:gelb).
- Nach der teilweisen Zuordnung (WA:blau, NT:blau) wird direkt abgebrochen, da die Länder benachbart sind.

- Die Problemlösung startet mit der leeren Färbung ().
- Ziel ist es, eine komplette, zulässige Färbung zu erreichen.
- Die Schritte im Laufe der Problemlösung lassen sich durch Zustandsübergangsoperatoren beschreiben.

Die Lösung des Färbeproblems lässt sich als Suchbaum darstellen:

- Knoten entsprechen den Zuständen (zulässige Teilfärbungen).
- Kanten entsprechen den Operatoren.



- Ungültige Teillösungen werden nicht weiter verfolgt.

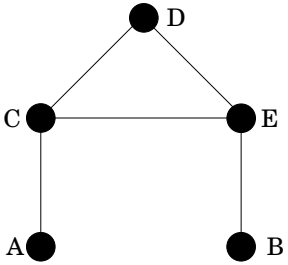
Für Suchprobleme lässt sich das Wissen wie folgt repräsentieren:

- Ein Zustand repräsentiert das Wissen zu einem bestimmten Zeitpunkt der Lösungsfindung.
- Die Menge aller Zustände ist der Zustandsraum.
- Zustandsübergangsoperatoren beschreiben den Wechsel von einem Zustand zu einem anderen Zustand des Zustandsraums.
- Den Zustand zu Beginn der Lösungsfindung nennt man Startzustand, er lässt sich explizit angeben.
- Die Menge der Zielzustände charakterisiert die Lösungen des Problems.
- Zielzustände lassen sich oft nur implizit angeben, z.B. über ein Testprädikat.
- Um verschiedene Lösungen vergleichen zu können, können Folgen von Aktionen Kosten zugewiesen werden.

  Typischerweise definiert man Kosten abhängig von einem Zustand und einer Aktion. Die Gesamtkosten für eine Aktionenfolge ergeben sich aus den Einzelkosten.

Gegeben ist eine Karte mit Städten und Straßen, die die Städte miteinander verbinden.

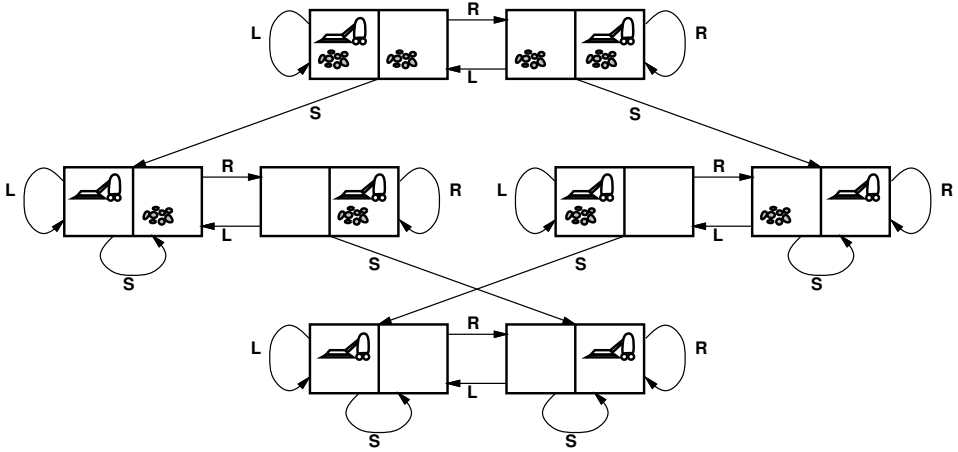
Gesucht ist eine Route von einem Startort zu einem Zielort.



Übung 11.

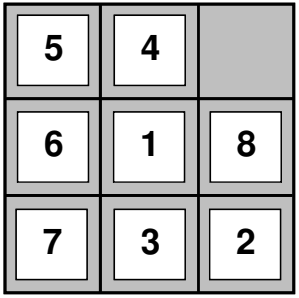
- Erstellen Sie den Suchbaum zu obigen Beispiel für den Fall, dass Sie eine Route von A nach B finden wollen.
- Wie können Sie verhindern, dass die Route Kreise enthält?

Staubsaugerwelt

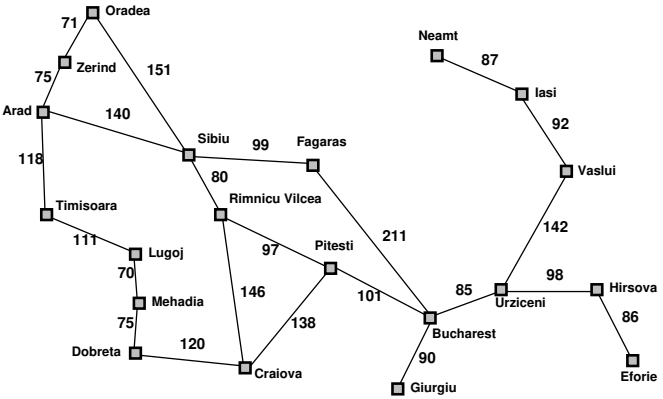


- *Operationen:* rechts, links, saugen
- *Ziel:* alle Räume müssen sauber sein

8-Puzzle



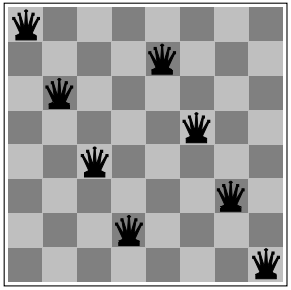
- *Operationen:* schiebe das leere Feld nach rechts, links, oben oder unten
- *Ziel:* Endkonfiguration



Straight−line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

- *Operationen:* Gehe von einer Stadt zu einer benachbarten Stadt entlang einer Kante.
- *Ziel:* Der kürzeste Weg vom Start zum Ziel ist berechnet.
- Der Zielzustand ist hier nicht explizit bekannt. Wie sieht in diesem Fall das Testprädikat aus?



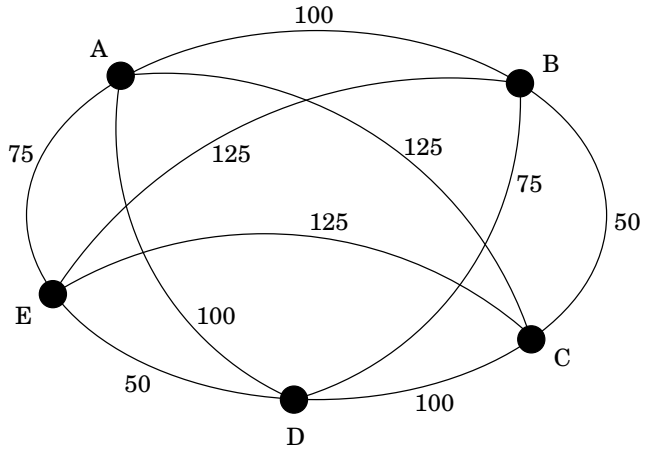
- *Zustand:* Plazierung zwischen 0 und 8 Damen.
- *Operationen:* Eine Dame plazieren.
- *Ziel:* Alle Damen sind so plaziert, dass sie sich nicht gegenseitig bedrohen.

Übung 12.

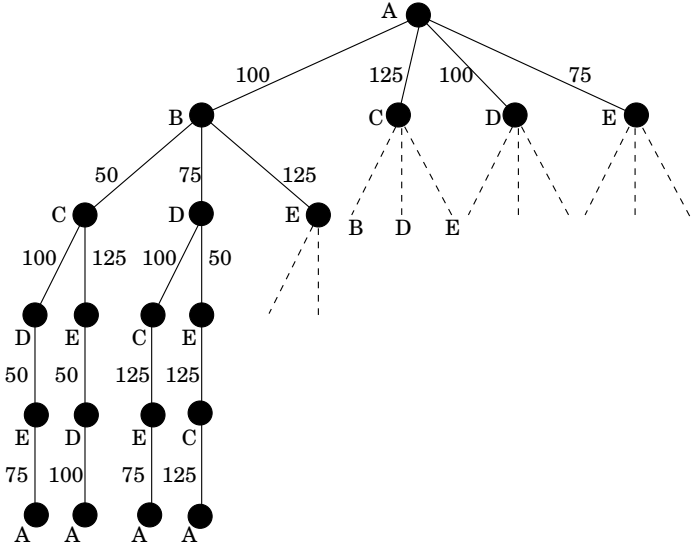
- Auch bei diesem Problem sind die Endzustände nicht explizit bekannt. Wie sieht hier das Testprädikat aus?
- Implementieren Sie eine Zustandsraumsuche und ein naives generate-and-test Verfahren und vergleichen Sie die Lösungen.
- Wie kann der Suchraum verkleinert werden? Bringt das signifikante Laufzeitvorteile?

Problem des Handlungsreisenden

Finde den kürzesten Reiseweg, der genau einmal durch alle Städte führt und wieder bei der ersten Stadt endet.



Problem des Handlungsreisenden



Da die Stadt, in der die Rundreise beginnt, festgelegt ist, hat die erschöpfende Suche bei n Städten eine Komplexität von $(n-1)!$.

Reduktion der Suchkomplexität durch *Branch-and-Bound*:

- Generiere zu einem Zeitpunkt jeweils nur einen Weg.
- Die Kosten des bisher besten Weges wird als Schranke bei weiteren Kandidaten eingesetzt.
- Ist ein Teilweg bereits teurer als die bisher beste Lösung, so werden alle Erweiterungen des Weges nicht weiter betrachtet.
- Weitere Reduktion durch Schätzen der verbleibenden Kosten:
  - Jede der noch nicht besuchten Städte muss besucht und auch wieder verlassen werden.
  - Addiere jeweils die Werte der zwei kostengünstigsten Kanten, die inzident zu einem noch unbesuchten Knoten sind.
  - Die Summe muss noch durch 2 dividiert werden, da jede Kante zweimal genutzt wird: um einen Knoten zu verlassen und um den nächsten zu besuchen.

$\rightarrow$ Komplexität: 1.26^n, leider immer noch exponentiell!

Die Berechnung der Nachfolger eines Knotens s wird als *Expansion* des Knotens s bezeichnet.

Der Zustandsraum beschreibt lediglich, *wie* man zu einer Lösung gelangen kann, aber nicht, wie man effizient zu dieser kommt.

Wesentlich für eine effiziente Problemlösung sind:

- Die Reihenfolge, in der die Zustände untersucht bzw. expandiert werden
- sowie die Bewertung der einzelnen Zustände.

Übersicht

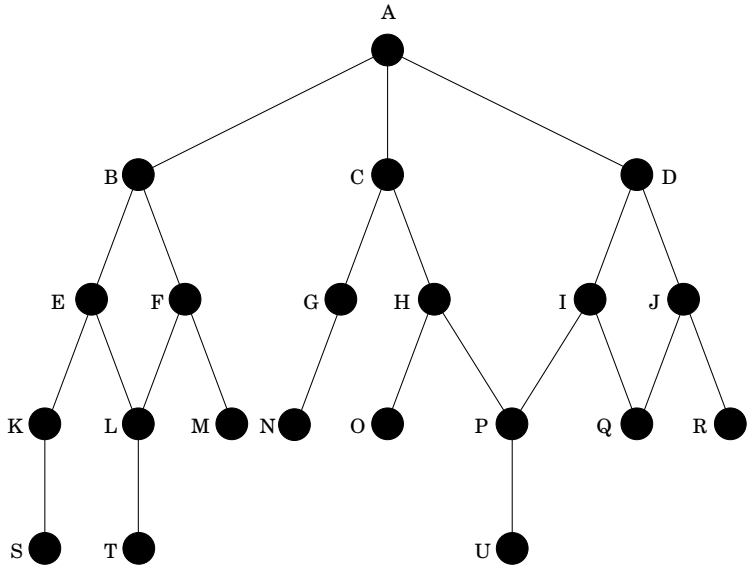
Uninformierte Suchverfahren

- Suchverfahren, die über die Beschreibung des Zustandsraums hinaus keine Zusatzinformation benutzen.
- Insbesondere findet keine Bewertung der einzelnen Zustände statt. Um Wiederholungen zu vermeiden, werden bereits untersuchte Zustände markiert oder gespeichert.
- Die Verfahren unterscheiden sich im wesentlichen darin, in welcher Reihenfolge die Zustände expandiert werden.
- Die wichtigsten Vertreter der uninformierten Suchverfahren sind die *Breitensuche* und die *Tiefensuche*.
- Ausgehend von der Wurzel des Suchbaums (Startzustand) werden die Knoten sukzessive expandiert.
- Später wird von den Nachfolgern des expandierten Knotens weiter gearbeitet, solange bis ein Zielknoten gefunden wurde.

Breiten- und Tiefensuche laufen nach dem gleichen Schema ab:

- *Agenda*: Liste der Knoten, die gerade in Bearbeitung sind.
- Knoten der Agenda sind generiert, aber noch nicht expandiert.
- Zu Beginn der Suche besteht die Agenda nur aus dem Startzustand.
- In einer beliebigen Iteration wird der erste Knoten s_{akt} aus der Agenda genommen.
- Wenn s_{akt} ein Zielzustand ist, hat man eine Lösung gefunden.
- Ist s_{akt} kein Zielzustand, so wird s_{akt} expandiert, d.h. alle Nachfolger von s_{akt} werden in die Agenda eingefügt.
- Breiten- und Tiefensuche unterscheiden sich darin, wo die Nachfolger in die Agenda eingefügt werden.

Beispiel für Tiefen- und Breitensuche

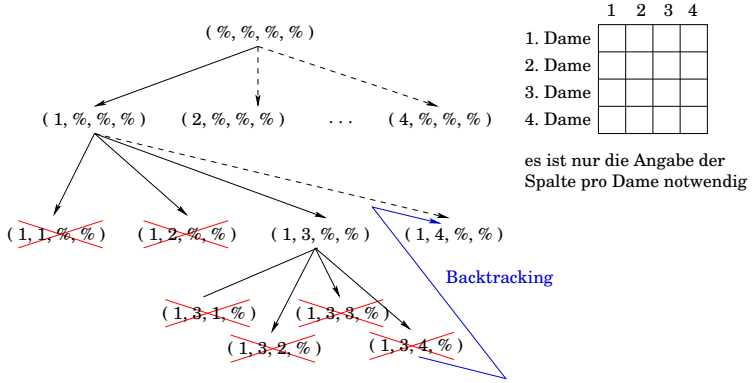


Bei der Tiefensuche werden die Nachfolger eines expandierten Knotens an den Anfang der Agenda eingefügt. Die Agenda entspricht einem Kellerspeicher (Stack).

```
Node * DFsearch(Node * pRoot) {
    Stack<Node *> agenda;    // Tiefensuche
    agenda.push(pRoot);
    while (!agenda.isEmpty()) {
        Node *pCurr = agenda.pop();
        if (zielknoten(pCurr))
            return pCurr;
        for (Node* n : nachfolger(pCurr))
            agenda.push(n);
    }
    return null;
}
```

- Liefert ein Knoten, der kein Zielknoten ist, keine neuen Knoten, so wird die Suche fortgesetzt an dem nächstgelegenen Knoten, für den noch nicht alle Nachfolger expandiert wurden.

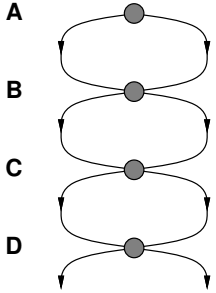
→ Das Verhalten entspricht einem Backtracking (zurückverfolgen).

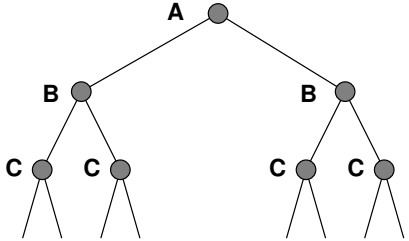


- Die Abarbeitung wird an den Knoten abgebrochen, die zu keiner zulässigen Lösung führen können.

Wie verhindern wir, dass Zustände wiederholt untersucht werden?

- Problem tritt immer dann auf, wenn Operationen umkehrbar sind: Schiebepuzzle, Routenproblem, Spiele, usw.
- Wird nicht erkannt, das Zustände bereits untersucht wurden, kann aus einem linearen Problem ein exponentielles werden:





- Der einzige Weg, wiederholtes Untersuchen von Zuständen zu vermeiden, ist: Speichere bereits besuchte Zustände. → Tradeoff between space and time!

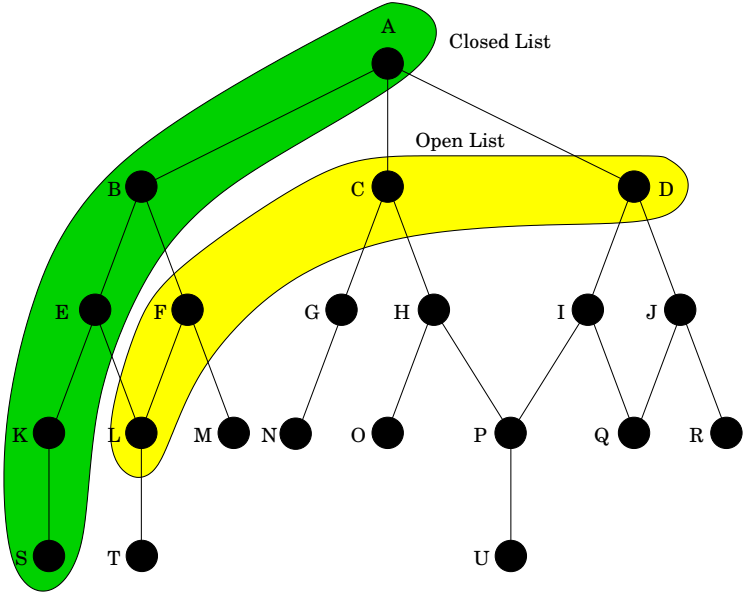
Repeated States

- *Algorithms that forget their history are doomed to repeat it.*
- Um die bereits untersuchten Zustände zu speichern, werden expandierte Knoten in der *closed list* verwaltet.
- Wenn der aktuell zu untersuchende Zustand bereits in der closed list gespeichert ist, wird er verworfen anstatt expandiert zu werden.
- Die Agenda wird daher auch *open list* genannt.

Die um eine Closed-List erweiterte Tiefensuche zur Vermeidung wiederholter Zustände:

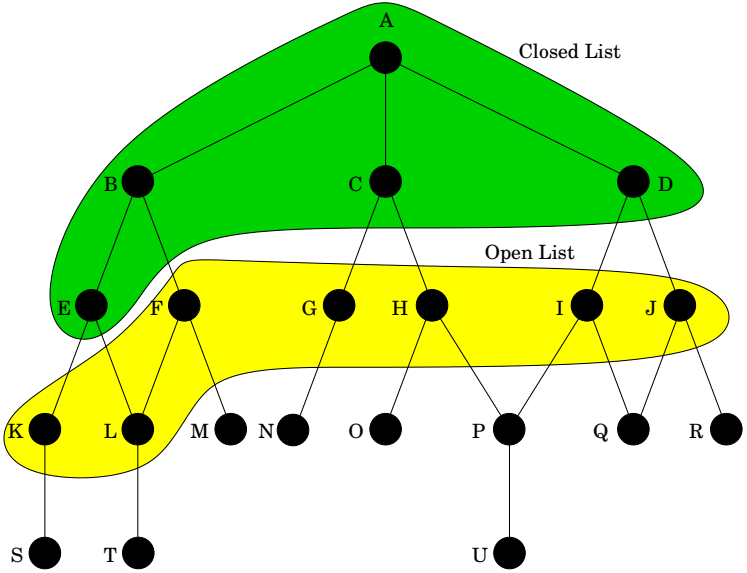
```
Node * search(Node * pRoot) {
    Stack<Node *> openList;    // Tiefensuche
    openList.push(pRoot);
    HashSet<Node *> closedList;
    while (!openList.isEmpty()) {
        Node *pCurr = openList.pop();
        if (zielknoten(pCurr))
            return pCurr;
        if (!closedList.contains(pCurr)) {
            closedList.insert(pCurr);
            for (Node *n : nachfolger(pCurr))
                openList.push(n);
        }
    }
    return null;
}
```

Tiefensuche



Bei der Breitensuche werden die Nachfolger eines expandierten Knotens an das Ende der Agenda eingefügt. Die Agenda entspricht einer Warteschlange (Queue).

```
Node * BFsearch(Node * pRoot) {
    Queue<Node *> openList;    // Breitensuche
    openList.push(pRoot);
    HashSet<Node *> closedList;
    while (!openList.isEmpty()) {
        Node *pCurr = openList.pop();
        if (zielknoten(pCurr))
            return pCurr;
        if (!closedList.contains(pCurr)) {
            closedList.insert(pCurr);
            for (Node *n : nachfolger(pCurr))
                openList.push(n);
        }
    }
    return null;
}
```


A
Closed List
B
C
D
Open List
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U

Übung 13. Ein Weinhändler hat drei Krüge mit 9l, 7l bzw. 4l Inhalt.

- Auf den Krügen sind keine Litermarkierungen angebracht.
- Der 9l-Krug ist gefüllt, die anderen beiden sind leer.
- Die Krüge sollen so umgefüllt werden, dass der 9l-Krug sechs Liter und der 4l-Krug drei Liter enthält.

Erstellen Sie den Suchbaum und ermitteln Sie mittels Tiefen- und Breitensuche eine Lösung des Problems.

Übung 14. Betrachten Sie das Kannibalen und Missionare-Problem:

- 3 Missionare und 3 Kannibalen wollen einen Fluss überqueren.
- Es steht nur ein Ruderboot für die Überfahrt zur Verfügung.
- Das Boot kann maximal 2 Personen aufnehmen.
- Es dürfen nie mehr Kannibalen als Missionare an einem Ort sein, sonst gibt es ein großes Fressen.

Aufgabe:

- Definieren Sie die Zustandsmenge und die Operatoren, die einen Zustand in einen anderen Zustand überführen.
- Erstellen Sie den Suchbaum und ermitteln Sie mittels Tiefen- und Breitensuche eine Lösung des Problems. Zustände, bei denen mehr Kannibalen als Missionare an einem Ort sind, werden nicht weiter verfolgt.

Komplexität der Suchverfahren

Ungünstiger Fall für Breiten- und Tiefensuche: Die Lösung liegt in der „äußersten, unteren rechten Ecke“ des Suchbaums.

- Sei b die Verzweigungsrate und
- sei t die Tiefe des Zielknotens.

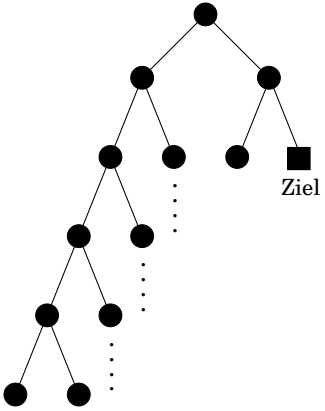
Breitensuche: Die Agenda kann eine komplette Ebene des Suchbaums enthalten, daher erhalten wir:

- Zeitkomplexität: $\Theta(b^t)$
- Platzkomplexität: $O(b^t)$

Tiefensuche: Sei m die maximale Tiefe des Baums.

- Platzkomplexität: $O(b \cdot m)$, denn die Agenda enthält nur die Knoten des aktuellen Suchpfades sowie deren Nachfolger.
- Zeitkomplexität: $O(b^m)$

Laufzeit Tiefensuche: $O(b^m)$



$b = 2$ (Verzweigungsrate)
$m = 5$ (Tiefe des Baums)
$t = 2$ (Tiefe des Ziels)

Eine Breitensuche würde das Ziel viel schneller finden. Leider ist bei vielen Problemen keine Breitensuche einsetzbar, z.B. bei Schach oder Go, da dort die Verzweigungsrate zu hoch ist!

Eigenschaften von Suchverfahren

Ein Suchverfahren heißt *vollständig*, wenn für jeden Suchbaum jeder Knoten expandiert werden könnte, solange noch kein Zielknoten gefunden wurde.

- Ein vollständiges Suchverfahren ist fair in dem Sinne, dass jeder Knoten die Chance hat, expandiert zu werden.
- Ein vollständiges Suchverfahren findet auch bei unendlichen Suchbäumen stets eine Lösung, falls eine existiert.

Nach dieser Definition gilt: Die Breitensuche ist vollständig.

- Tiefensuche ist nur bei endlichen Suchbäumen vollständig. Hat ein Zweig im Suchbaum unendliche Tiefe, dann können Knoten in anderen Zweigen nicht expandiert werden.

Für ein uninformiertes Suchverfahren heißt eine Lösung *optimal*, wenn sie unter allen Lösungen die geringste Tiefe im Suchbaum aufweist. Nach dieser Definition gilt:

- Die Breitensuche findet eine optimale Lösung, falls eine Lösung existiert.
- Eine Tiefensuche findet im Allgemeinen keine optimale Lösung.

Tiefensuche mit Begrenzung der Suchtiefe

Wir versuchen, die Nachteile der Tiefensuche auszuschalten, indem wir eine maximale Suchtiefe vorgeben:

- Zweige jenseits der maximalen Suchtiefe werden abgeschnitten.
- Dadurch wird garantiert, dass eine Lösung gefunden wird, wenn eine innerhalb der Suchtiefe existiert.

  Frage: Stimmt das so ohne Weiteres? Was ist mit den in der Closed-List gespeicherten Zuständen?
- Nachteil: Wie groß soll man die maximale Suchtiefe wählen?

Da man nicht weiß, wie groß man die maximale Suchtiefe wählen soll, probiert man dies einfach systematisch durch: schrittweise vertiefende Suche

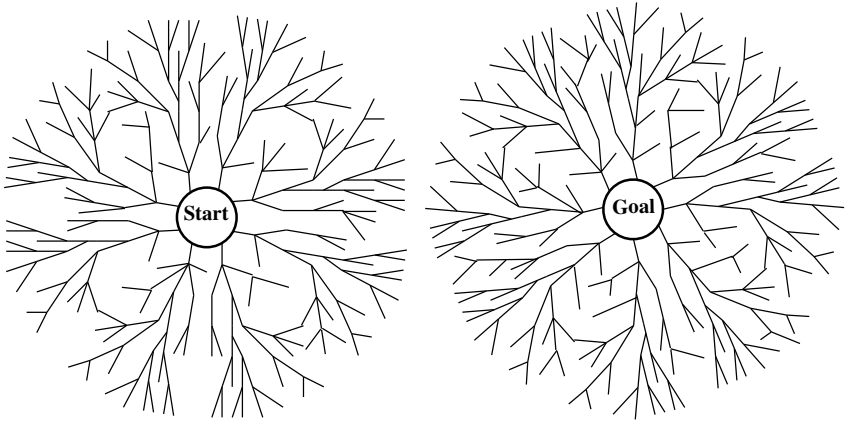
- Führe immer wieder eine Tiefensuche mit Begrenzung der Suchtiefe durch und erhöhe nach einer erfolglosen Suche die maximale Suchtiefe jeweils um eins.
- Bei einer maximalen Suchtiefe t und einem regulären Suchbaum mit Verzweigungsgrad b werden $O(b^t)$ Knoten erzeugt.
- Man verliert bei diesem Verfahren nur einen konstanten Faktor[(25)]:

$$\sum_{d=0}^{t} b^d = \frac{b^{t+1}-1}{b-1} < \frac{b}{b-1} b^t = O(b^t)$$

[(25)] dies gilt nur, falls der Verzweigungsgrad konstant ist, also die Suche einen exponentiellen Zeitaufwand hat

Bidirektionale Suche

- Gleichzeitige Suche vom Start zum Ziel und vom Ziel zum Start.
- Lösungen ergeben sich dort, wo sich die Suchpfade treffen.



Durch Halbierung der Länge der Suchpfade ergibt sich ein Aufwand[(26)] von $O(2b^{\frac{d}{2}}) = O(b^{\frac{d}{2}})$.

$\Rightarrow$ Drastische Reduzierung der Laufzeit! Die in akzeptabler Zeit lösbare Problemgröße kann verdoppelt werden.

Probleme:

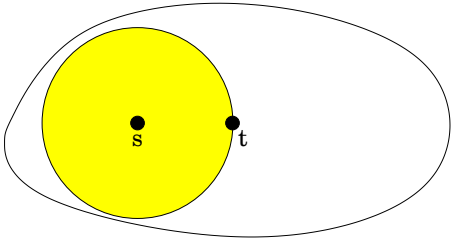
- Zielzustände sind oft nicht explizit bekannt: n-Damen-Problem
- Zielzustände müssen nicht eindeutig sein: n-Damen-Problem
- Operatoren sind nicht unbedingt umkehrbar.

Übung 15. Implementieren Sie eine bidirektionale Suche für Brett-Solitär. Ergibt sich eine signifikante Verbesserung der Laufzeit gegenüber Ihrem ursprünglichen Programm?

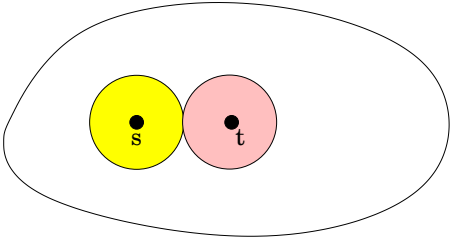
[(26)] dies gilt nur, falls der Verzweigungsgrad konstant ist, also die Suche einen exponentiellen Zeitaufwand hat

Bei der Routenplanung mittels Dijkstras Algorithmus kann die Berechnung durch die bidirektionale Suche nur um den Faktor 2 beschleunigt werden.

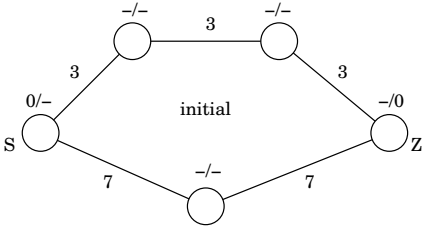
bisher: $d(s,t)^2 \cdot \pi$

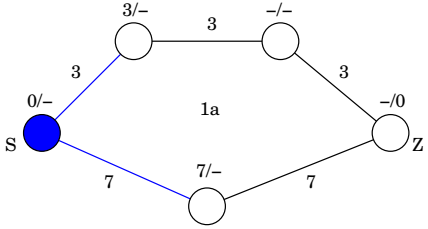


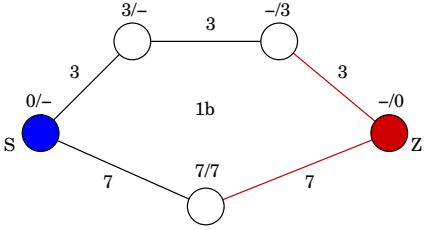
jetzt: $2 \cdot (\frac{d(s,t)^2}{4} \cdot \pi)$



Außerdem darf bei der Routenplanung die Suche nicht dann abgebrochen werden, wenn sich zwei Suchpfade treffen.

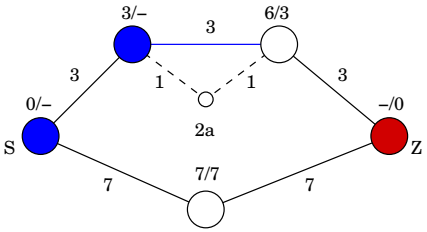






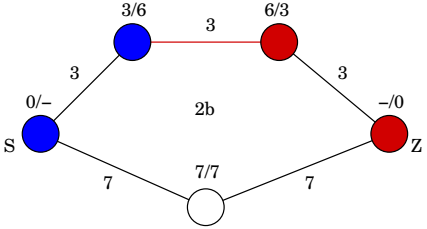
Die Suchpfade treffen sich zwar in dem unteren Knoten, aber der kürzeste Weg wurde noch nicht gefunden.

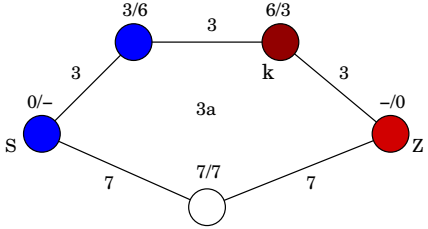
Die Suche wird also fortgesetzt.



Auch hier darf die Suche noch nicht abgebrochen werden, denn es könnte noch einen kürzeren Weg geben.

Die Suche wird also fortgesetzt.

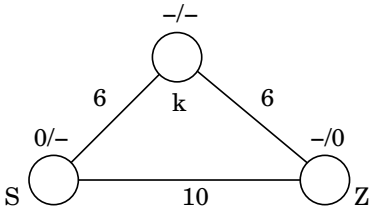




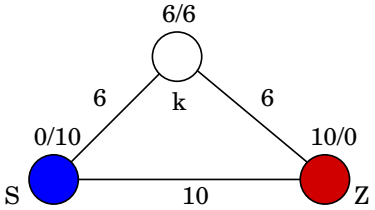
Erst wenn ein Knoten k von beiden Suchrichtungen aus als final markiert wurde, kann die Suche abgebrochen werden.

Frage: Ist k auf kürzesten Weg vom Start zum Ziel enthalten?

Nein! Ein Knoten k, der von beiden Suchrichtungen aus als final markiert wurde, muss nicht notwendig auf einem kürzesten Weg vom Start zum Ziel enthalten sein.



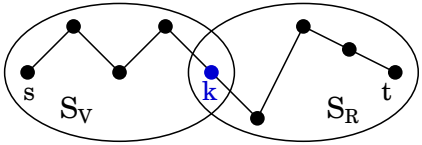
Bei beiden Suchrichtungen wird im ersten Schritt jeweils die Distanz zum Knoten k auf den Wert 6 aktualisiert.

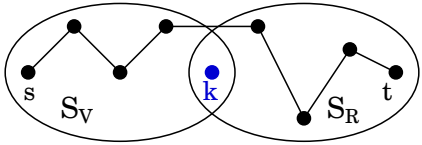


Im nächsten Schritt wird jeweils der Knoten k ausgewählt und aus der Priority-Queue entfernt. Aber k liegt nicht auf dem kürzesten Weg vom Start S zum Ziel Z.

Aber: Der kürzeste Weg wurde korrekt berechnet, sobald ein Knoten k gefunden wurde, der von beiden Suchen als final markiert wurde.

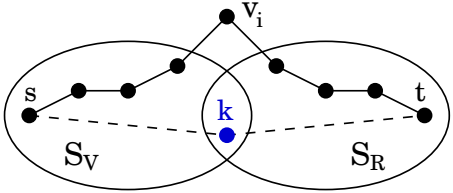
- natürlich gilt: $\overleftarrow{d[v]} + \overrightarrow{d[v]} \geq dist(s,t)$
- Seien S_V und S_R die abgearbeiteten Knoten der Vorwärts- bzw. Rückwärtssuche nach Terminierung.
- Sei $P = (v_1, \ldots, v_m)$ ein kürzester Weg vom Start zum Ziel.
- zeige:
  - $\{v_1, \ldots, v_m\} \subseteq S_V \cup S_R$ oder





  - $|P| = dist(s,k) + dist(k,t)$

- Angenommen, es gibt ein $v_i \notin S_V \cup S_R$



- Dann gilt $dist(s, v_i) \geq dist(s, k)$ und $dist(v_i, t) \geq dist(k, t)$, da sonst Dijkstra Knoten v_i ausgewählt hätte und nicht k.
- Also ist $|P| = dist(s, v_i) + dist(v_i, t) \geq dist(s, k) + dist(k, t)$.
- Da P nach Voraussetzung ein kürzester Weg ist, muss gelten: $|P| = dist(s, v_i) + dist(v_i, t) = dist(s, k) + dist(k, t)$

Übersicht

- Für größere Suchbäume sind Breiten- und Tiefensuche nicht effizient genug.
- Vielversprechender sind Ansätze, bei denen Problemwissen zur Steuerung des Suchprozesses eingesetzt wird.
- Dies kann dadurch geschehen, dass die Zustände (Knoten) danach bewertet werden, wie erfolgversprechend sie sind.
- Man schätzt bspw. für jeden Knoten, wie nahe er an einem Zielknoten liegt.
- Solch eine Bewertung nennt man heuristische Funktion.
  - griech. eurisco heißt „ich entdecke“
  - „Heureka“: ich habe es gefunden
- Heuristik bei Zustandsraumsuche: Regeln zur Auswahl solcher Zweige, die mit hoher Wahrscheinlichkeit zu einer akzeptablen Problemlösung führen.

Wir lernen die Zustandsraumsuche im wesentlichen bei Spielen kennen, weil

- die Suchräume groß genug sind, um eine heuristische Beschneidung (engl. Pruning) zu erfordern.
- die meisten Spiele komplex genug sind, so dass eine Vielzahl heuristischer Bewertungen für den Vergleich und die Analyse einsetzbar sind.
- Spiele i.Allg. keinen komplexen Repräsentationsaufwand erfordern, so dass man sich auf das Verhalten der Heuristik statt auf Probleme der Wissensrepräsentation konzentrieren kann.
- alle Knoten des Zustandsraums eine gemeinsame Repräsentation besitzen, so dass im gesamten Suchraum eine einzige Heuristik angewendet werden kann.

Wann werden informierte Suchverfahren eingesetzt?

- Wenn ein Problem keine exakte Lösung hat, weil bspw. die Problembeschreibung oder die verfügbaren Daten zweideutig sind.
  - Medizinische Diagnose: Eine Menge von Symptomen kann verschiedene Ursachen haben.
    → Wähle die wahrscheinlichste Lösung.
  - Visuelle Wahrnehmung: Optische Täuschung, Ausrichtung oder Orientierung von Objekten nicht klar.
    → Wähle die wahrscheinlichste Interpretation der Szene.
- Wenn die Rechenkosten für die Suche nach einer exakten Lösung zu groß sind, weil die Anzahl der Zustände exponentiell oder faktoriell mit der Tiefe der Suche wächst, wie z.B. bei Schach oder Go.

→ Suche entlang eines „viel versprechenden“ Weges.

Anwendungen:

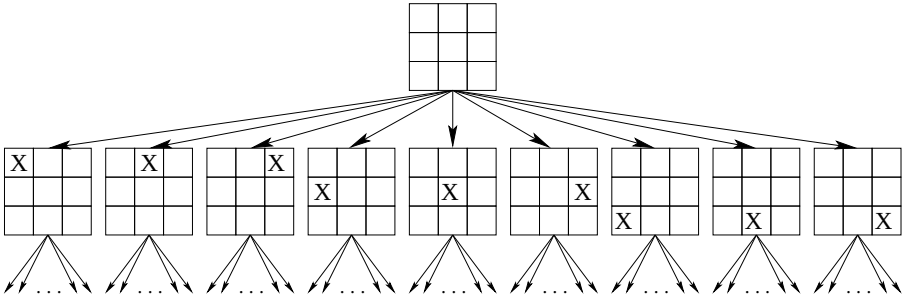
- Spiele: Es kann nicht jede erreichbare Spielsituation beim Schach oder Go weiterverfolgt werden.
- Theorembeweise: Es kann nicht jede Schlussfolgerung weiterverfolgt werden.

Expertensysteme:

- Heuristiken sind essentielle Komponenten von Problemlöseverfahren.
- Menschliche Experten verwenden Faustregeln.
- Faustregeln sind weitgehend heuristisch.

$\rightarrow$ Extrahiere und formalisiere Faustregeln.

Tic-Tac-Toe ohne Heuristik:

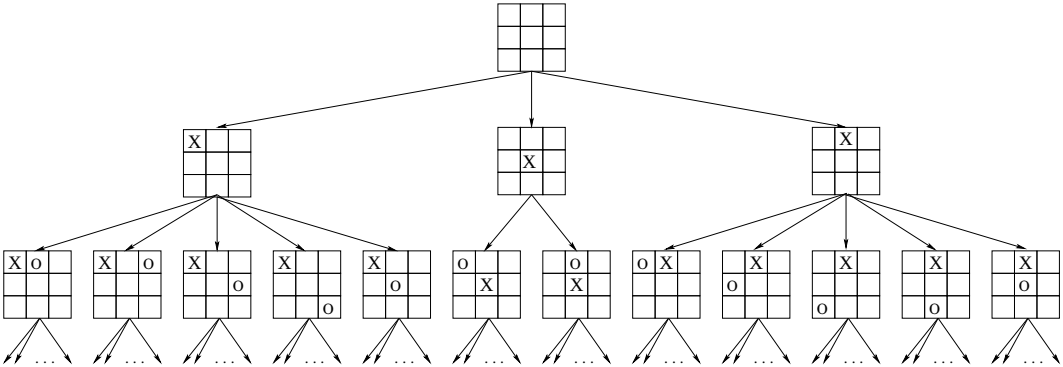


$\Rightarrow$ $9! = 9 \cdot 8 \cdot 7 \cdot \ldots \cdot 2 \cdot 1 = 362.880$ mögliche Zustände

Beachte: Viele doppelte Zustände durch Zuggenerierung. Es gibt weniger als $3^9 = 19.683$ verschiedene Zustände!

$\Rightarrow$ Es scheint sinnvoll zu sein, bereits untersuchte Zustände in einer Hash-Tabelle zu verwalten.

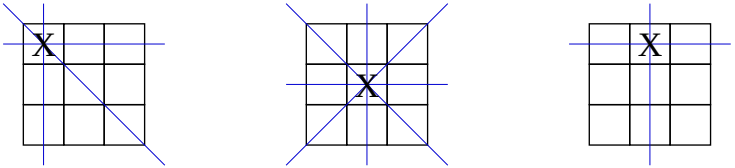
Reduktion bzgl. Symmetrien bei Tic-Tac-Toe:



$\Rightarrow 12 \cdot 7! = 60.480$ mögliche Zustände

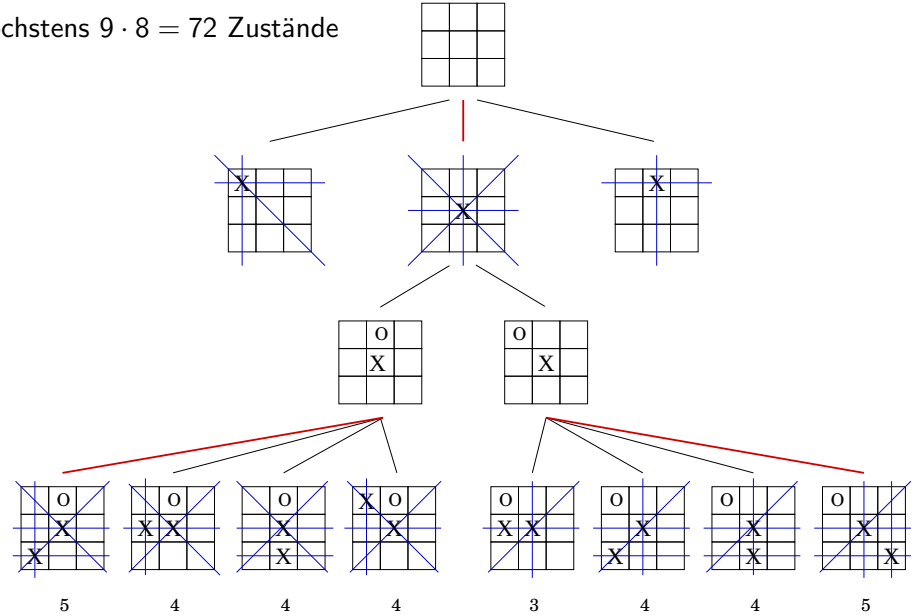
Problem: Wie baut man einen Zuggenerator, der Symmetrien berücksichtigt?

Heuristik der „meisten Gewinnmöglichkeiten“ bei Tic-Tac-Toe:



Wir beschränken also die Suche auf die Nachfolgezustände des obigen, mittleren Zustands. Anschließend wenden wir noch die Reduktion bzgl. Symmetrien an.

$\Rightarrow$ höchstens $9 \cdot 8 = 72$ Zustände

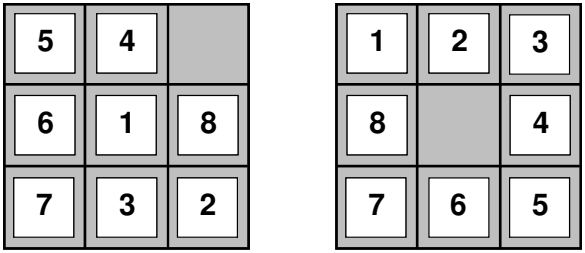


Heuristische Funktion

Eine Funktion, die jedem Zustand (Knoten) s eines Zustandsraums (Suchbaums) eine nichtnegative Zahl $h(s)$ zuordnet, heißt *heuristische Funktion*. Für einen Zielzustand s gilt dabei $h(s) = 0$.

Ein Suchverfahren, das eine heuristische Funktion zur Auswahl der zu expandierenden Zustände einsetzt, heißt *informiertes Suchverfahren* oder auch *heuristisches Suchverfahren*.

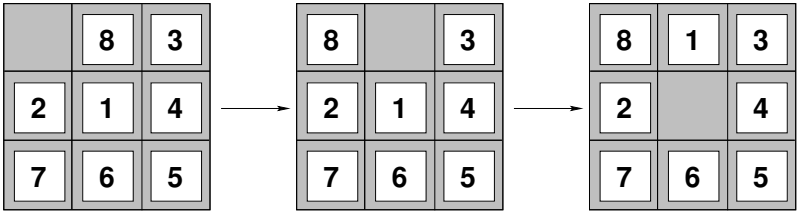
Beispiel Schiebepuzzle:



Mögliche heuristische Funktionen:

- $h_1(s)$: Anzahl der Plättchen, die nicht an der richtigen Stelle liegen. Hier: $h_1(s) = 7$
- $h_2(s)$: Summe der Entfernungen aller Plättchen von der Zielposition, also die Manhatten-Distanz. Hier: $h_2(s) = 2 + 3 + 3 + 2 + 4 + 2 + 0 + 2 = 18$

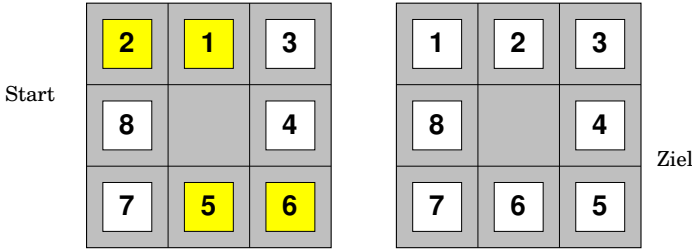
Heuristische Funktion



- Die heuristische Funktion h_2 differenziert stärker als h_1, d.h. h_2 kann Zustände unterscheiden, die von h_1 gleich bewertet werden.
- Eine heuristische Funktion ist um so brauchbarer, je mehr Zustände sie unterschiedlich bewertet.
- Eine heuristische Funktion, die alle Zustände gleich bewertet, ist unbrauchbar.

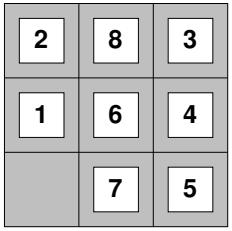
Beide Heuristiken sind zu kritisieren, da sie die Schwierigkeiten eines Spielsteintauschs nicht berücksichtigen:

- Es sind wesentlich mehr als zwei Verschiebungen notwendig, da die Plättchen umeinander herum verschoben werden müssen.
- Man kann eine kleine Zahl wie 2 auf die Heuristik multiplizieren, um dies zu berücksichtigen.

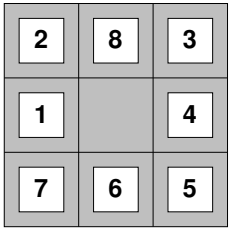


- $h_3(s) = 2 \times$ Anzahl direkter Spielsteintausch

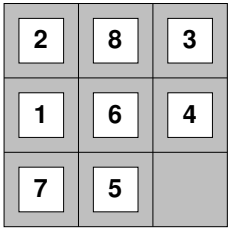
- h_3 versagt, wenn ein Zustand keinen direkten Tausch enthält:



$h_1: 5$ $h_2: 6$ $h_3: 0$



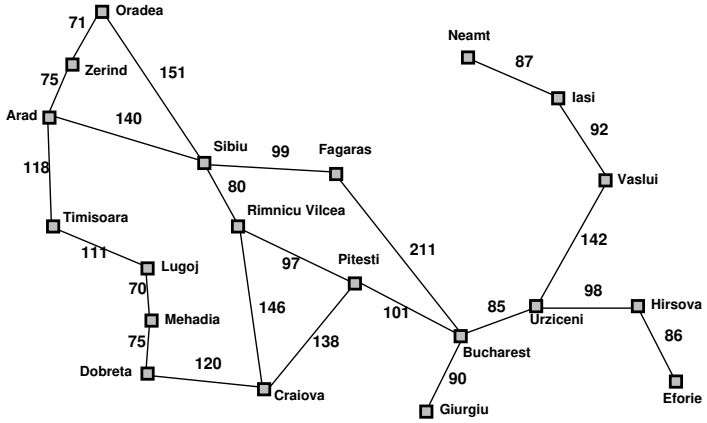
$h_1: 3$ $h_2: 4$ $h_3: 0$



$h_1: 5$ $h_2: 6$ $h_3: 0$

- Heuristik h_4: Die Summe der Entfernungen plus das Doppelte der Anzahl direkter Tauschkonfigurationen.
- Heuristik: Begrenzte Information über Zustandsbeschreibung einsetzen, um intelligente Auswahl zu treffen.
- Jede obige Heuristik ignoriert wichtigen Informationsanteil.
- Urteilsvermögen und Intuition sind hilfreich, aber Bewertung basiert letztendlich auf tatsächlicher Leistung beim Lösen von Probleminstanzen.

Beispiel Routenplanung: Die Straßenentfernung wird durch die Luftlinienentfernung abgeschätzt, hier am Beispiel von Rumänien:



Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Bestensuche

Bei der Bestensuche erfolgt die Expansion eines Knotens auf Basis der heuristischen Funktion:

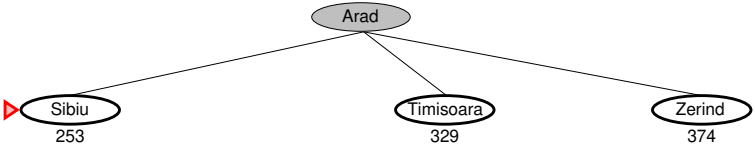
- Die Heuristik wählt den vielversprechendsten Zustand aus.
- Hierzu werden in der Agenda die Knoten zusammen mit ihrer Bewertung abgelegt.
- Es wird jeweils der Knoten der Agenda expandiert, der die geringste Bewertung aufweist.
- Die Agenda hat also die Form einer Prioritätswarteschlange (priority queue).
- Ansonsten ist die Bestensuche analog zur Tiefen- und Breitensuche.

Algorithmus:

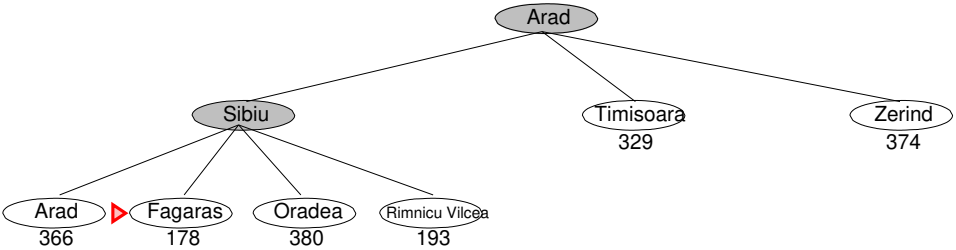
```
Node * BestSearch(Node *pRoot) {
    PriorityQueue<Node *> agenda;
    agenda.insert(pRoot, h(pRoot));
    HashSet<Node *> closedList;
    while (!agenda.isEmpty()) {
        Node *pCurr = agenda.extractMin();
        if (zielknoten(pCurr))
            return pCurr;
        if (!closedList.contains(pCurr)) {
            closedList.insert(pCurr);
            for (Node *n : nachfolger(pCurr))
                agenda.insert(n, h(n));
        }
    }
    return null;
}
```

Routenplanung:

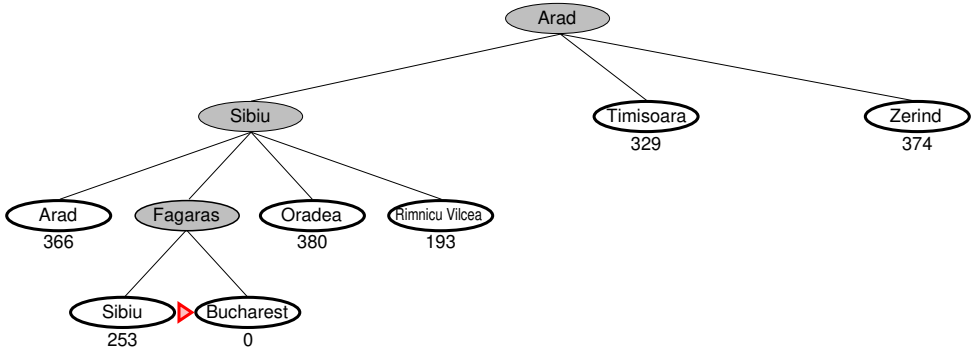
Routenplanung:



Routenplanung:



Routenplanung:



- Die gefundene Lösung ist nicht optimal!
- Bestensuche wählt Route über Sibiu und Fagaras, obwohl die Route über Rimnicu Vilcea und Pitesti kürzer ist.
- Heuristiken können versagen: Eine Heuristik ist nur eine informierte Vermutung, basierend auf Erfahrung und Intuition.

Die Bestensuche ist anfällig für falsche Startknoten!

- Beispiel: Suche Route von Iasi nach Fagaras.
- Zuerst wird Neamt expandiert, weil es näher am Ziel ist als Vaslui, aber die Route führt in eine Sackgasse.
- Die richtige Route führt zunächst über Vaslui, obwohl es weiter vom Ziel entfernt ist als Neamt.

→ Heuristik expandiert unnötige Knoten.

Ohne Erkennung wiederholter Zustände würde die Suche zwischen Iasi und Neamt oszillieren.

Analogie zur Tiefensuche:

- Verfolge nur einen Pfad zum Ziel, um die Speicherkomplexität gering zu halten
- und springe zurück, falls in eine Sackgasse verzweigt wurde.

Gleiche Nachteile wie Tiefensuche:

- Die Bestensuche ist nicht optimal und
- nicht vollständig (kann in unendlichem Pfad hängen bleiben).

Komplexität: Sei b die Verzweigungsrate und m die maximale Tiefe des Baums. Dann ergeben sich folgende Komplexitäten:

- Zeit: $O(b^m)$
- Platz: $O(b^m)$

Eine heuristische Funktion h heißt *fair* gdw. es zu jedem $n \geq 0$ nur endlich viele Knoten s gibt mit $h(s) \leq n$.

- Fairness entspricht der Vollständigkeit bei uninformierten Suchverfahren.
- Ist eine heuristische Funktion fair, so wird ein Zielknoten gefunden, falls ein solcher existiert.

Problem:

- Die Bestensuche vernachlässigt die Kosten bei der Anwendung der Operatoren.
- Wird die Güte einer Lösung charakterisiert durch diese Operatorkosten, so findet die Bestensuche i.Allg. keine optimale Lösung.
  - Routenberechnung: Distanz vom Start zum aktuellen Knoten
  - 8-Puzzle: Anzahl bisher durchgeführter Züge

Heuristiken sind fehlbar, daher verfolgt ein Suchalgorithmus evtl. einen Weg, der nicht zu einem Ziel führt.

- Lösung des Problems bei Tiefensuche: Ein Tiefenzähler wurde genutzt, um aussichtslose Wege zu entdecken.
- Diese Idee ist auch bei heuristischer Suche anwendbar.

Bei gleicher heuristischer Bewertung ziehe den Zustand vor, der sich näher an der Wurzel des Graphen befindet.

- Dieser Knoten hat eine größere Wahrscheinlichkeit, auf dem *kürzesten Weg* zum Ziel zu liegen.
- Jedem Zustand wird ein Tiefenzähler zugewiesen, der für jede Ebene um eins hochgezählt wird.
- Bewertungsfunktion f ist die Summe aus zwei Komponenten.
  - $g(n)$: Tatsächliche Länge des Weges vom Start zum Zustand n.
  - $h(n)$: Schätzung der Entfernung von n zu einem Ziel.

Die Komponente $g(n)$ verleiht der Suche eine stärkere Tendenz zur Breitensuche.

- Dies verhindert, dass die Suche von einer fehlerhaften Bewertung in die Irre geführt wird.
- Der Algorithmus verliert sich nicht dauerhaft in einem unendlichen Zweig, falls eine Heuristik kontinuierlich gute Bewertungen abgibt.
- Jeder Zustand muss darauf untersucht werden, ob er schon mal aufgetreten ist, um Schleifen zu verhindern. $\rightarrow$ Closed-List

Wird ein bereits generierter Zustand n auf einem kürzeren Weg noch einmal erreicht, muss die Bewertung angepasst werden.

- Ist Zustand n in der Open-List enthalten, dann muss der neue Wert $g(n) + h(n)$ in der Priority-Queue gesetzt werden.
- Wurde Zustand n bereits abgearbeitet und in die Closed-List eingefügt, dann muss n der Closed-List entnommen und mit neuem Wert wieder in die Agenda aufgenommen werden.

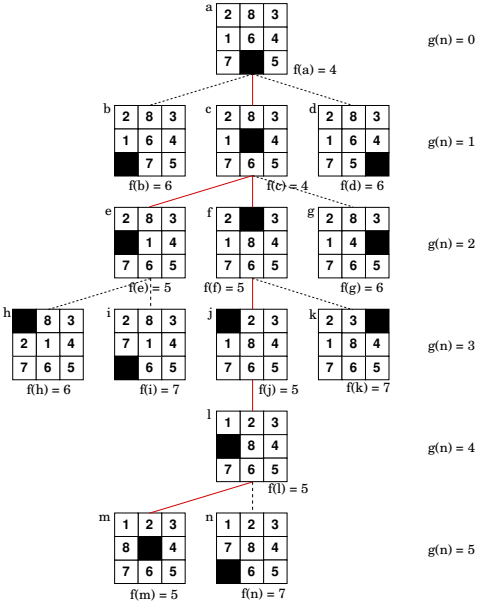
$A^\star$-Algorithmus

```
// global data structures
PriorityQueue<Node *> openList;
HashMap<Node *, double> g;
HashSet<Node *> closedList;

Node * AStarSearch(Node *pRoot) {
    g[pRoot] := 0;
    openList.insert(pRoot, h(pRoot));
    while (!openList.isEmpty()) {
        Node *pCurr = openList.extractMin();
        if (zielknoten(pCurr))
            return pCurr;

        expandNode(pCurr);
        closedList.insert(pCurr);
    }
    return null;
}
```

```
void expandNode(Node *pCurr) {
    for (Node *n : nachfolger(pCurr)) {
        double c := g[pCurr] + cost(pCurr, n);
        if (openList.contains(n)) {
            if (c < g[n]) // shorter path found
                updateOpenList(n, c);
            else ;  // do nothing
        } else {
            // n not contained in open list
            if (closedList.contains(n)) {
                if (c < g[n]) // shorter path found
                    updateClosedList(n, c);
                else ;  // do nothing
            } else
                // first visit of n
                insertOpenList(n, c);
        }
    }
}
```

```
void updateOpenList(Node *n, double c) {
    g[n] := c;
    openList.decreaseKey(n, c + h(n));
}

void updateClosedList(Node *n, double c) {
    g[n] := c;
    openList.insert(n, c + h(n));
    closedList.erase(n);
}

void insertOpenList(Node *n, double c) {
    g[n] := c;
    openList.insert(n, c + h(n));
}
```

Schiebepuzzle



Sei $p = (s_0, s_1, \ldots, s_r)$ eine Folge von Zuständen und s_{i+1} sei durch Anwendung eines Zustandsübergangsoperators auf s_i erreichbar.

Wenn beim Übergang von s_i nach s_{i+1} Kosten in Höhe von $k(s_i, s_{i+1})$ anfallen, dann seien die Kosten $k(p)$ der Zustandsfolge definiert durch:

$$k(p) := \sum_{i=0}^{r-1} k(s_i, s_{i+1})$$

Für einen Zustand s bezeichne

- $g^\star(s)$ die minimalen Kosten vom Start nach s:

$$g^\star(s) := \inf\{k(p) \mid p \text{ ist Weg vom Startzustand nach } s\}$$

- $h^\star(s)$ die minimalen Kosten von s zum Ziel:

$$h^\star(s) := \inf\{k(p) \mid p \text{ ist Weg von } s \text{ zu einem Zielzustand}\}$$

Betrachtet man also einen Zustand s, der auf einem Suchpfad vom Startzustand zum Zielzustand liegt, dann hat der kürzeste Weg, der über den Zustand s läuft, die Länge $g^\star(s) + h^\star(s)$:

$$\text{Start} \xrightarrow{g^\star(s)} s \xrightarrow{h^\star(s)} \text{Ziel}$$

Der Wert $g^\star(n)$ ist eine untere Schranke für den Wert $g[n]$ aus dem $A^\star$-Algorithmus, es gilt also $g[n] \geq g^\star(n)$. Da in der Regel ein Knoten auf verschiedenen Wegen erreicht werden kann, gilt die Gleichheit $g[n] = g^\star(n)$ oft erst nach mehreren Runden.

Problem: Finde eine Zustandsfolge p' vom Startzustand s_0 in einen Zielzustand z, die minimale Kosten aufweist, d.h.:

$$\begin{aligned} k(p') &= h^\star(s_0) \text{ bzw.} \\ k(p') &= \inf\{g^\star(z) \mid z \text{ ist Zielzustand}\} \end{aligned}$$

Zulässige Schätzer

Eine heuristische Funktion h heißt *zulässiger Schätzer* bzw. *zulässig*, falls $h(s) \leq h^\star(s)$ für alle Zustände s des Zustandsraums gilt. Mit anderen Worten: Ein zulässiger Schätzer darf die echten Kosten nicht überschätzen!

Zulässige Schätzer sind:

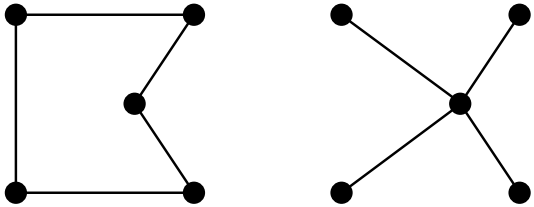
- Die heuristischen Funktionen für das Schiebepuzzle.
- Die Luftlinienentfernung beim Routenproblem.

Bei kombinatorischen Optimierungsproblemen werden als zulässige Schätzer häufig effizient lösbare Relaxationen des Problems verwendet: Vernachlässige Nebenbedingungen, so dass sich ein einfacher zu lösendes Problem ergibt.

Beispiel: Minimaler Spannbaum als Relaxation für die Berechnung eines minimalen Hamiltonschen Weges.

Well-known example: Travelling salesperson problem (TSP).

Find the shortest tour visiting all cities exactly once.



Minimum spanning tree (MST) can be computed in $O(n^2)$ and is a lower bound on the shortest (open) tour.

Frage: Was hat MST mit TSP zu tun?

Antwort: Lässt man auf einer Rundreise eine Kante weg, erhält man einen Spannbaum. Die Kosten eines MST sind also höchstens so groß wie die Kosten einer minimalen Rundreise.

Relaxationen entstehen oft durch Weglassen oder Lockern von Bedingungen.

Beispiel: 8-Puzzle

- Ein Stein kann direkt überall plaziert werden, auch auf bereits belegten Feldern.
  $\rightarrow$ h_1 ergibt exakte Anzahl Schritte der optimalen Lösung.
- Ein Stein kann in alle Richtungen bewegt werden, auch auf ein schon belegtes Feld.
  $\rightarrow$ h_2 ergibt exakte Anzahl Schritte der optimalen Lösung.

Achtung: Die Kosten einer optimalen Lösung eines relaxierten Problems sind *zulässige Schätzer* des originalen Problems!

Eine optimale Lösung des Originalproblems ist auch eine Lösung des relaxierten Problems, und ist mindestens so teuer wie die optimale Lösung des relaxierten Problems.

$A^\star$-Algorithmus

- Durch eine Verringerung von $g(s)$ für einen Zustand s kann auch eine Verringerung von $f(s)$ auftreten.
- Dies kann auch für schon expandierte Knoten der Fall sein.
- Sind Bewertungen von bereits expandierten Knoten anzupassen, *werden diese aus der Closed-List entfernt und wieder in die Agenda aufgenommen.*
- Dadurch können sie später erneut betrachtet werden.

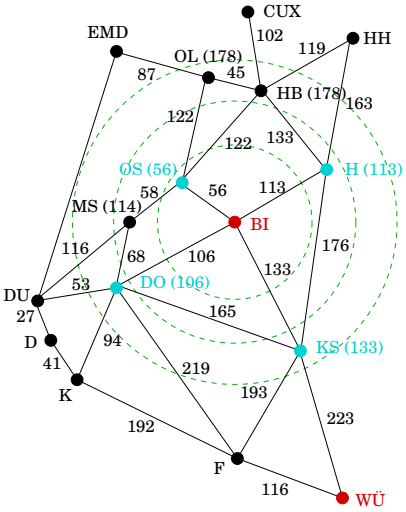
Monotone Schätzer

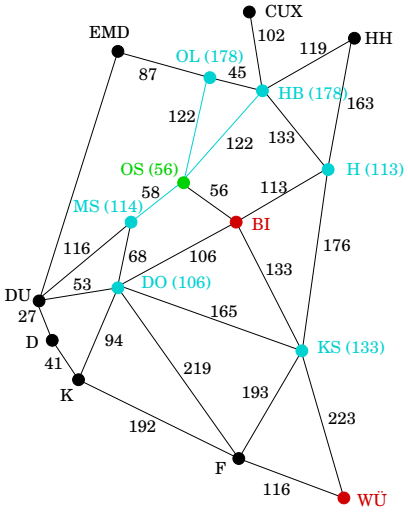
- Zulässiger Schätzer, der die Dreiecksungleichung erfüllt, d.h. für alle Zustände s und alle Nachfolger s' von s gilt: $h(s) \leq k(s, s') + h(s')$
- *Bei Einsatz von monotonen Schätzern werden expandierte Knoten nie wieder betrachtet.*

Vereinfachung des A^*-Algorithmus für monotone Heuristik:

```
Node * AStarSearch(Node *pRoot) {
    HashMap<Node *, double> g;
    PriorityQueue<Node *> agenda;
    g[pRoot] := 0;
    agenda.insert(pRoot, h(pRoot));
    while (!agenda.isEmpty()) {
        Node *pCurr = agenda.extractMin();
        if (zielknoten(pCurr))
            return pCurr;

        for (Node *n : nachfolger(pCurr)) {
            g[n] := g[pCurr] + cost(pCurr, n);
            agenda.insert(n, g[n] + h(n));
        }
    }
    return null;
}
```

Dijkstra: Suche von Bielefeld nach Würzburg

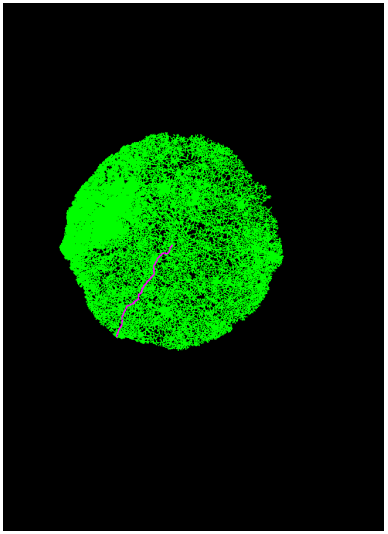




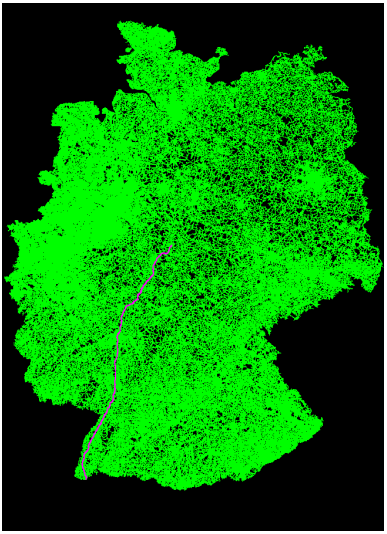
Die Suche breitet sich in alle Richtungen gleichmäßig aus, also kreisförmig um den Startknoten, bis das Ziel gefunden wird.

Dijkstra: kreisförmige Ausbreitung der Suche

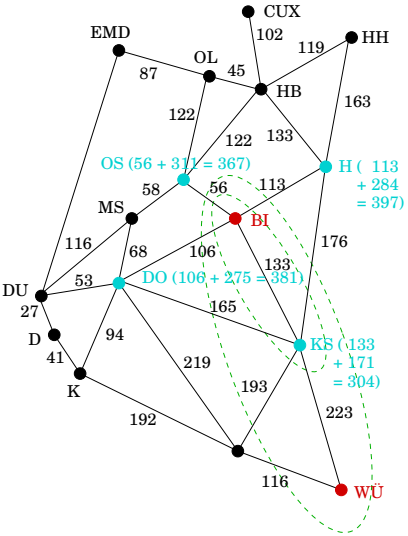
von Göttingen nach Frankfurt

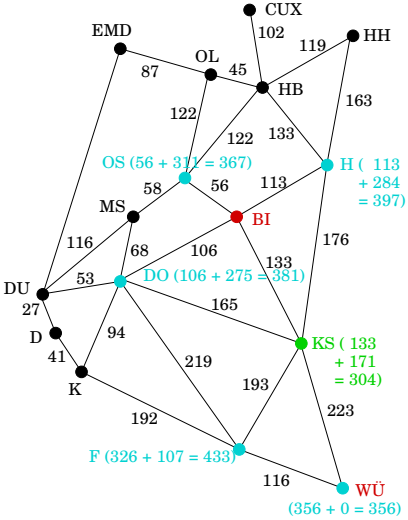


von Göttingen nach Freiburg i.B.



$A^\star$: Suche Weg von Bielefeld nach Würzburg

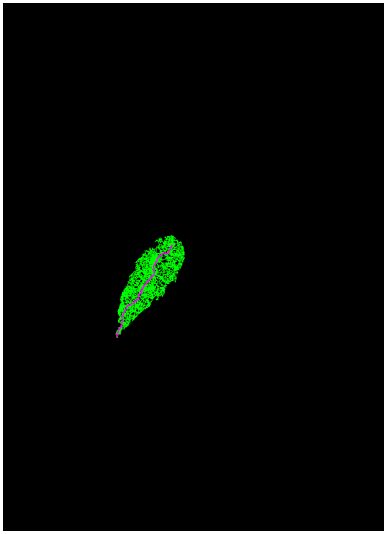




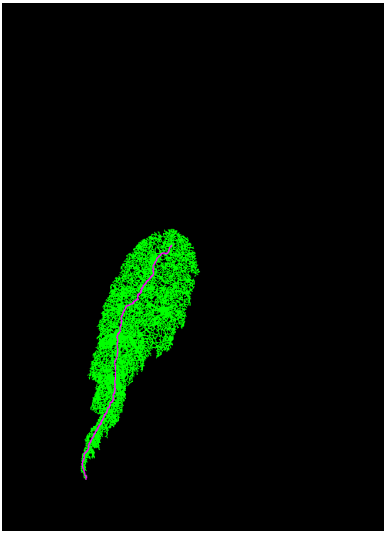
Durch die Schätzfunktion findet die Suche „zielgerichtet“ statt, nicht in zentrischen Kreisen, sondern in Konturen.

$A^\star$: zielgerichtete Ausbreitung der Suche

von Göttingen nach Frankfurt

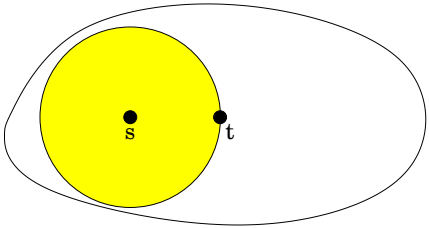


von Göttingen nach Freiburg i.B.

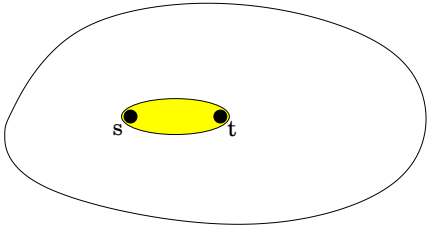


Durch die Schätzfunktion findet die Suche „zielgerichtet“ statt, nicht in zentrischen Kreisen, sondern in Konturen.

Dijkstra:



$A^\star$:



Beschleunigung: umso größer, je weiter s und t auseinander liegen

Aus der Vorlesung „Effiziente Algorithmen“ wissen wir, dass der Algorithmus von Dijkstra korrekt ist, solange keine Kante e mit negativer Bewertung $cost(e)$ im Graphen enthalten ist.

```
d[s] := 0
Q.clear()
Q.add(s, 0)
while not Q.isEmpty() do
    u := Q.extractMin()
    for each e = (u, v) ∈ E do
        if not Q.contains(v) or d[v] > d[u] + cost(e)
        then d[v] := d[u] + cost(e)
            if Q.contains(v)
            then Q.decreaseKey(v, d[v])
            else  Q.insert(v, d[v])
```

Korrektheit $A^\star$-Algorithmus

Sei h ein monotoner Schätzer, so dass $h(u) \leq cost(u, v) + h(v)$ für alle $(u, v) \in E$ gilt.

```
d[s] := 0
Q.clear()
Q.add(s, h(s))
while not Q.isEmpty() do
    u := Q.extractMin()
    for each e = (u, v) ∈ E do
        if not Q.contains(v) or d[v] > d[u] + cost(e)
        then d[v] := d[u] + cost(e)
            if Q.contains(v)
            then Q.decreaseKey(v, d[v] + h(v))
            else  Q.insert(v, d[v] + h(v))
```

Abarbeitung der Knoten:

- Dijkstra: Knoten u vor Knoten v, wenn $d[u] < d[v]$
- $A^\star$: Knoten u vor Knoten v, wenn $d[u] + h(u) < d[v] + h(v)$

Wir definieren $cost_h(u, v) := cost(u, v) - h(u) + h(v)$ und ersetzen in G die alte durch die neue Kostenfunktion und erhalten so G_h.

- Die Länge aller Pfade vom Start s zum Ziel t ändert sich um den selben Wert $h(t) - h(s)$, daher bleiben alle kürzesten Wege erhalten: $d_h(s, u) = d(s, u) - h(s) + h(u)$
- Die Reihenfolge der besuchten Knoten von Dijkstra im neuen Graphen G_h entspricht also der von $A^\star$ in G, da $h(s)$ für alle Knoten gleich ist.
- $A^\star$ ist also korrekt, wenn $cost_h(u, v) \geq 0$ für alle $(u, v) \in E$ ist. Dies folgt unmittelbar aus der Dreiecksungleichung $h(u) \leq cost(u, v) + h(v)$ des monotonen Schätzers h.

Übung 16. Betrachten Sie das Rucksackproblem:

- Gegeben: Eine Menge $G = \{g_1, \ldots, g_n\}$ von Gegenständen und ein maximales Gesamtgewicht C.
- Jeder Gegenstand $g \in G$ hat ein Gewicht $w(g)$ und liefert einen Profit $p(g)$.

Bestimme eine Teilmenge $F \subseteq G$, so dass folgendes gilt:

- Der Gesamtprofit $\sum_{g \in F} p(g)$ von F ist maximal
- unter der Nebenbedingung $\sum_{g \in F} w(g) \leq C$.

Zentrale Fragen:

- Formulierung des Rucksackproblems als Suchproblem?
- Was muss ein $A^\star$-Algorithmus für das Rucksackproblem berücksichtigen?
- Abschätzung des noch erzielbaren Nutzens?

Übung 17. Implementieren Sie den $A^\star$-Algorithmus zum Lösen des m-Puzzles mit $m = n^2 - 1$ und vergleichen Sie die Laufzeiten für $n = 3, 4, 5, 6$.

- Wie effektiv ist die heuristische Funktion h_2 im Vergleich zu h_1?
- Ist die heuristische Funktion h_2 auch für größere Puzzle effektiv?
- Recherchieren Sie im Internet, wie die Heuristik h_2 erweitert werden kann, um genauere Abschätzungen zu liefern.

Vergleichen Sie die Laufzeiten des obigen Programms mit den Laufzeiten des Integer-Programms, das wir mittels GLPK implementiert hatten.

Je genauer die Heuristik ist, desto weniger Knoten werden untersucht.

Begrenzender Faktor beim $A^\star$-Algorithmus ist der Speicherplatz, da alle bekannten Knoten im Speicher gehalten werden.

- $A^\star$-Algorithmus ist nicht geeignet, falls keine monotone Heuristik bekannt ist.
- iterative deepening: Suchtiefe begrenzen und schrittweise vergrößern, bis eine Lösung gefunden wird.

IDA*: Iterative Deepening $A^\star$

- Die erste Grenze für die Suchtiefe ist der Wert der Heuristik für den Startknoten. Da die Heuristik zulässig ist, gibt es keinen kürzeren Weg vom Start zum Ziel.
- Die rekursive Suche innerhalb der Schleife liefert entweder eine Lösung oder das Minimum der per Heuristik bestimmten Weglänge für alle Knoten, die als erste Knoten hinter der Grenze gefunden wurden. Dieses Minimum wird für den nächsten Schleifendurchlauf als Grenze für die Suchtiefe verwendet.

Iterative Deepening $A^\star$

```
HashMap<Node *, int> g;

Node * idaStar(Node *root) {
    g[root] := 0;
    int limit := h(root);

    while (true) {
        pair<Node *, int> p := search(root, limit);
        if (p.first != null)
            return p.first;
        limit := p.second;
    }
    return null;
}
```

Iterative Deepening $A^\star$

```
pair<Node *, int> search(Node *node, int limit) {
    int f := g[node] + h(node);
    if (f > limit)
        return pair<Node *, int>(null, f);

    if (zielknoten(node))
        return pair<Node *, int>(node, 0);

    int min := Integer.MAX_VALUE;
    for (Node *n : nachfolger(node)) {
        pair<Node *, int> p := search(n, limit);
        if (p.first != null)
            return p;
        if (p.second < min)
            min := p.second;
    }
    return pair<Node *, int>(null, min);
}
```

Manhattan distance enhanced by *linear conflicts*: Historically, the linear-conflict heuristic was the first significant improvement over Manhattan distance[29]. It applies to tiles in their goal row or column, but reversed relative to each other. For example, assume the top row of a given state contains the tiles (2 1) in that order, but in the goal state they appear in the order (1 2). To reverse them, one of the tiles must move out of the top row, to allow the other tile to pass by, and then move back into the top row. Since these two moves are not counted in the Manhattan distance of either tile, two moves can be added to the sum of the Manhattan distances of these two tiles without violating admissibility.

Übung 18. Implementieren Sie den *IDA**-Algorithmus am Beispiel des 15-Puzzles sowie die Linear-Conflicts-Heuristik. Vergleichen Sie die Laufzeiten mit Ihrer Implementierung des $A^\star$-Algorithmus.

[29] O. Hansson, A. Mayer, M. Yung, Criticizing solutions to relaxed models yields powerful admissible heuristics, Information Science 63 (3) (1992) 207–227.

15-Puzzle[30]: The efficiency of $A^\star$ searching depends on the quality of the lower bound estimates of the solution cost. The $A^\star$ search algorithm is of fundamental importance in artificial intelligence. Improvements to the search efficiency can take the range from general algorithm enhancements (application independent) to problem-specific heuristics (application dependent):

- General search space properties. Many search domains can be represented as directed graphs rather than as trees. The removal of duplicate nodes from the search can result in potentially large savings (Marsland and Reinefeld 1994; Taylor and Korf 1993).
- Solution databases. In many problems, the states near to the goal nodes can be precomputed by a backwards search. In two-player games, such as chess and checkers, the backwards searches are saved in *endgame databases* and are used to stop the search early, improving search efficiency and accuracy (Schaeffer et al. 1992).

[30] Joseph C. Culberson, Jonathan Schaeffer: Pattern Databases. Computational Intelligence 14(3): 318-334 (1998)

15-puzzle[31]: On larger problems, *IDA** with Manhattan distance takes too long, and more accurate heuristic functions are needed. While Manhattan distance sums the cost of solving each tile independently, we consider the costs of solving several tiles simultaneously, taking into account the interactions between them.

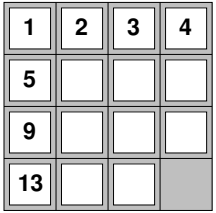
The reason the problem is difficult, and why Manhattan distance is only a lower bound on actual solution cost, is that the tiles get in each other's way. By taking into account some of these interactions, we can compute more accurate admissible heuristic functions.

Consider a subset X of the Fifteen Puzzle tiles. For a given state, the minimum number of moves needed to get the tiles of X to their goal positions, including required moves of other tiles, is a lower bound on the number of moves needed to solve the entire puzzle.

[31] Richard E. Korf, Ariel Felner: Disjoint pattern database heuristics. Artificial Intelligence 134(1-2): 9-22 (2002)

This number depends on the current positions of the tiles of X and the blank, but is independent of the positions of the other tiles. Thus, we can precompute all these values, store them in memory, and look them up as they are needed during the search.

We can compute this entire table with a single breadth-first search backward from the goal state. The unlabelled tiles are all equivalent, and a state is uniquely determined by the positions of the tiles in X and the blank. As each configuration of these tiles is encountered for the first time, the number of moves made to reach it is stored in the corresponding entry of the table, until all entries are filled. Note that this table is only computed once for a given goal state, and its cost is amortized over the solution of multiple problem instances with the same goal state.

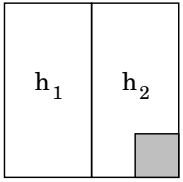


goal state

The main limitation of non-additive pattern databases is that they can't solve larger problems. *With multiple databases, the best way to combine them admissibly is to take the maximum of their values,* even if the sets of tiles are disjoint. The reason is that *non-additive pattern database values include all moves needed to solve the pattern tiles, including moves of other tiles.*

Instead of taking the maximum of different pattern database values, we would like to be able to sum their values, to get a more accurate heuristic, without violating admissibility. This is the main idea of disjoint pattern databases.

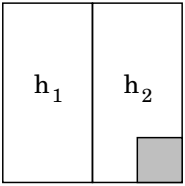
To construct a disjoint pattern database for the sliding-tile puzzles, we partition the tiles into disjoint groups, such that no tile belongs to more than one group. We then precompute tables of the *minimum number of moves of the tiles in each group that are required to get those tiles to their goal positions.* We call the set of such tables, one per group of tiles, a disjoint pattern database, or a disjoint database for short.

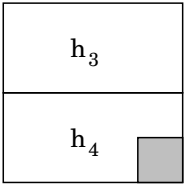


Then, given a particular state in the search, for each group of tiles, we use the positions of those tiles to compute an index into the corresponding table, retrieve the number of moves required to solve the tiles in that group, and then add together the values for each group, to compute an overall heuristic for the given state. This value will be at least as large as the Manhattan distance, and usually larger, since it accounts for interactions between tiles in the same group.

The key difference between disjoint databases and the non-additive databases described above is that *the non-additive databases include all moves required to solve the pattern tiles, including moves of tiles not in the pattern set.* As a result, given two such databases, even if there is no overlap among their tiles, we can only take the maximum of the two values as an admissible heuristic, because moves counted in one database may move tiles in the other database, and hence these moves would be counted twice. *In a disjoint database, we only count moves of the tiles in the group.*

A second difference between these two types of databases is that our disjoint databases don't consider the blank position, decreasing their size. A disjoint database contains the minimum number of moves needed to solve a group of tiles, for all possible blank positions. In neither case is the blank position part of the index to the database.





Die resultierende Heuristik h für die disjoint-database kann noch verbessert werden:

$$h := \max\{h_1(s) + h_2(s),\ h_3(s) + h_4(s)\}$$

Übung 19. Erweitern Sie Ihre Implementierung des *IDA** um eine Pattern Database. Vergleichen Sie die Laufzeiten mit Ihrer früheren Implementierung für verschiedene Puzzle-Größen. Wie viel Speicherplatz benötigen die Disjoint-Pattern-Databases?

Übersicht

Lokale Suche

Bei vielen Problemen ist nicht der Lösungsweg interessant, stattdessen ist der Zielzustand die eigentliche Lösung.

- 8-Damen-Problem: Nur die Positionen der Damen sind wichtig, nicht in welcher Reihenfolge die Damen platziert wurden.
- Entwurf Integrierter Schaltungen: Nur die letztendliche Platzierung der Bauteile und die Verdrahtung ist wichtig, nicht der Weg, wie das Ziel erreicht wurde.
- Problem des Handlungsreisenden: Nur die Tour ist wichtig.
- Stunden-/Fahrplan Erstellung: Nur der Plan ist wichtig.

Trotz Einsatz von Heuristiken ist die Laufzeit der bisher betrachteten Suchalgorithmen oft noch zu schlecht für einen praktischen Einsatz.

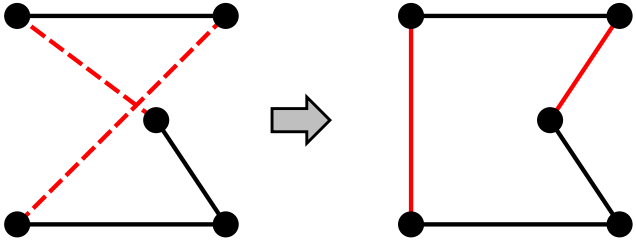
Lokale Suchalgorithmen

- arbeiten mit nur einem einzigen aktuellen Zustand anstelle von mehreren Pfaden.
- versuchen iterativ den aktuellen Zustand zu verbessern.
- besuchen in der Regel nur benachbarte Zustände. Im Gegensatz zu Breiten-, Tiefen- oder Bestensuche, wo die Suche mit dem nächsten Zustand der Agenda fortgesetzt wird.
- speichern die Pfade zur Problemlösung nicht ab.

Vorteile:

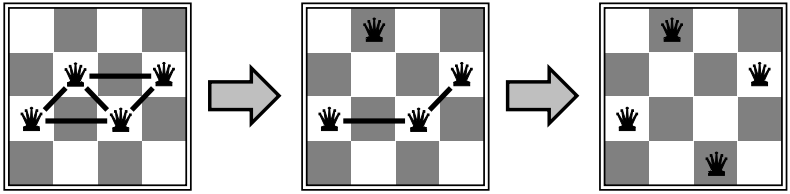
- Sehr geringer Speicherverbrauch.
- Oft werden vernünftige Lösungen in riesigen Suchräumen gefunden.

traveling salesperson problem TSP: start with any complete tour, perform pairwise exchanges



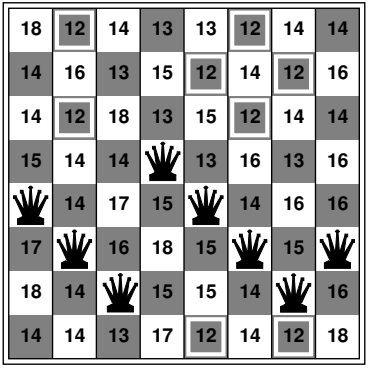
variants of this approach get within 1% of optimal very quickly with thousands of cities

n-queens problem: move a queen to reduce number of conflicts



almost always solves n-queens problems almost instantaneously for very large n, e.g., $n = 1$ million

number of pairs of queens that are attacking each other, either directly or indirectly



choose randomly among the set of best successors, if there is more than one

Hill Climbing Search

sometimes called *greedy local search*, because it grabs a good neighbor state without thinking ahead about where to go next

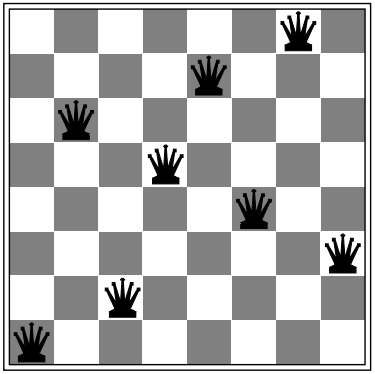
```
function HILL-CLIMBING( problem) returns a state that is a local maximum
    inputs: problem, a problem
    local variables: current, a node
                     neighbor, a node

    current ← MAKE-NODE(INITIAL-STATE[problem])
    neighbor ← a highest-valued successor of current
    while VALUE[neighbor] ≤ VALUE[current] do
        current ← neighbor
        neighbor ← a highest-valued successor of current
    end
    return STATE[current]
```

steepest ascent version

"it's like climbing Everest in thick fog with amnesia"

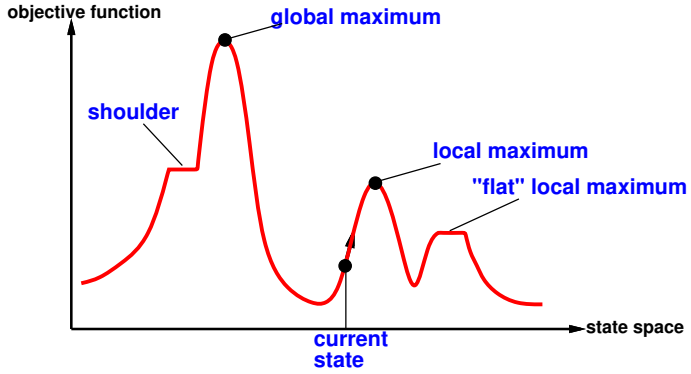
Hill Climbing Search

search may be lost in local minimum



each move of a queen would increase the number of conflicts

useful to consider state space landscape



NP-complete problems typically have an exponential number of local maxima to get stuck on

Variants of Hill Climbing

- *random sideways moves*
  - ☺ escape from shoulders → comes at a cost of higher running time
  - ☹ loop on flat maxima → put a limit on the number of consecutive sideways moves allowed
- *stochastic* choose at random from among the uphill moves
  - ☺ in some landscapes it finds better solutions than steepest ascent
  - ☹ usually converges more slowly
- *first-choice* implements stochastic hill climbing by generating successors randomly until one is generated that is better than the current state
  good strategy when a state has many of successors
- *random-restart* a reasonably good local maximum can often be found after a small number of restarts
  overcomes local maxima — trivially complete

Local Beam Search

Idea: keep k states instead of 1; choose top k of all their successors

- Not the same as k searches run in parallel! Searches that find good states recruit other searches to join them.
- useful information is passed among the k parallel search threads
- Problem: quite often, all k states end up on same local hill

Idea: choose k successors randomly, biased towards good ones

$\rightarrow$ stochastic beam search

- observe the close analogy to natural selection

Idee der Tabu-Suche:

- Um lokale Optima verlassen zu können, wird in jedem Schritt die beste erlaubte Nachbarlösung gewählt, auch wenn diese schlechter ist als die aktuelle. → Best-Improvement-Methode
- Nutze eine Gedächtnisstruktur, um die Suche zu steuern.

Um Zyklen beim Traversieren des Suchraums zu vermeiden, werden bereits besuchte Knoten in einer Liste gespeichert.

- Die in der Liste enthaltenen Knoten dürfen in den nächsten Schritten nicht ausgewählt werden.
- Verbiete den Knoten so lange, bis das Wiedererreichen dieser Lösung unwahrscheinlich ist. → Kurzzeitgedächtnis
- Probleme, die experimentell gelöst werden müssen:
  - Wie groß soll die Liste sein?
  - Nach wie vielen Schritten soll ein Eintrag entfernt werden?

Tabu-Suche

Großer Speicherbedarf, wenn ganze Knoten in der Tabu-Liste gespeichert werden. Daher werden oft nur Attribute verboten:

- Traveling Salesperson: verbiete den Tausch zweier Städte
- Färbung von Graphen: verbiete Farbzuweisung i an Knoten k

Ein Langzeitgedächtnis kann das Verfahren weiter verbessern:

- Mit der Information, wie oft Lösungen untersucht wurden, kann die Suche so gesteuert werden, das bisher wenig betrachtete Bereiche des Lösungsraums untersucht werden.
- Problem: Speicherbedarf wird noch größer.

Simulated Annealing Search

Idea: escape local maxima by allowing some "bad" moves but gradually decrease their size and frequency

- devised by Metropolis et al., 1953, for physical process modelling
- widely used in VLSI layout, airline scheduling, etc.

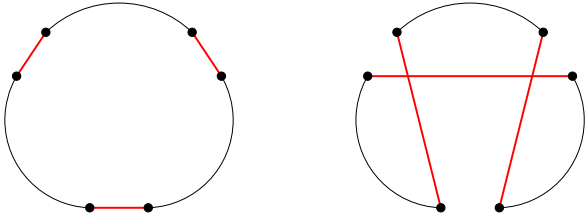
Simulated Annealing Search

```
function SIMULATED-ANNEALING( problem, schedule) returns a solution state
    inputs: problem, a problem
            schedule, a mapping from time to "temperature"
    local variables: current, a node
                     next, a node
                     T, a "temperature" controlling prob. of downward steps

    current ← MAKE-NODE(INITIAL-STATE[problem])
    for t ← 1 to ∞ do
        T ← schedule[t]
        if T = 0 then
            return current
        next ← a randomly selected successor of current
        ΔE ← VALUE[next] – VALUE[current]
        if ΔE > 0 then
            current ← next
        else current ← next only with probability e^(ΔE/T)
    end
```

Übung 20. Solve instances of n-queens and TSP by hill climbing, tabu search, local beam search and simulated annealing.

Measure the search cost and the percentage of solved problems and graph these against the optimal solution cost. Comment on your results.

TSPLIB is a library of sample instances for the TSP. Instances of symmetric traveling salesman problem are available. Use 2-opt and 3-opt on these instances and compare the results.

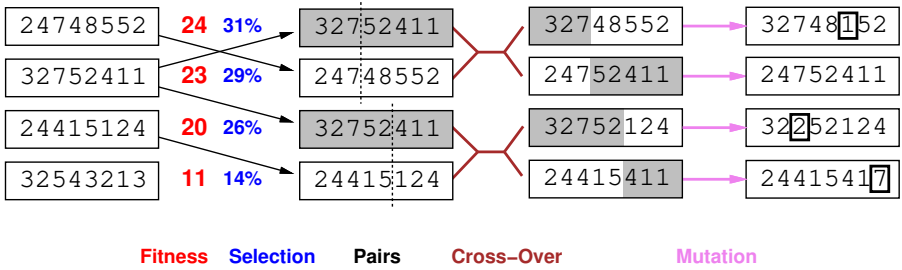


stochastic beam search + generate successors from pairs of states

- GAs begin with a set of k randomly generated states, called the *population*
- each state, or *individual*, is represented as a string over a finite alphabet
- each state is rated by the evaluation function or *fitness function*
- a fitness function should return higher values for better states

  for the 8-queens problem we use the number of nonattacking pairs of queens, which has a value of 28 for a solution
- in the following example the probability of being chosen for reproducing is directly proportional to the fitness score

8-queens problem



- a random choice of two pairs is selected for reproduction, in accordance with the shown probabilities
- a variant of this selection rule, called *culling*, in which all individuals below a given threshold are discarded, can be shown to converge faster than the random version
- for each pair to be mated, a *crossover* point is randomly chosen from the positions in the string

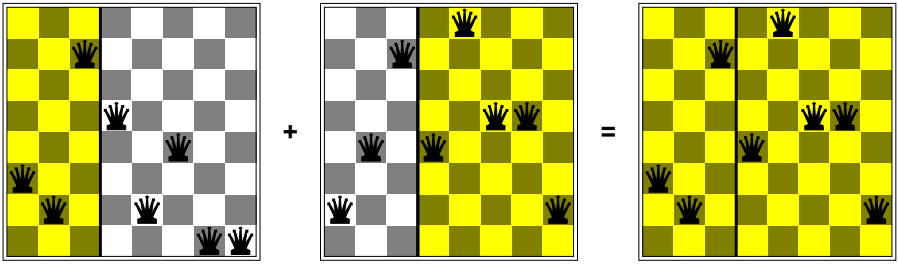
- finally, each location is subject to random mutation with a small independent probability

  in the 8-queens problem, this corresponds to choosing a queen at random and moving it to a random square in its column
- primary advantage, if any, of genetic algorithms comes from the crossover operation: combine large blocks of letters that have evolved independently to perform useful functions, thus raising the level of granularity at which the search operates

GAs require states encoded as strings (GPs use programs)

- an 8-queens state must specify the positions of 8 queens, each in a column of 8 squares, and so requires $8 \times \log_2(8) = 24$ bits
- alternatively, the state could be represented as 8 digits, each in the range from 1 to 8
- behave the two encodings differently?

Crossover helps iff substrings are meaningful components



- *schema:* a substring in which some of the positions can be left unspecified, for example, 246*****
- strings that match the schema such as 24613578 are called *instances* of the schema
- if the average fitness of the instances of a schema is above the mean, than the number of instances of the schema within the population will grow over time

- successful use of GAs require careful engineering of the representation
- at present, it is not clear whether the appeal of GAs arises from their performance or from their pleasing origins in the theory of evolution

Übersicht

Was fasziniert uns an Spielen?

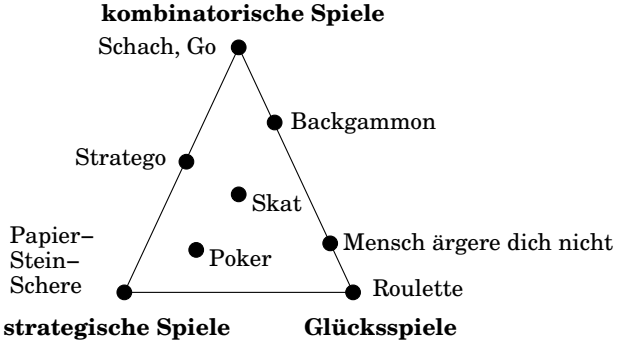
- Vor allem der ungewisse Ausgang des Spiels.

Diese Ungewissheit basiert auf

- *Zufall:* Tritt z.B. durch Würfeln (Backgammon) oder Mischen von Spielkarten (Skat) auf.
- *Kombinatorik:* Spielregeln legen die Handlungsmöglichkeiten der Spieler fest, z.B. bei Schach, Go oder Vier gewinnt.
- *mangelnder Information:* Da man z.B. nicht die Karten der Mitspieler oder deren Absichten kennt, erhält man nur eine Teilkenntnis des erreichten Spielstandes.

Arten von Spielen

- *Glücksspiel:* Der Einfluss des Zufalls dominiert gegenüber denen der Spieler.
- *Kombinatorisches Spiel:* Ungewissheit beruht auf den vielfältigen, praktisch unüberschaubaren Zugmöglichkeiten.
- *Strategisches Spiel:* Ungewisser Ausgang durch imperfekte Information.



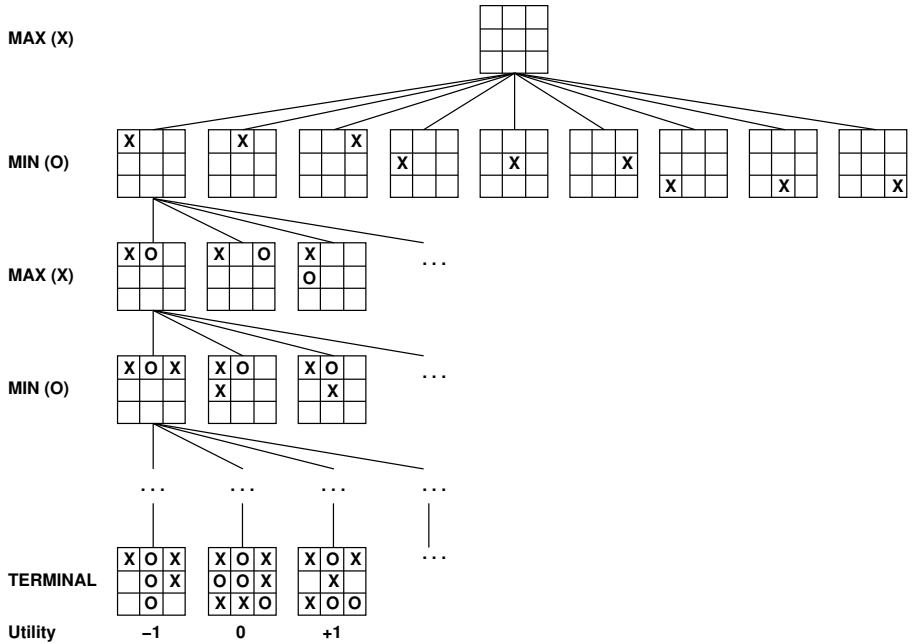
- *Zug:* Spielabschnitt, der genau eine durch die Spielregeln festgelegte Handlungsmöglichkeit eines Spielers umfasst. Wird manchmal auch Halbzug genannt.
- *Zugzwang-Phänomen:* Wenn ein Spieler nicht passen kann, verschlechtert er evtl. seine Situation durch jeden ihm möglichen Zug.
- *Kooperative Züge* stärken beide Spieler gleichzeitig (selten).
- *Nullsummenspiel:* Normalerweise fördert ein Zug den einen Spieler genauso stark, wie er den anderen behindert – die Summe der Nutzeneffekte beider Spieler ist immer gleich Null.

Kombinatorische Spiele als Suchprobleme

- Es gibt genau zwei Spieler: Max und Min.
- Max hat immer den ersten Zug.
- Es ist eine Anfangsstellung des Spiels definiert.
- Jeder Spieler besitzt die vollständige Information über die Zugmöglichkeiten des Gegners.
- Die Spieler ziehen abwechselnd, es ist kein Aussetzen möglich.
- Nach endlich vielen Zügen wird stets eine Spielstellung ohne Fortsetzungsmöglichkeiten erreicht.
- Gesucht ist eine Gewinn-Strategie für Max.
- Eine Nutzenfunktion bewertet die Endzustände und damit den Ausgang des Spiels numerisch.

win	+1	Max hat gewonnen
draw	0	unentschieden
loss	−1	Min hat gewonnen

Suchbaum für Tic-Tac-Toe

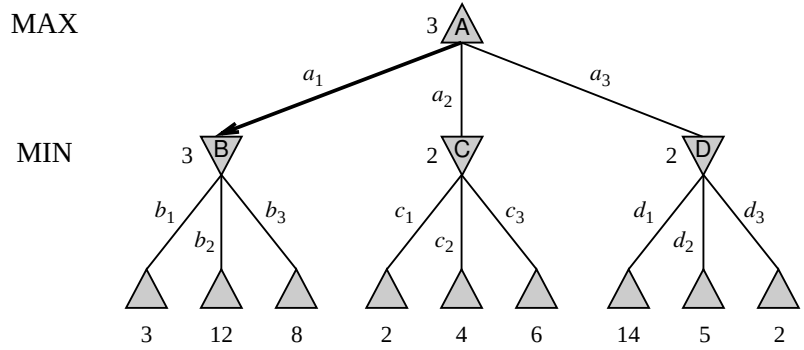


Rational agierende Spieler

- denken Zug für Zug voraus, abwechselnd einen eigenen und einen des Gegners.
- weisen den erreichbaren Positionen Werte zu: Gewinn für mich, unentschieden oder Gewinn für den Gegner.
- versuchen in jedem Zug ihren Vorteil zu maximieren, während der Gegner diesen minimieren will.

Welchen Zug muss Max machen, um zu gewinnen?

- Generiere den vollständigen Suchbaum für das Spiel.
- Wende die Nutzenfunktion auf jeden Endzustand an.
- Weise den Knoten im Suchbaum wie folgt Werte zu:
  - Ein Knoten von Max erhält als Wert das Maximum der Kinder.
  - Ein Knoten von Min erhält als Wert das Minimum der Kinder.
- Max wählt einen Zug passend zum Wert des Wurzelknotens



Minimax-Algorithmus

```
function MINIMAX-DECISION(state) returns an action
    inputs: state, current state in game

    v ← MAX-VALUE(state)
    return the action in SUCCESSORS(state) with value v
```

```
function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state)
        then return UTILITY(state)
    v ← −∞
    foreach s in SUCCESSORS(state) do
        v ← MAX(v, MIN-VALUE(s))
    return v
```

```
function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state)
        then return UTILITY(state)
    v ← ∞
    foreach s in SUCCESSORS(state) do
        v ← MIN(v, MAX-VALUE(s))
    return v
```

Beispiel: Nimm-Spiel

- Das Spiel beginnt mit einem Haufen von n Spielmarken.
- Die Spieler ziehen abwechselnd.
- Bei jedem Zug muss ein Spieler ein Häufchen Spielmarken in nicht-leere unterschiedlich große Häufchen teilen.
- Der erste Spieler, der nicht mehr ziehen kann, verliert.

Spielbaum? Minimax-Verfahren?

Eine erschöpfende Suche bis zu den Endknoten ist in der Regel nicht möglich:

- Time complexity: $O(b^m)$
- Space complexity: $O(b \cdot m)$ (depth-first exploration)
- for chess, $b \approx 35$, $m \approx 100$ for "reasonable" games

$\rightarrow$ exact solution completely infeasible

Minimax mit fester Tiefe

- Durchsuche den Zustandsraum nur bis zu einer vordefinierten Anzahl n von Ebenen.
- Um den Wert von n festzulegen, ist der Ressourcenverbrauch für Zeit und Platz zu berücksichtigen.
- Die zu bewertenden Zustände sind dann aber in der Regel keine Endzustände.

  Daher ist eine heuristische Bewertungsfunktion anzuwenden.
- Der Wert am Wurzelknoten zeigt nicht mehr an, ob das Spiel gewonnen wird!

  Es handelt sich nur um den Wert des am höchsten bewerteten Zustandes, der in n Zügen vom Startknoten aus mit Sicherheit erreichbar ist.
- Diese Strategie heißt Vorausschau über n Züge.

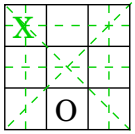
Eine heuristische Bewertungsfunktion enthält Wissen über das Spiel, um eine Stellung zu bewerten.

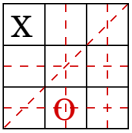
Beispiel: Heuristische Bewertungsfunktion für Tic-Tac-Toe.

$$Eval(z) = W(z) - L(z)$$

mit

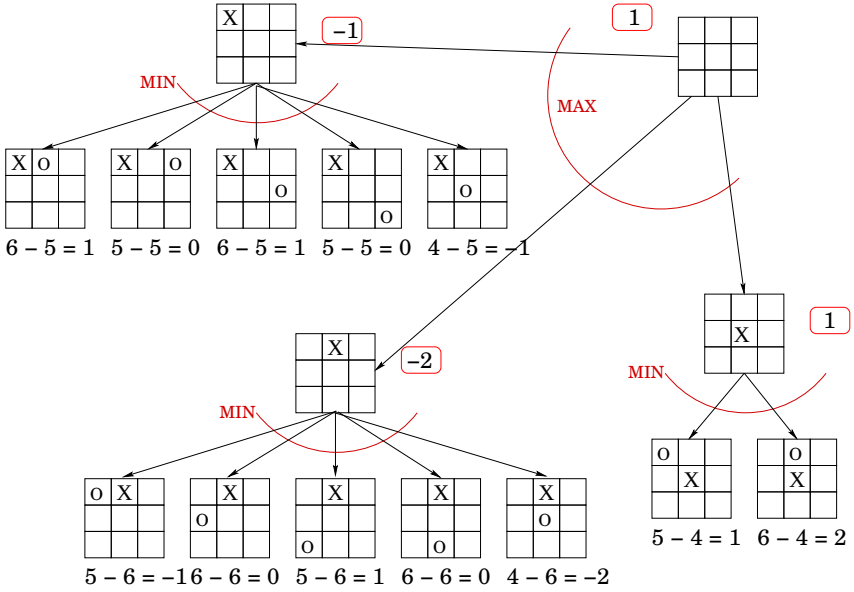
- $W(z)$: Anzahl eigener Gewinnmöglichkeiten (Win).
- $L(z)$: Anzahl gegnerischer Gewinnmöglichkeiten (Loss).
- $Eval(z)$: Bewertung des Zustandes.





- X hat 6 mögliche Gewinnwege
- O hat 5 mögliche Gewinnwege

Heuristische Bewertungsfunktion



Eine Bewertungsfunktion kann auch aus n einzelnen Merkmalen bestehen, die durch eine gewichtete Summe aggregiert werden:

$$Eval(z) = w_1 \cdot f_1(z) + w_2 \cdot f_2(z) + \ldots + w_n \cdot f_n(z)$$

Die Koeffizienten $w_1, \ldots, w_n$ legen fest, welche Eigenschaften wichtiger sind als andere.

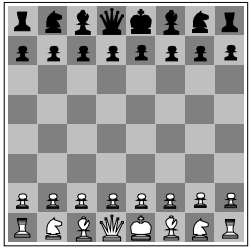
Beispiel Schach:

- $f_1(z)$ Vorteile und Position einer Figur.
- $f_2(s)$ Vorteile und Position einer anderen Figur.
- $f_3(s)$ Kontrolle über das Spielbrett usw.

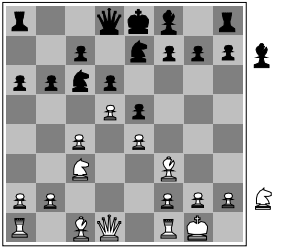
Ist der Vorteil einer Dame größer als bspw. die Kontrolle über das Spielbrett, so wird der entsprechende Koeffizient erhöht.

Führt das Bewertungspolynom zu einer schlechten Zugfolge, müssen die Koeffizienten angepasst werden. $\rightarrow$ Lernen

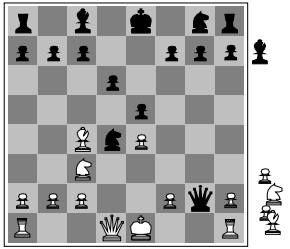
Heuristische Bewertungsfunktion



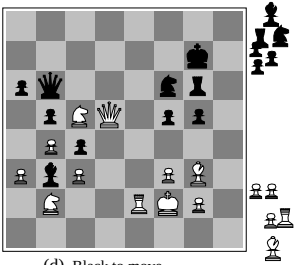
(a) White to move
Fairly even



(b) Black to move
White slightly better



(c) White to move
Black winning



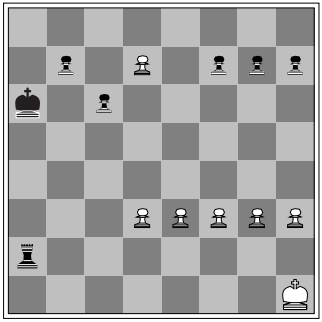
(d) Black to move
White about to lose

Horizont-Effekt

Ein Suchalgorithmus kann in gewissen Situationen zu kurzsichtig sein.

Hier: Materialvorteil für Schwarz, aber Weiß erhält eine Dame!

Besonders interessant, falls der Gegner einen für sich ungünstigen Spielzustand in wenigen Zügen jenseits der Suchtiefe in einen guten Zustand bringen kann.



Black to move

Bei begrenzter Suchtiefe kann auf der letzten Ebene das Schlagen eines gedeckten Bauern durch eine Dame günstig erscheinen, da es einen Materialvorteil bringt.

Lösung: Shannon definiert Ruhesuche, die noch heute wichtiger Bestandteil jedes Schachprogramms ist.

Ruhesuche

Zur Abschwächung des Horizontproblems wird die Ruhesuche eingesetzt.

- Eine *ruhige Position* ist dann erreicht, wenn der Gegner mit dem nächsten Zug nur eine geringe Änderung des Schätzwertes erzielen kann.

Modifiziere die Minimax-Suche mit beschränkter Tiefe:

- Relevante Blätter im Suchbaum werden durch eine Ruhesuche weitergehend analysiert.
- Zu dem so erweiterten Suchbaum wird mittels Minimax der optimale Zug bestimmt.
- Einen generellen Lösungsansatz zur Eliminierung des Horizonteffektes gibt es bisher nicht.

Heuristische Bewertungsfunktion

Ist es wünschenswert, möglichst viele Züge voraus zu schauen?

- Die Bewertungen, die sehr tief im Baum stattfinden, können durch die bloße Tiefe negativ beeinflusst werden.
- Eine Schätzung von Minimax (das wünschen wir uns) ist etwas anderes als das Minimax von Schätzungen (das tun wir).

→ Eine tiefere Suche mit Bewertung und Minimax bedeutet nicht unbedingt, dass dies die bessere Suche ist.

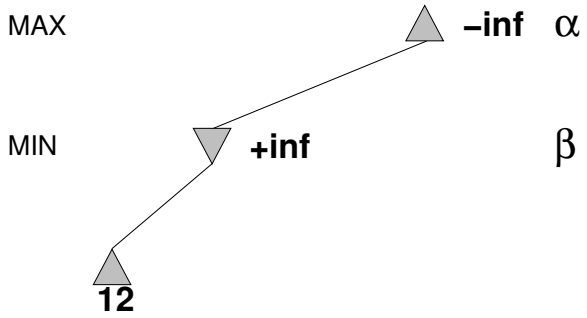
Trotzdem wollen wir versuchen, mittels einer Branch-and-Bound- Technik die Suchtiefe zu steigern. Teilbäume, die kein besseres Ergebnis als das bisher beste liefern können, werden abgeschnitten.

Alpha-Beta-Suche

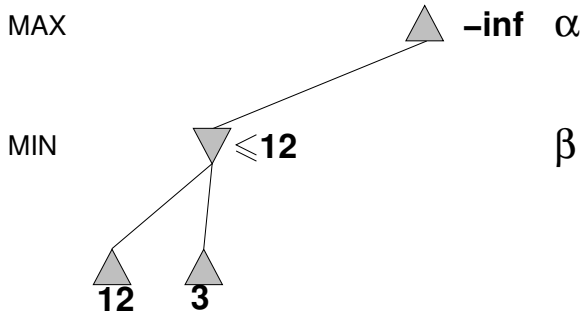
- Für die Berechnung der Minimax-Bewertung irrelevante Zweige werden nicht weiter untersucht.
- Wir benötigen zunächst die Bewertungen der Zustände in der maximalen Suchtiefe, also setzen wir Tiefensuche ein.
- Lege vorläufige Bewertungen fest:
  - Alpha-Werte sind Werte an Max-Knoten.
    - $\rightarrow$ Alpha-Werte können niemals kleiner werden!
  - Beta-Werte sind Werte an Min-Knoten.
    - $\rightarrow$ Beta-Werte können niemals größer werden!
- Diese Werte steuern das vorzeitige Beenden der Suche in einem Teilbaum.
- Wir können die Suche unterhalb eines Max-Knotens s vorzeitig beenden, wenn gilt:

$$\exists \text{Min-Vorgänger } s' \text{ von } s : \alpha(s) \geq \beta(s')$$

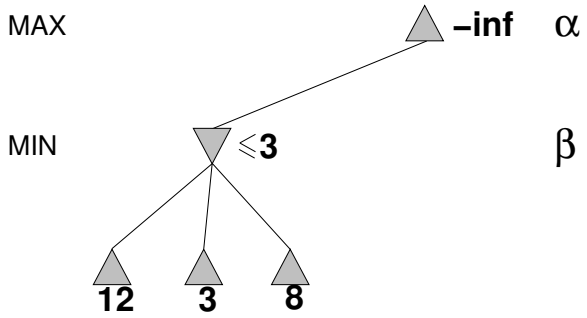
- analoge Definition für Min-Knoten (siehe nächste Folien)



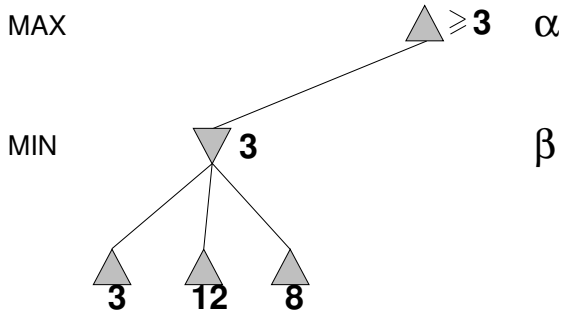
β-Werte können nicht größer werden!



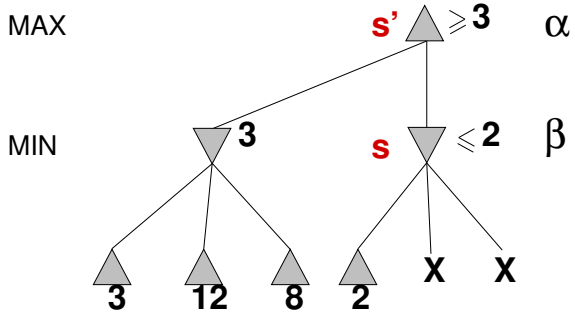
β-Werte können nicht größer werden!



β-Werte können nicht größer werden!



α-Werte können nicht kleiner werden!

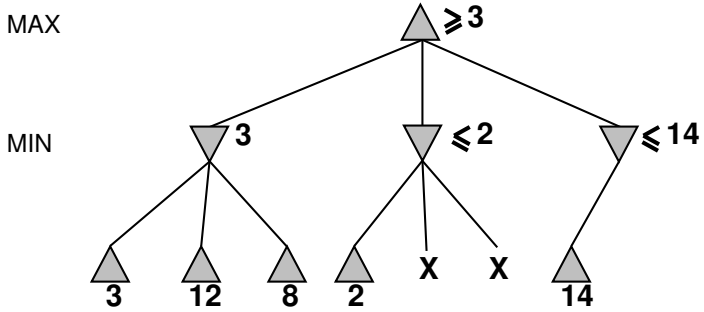


Suche unterhalb des Min-Knotens s vorzeitig beenden, wenn gilt:

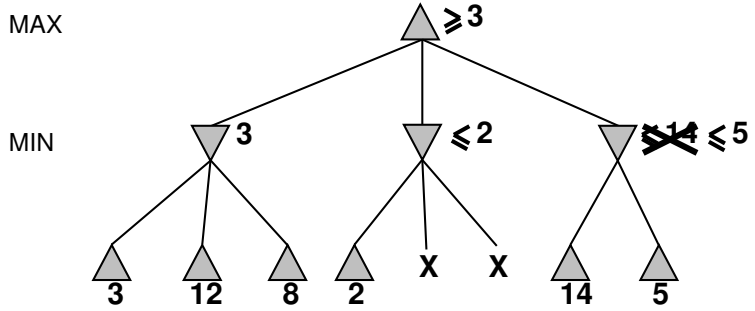
$$\exists \text{Max-Vorgänger } s' \text{ von } s : \beta(s) \leq \alpha(s')$$

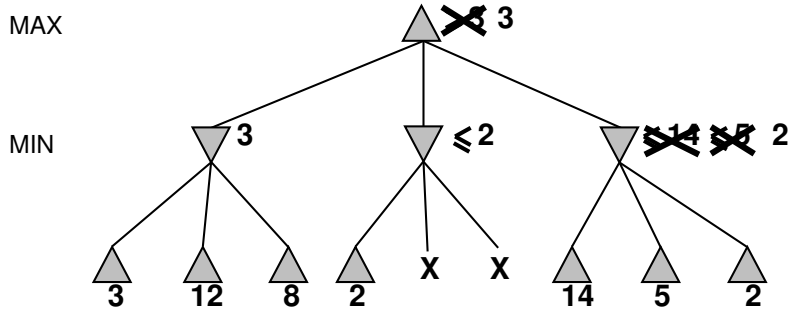
Im Zustand s' wird ein Maximum gebildet, das bereits jetzt größer als der Wert $\beta(s)$ ist. Da $\beta(s)$ nicht größer als 2 werden kann, wird die Bearbeitung des Teilbaums unter s abgebrochen.

Alpha-Beta-Suche



Alpha-Beta-Suche





Offensichtlich spielt die Reihenfolge, in der Kindknoten bearbeitet werden, eine wesentliche Rolle. Daher ist es wichtig, zuerst die Züge zu betrachten, die das beste Ergebnis versprechen.

Setze Heuristik ein: Sortiere beim Schach die Züge danach, welche Figur geschlagen wird, oder welche Figur schlägt. „Turm schlägt Dame“ vor „Bauer schlägt Turm“ vor „Turm schlägt Turm“.

Alpha-Beta-Suche

function ALPHA-BETA-DECISION(*state*) returns an action
  inputs: *state*, current state in game

  $v \leftarrow$ MAX-VALUE(*state*, $-\infty, +\infty$)
  return the *action* in SUCCESSORS(*state*) with value *v*

---

function MAX-VALUE(*state*, α, β) returns *a utility value*
  inputs: *state*, current state in game
    α, the value of the best alternative for MAX along the path to *state*
    β, the value of the best alternative for MIN along the path to *state*

  if TERMINAL-TEST(*state*) then return UTILITY(*state*)
  $v \leftarrow -\infty$
  foreach *s* in SUCCESSORS(*state*) do
    $v \leftarrow$ MAX(v, MIN-VALUE(*s*, α, β))
    if $v \geq \beta$ then return v
    $\alpha \leftarrow$ MAX(α, v)
  return v

---

function MIN-VALUE(*state*, α, β) returns *a utility value*
same as MAX-VALUE but with roles of α, β reversed

Übung 21. Implementieren Sie den Minimax-Algorithmus und die α/β-Suche für ein Spiel wie Vier-Gewinnt oder Reversi/Othello.

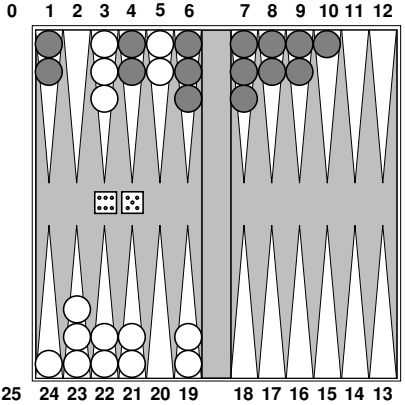
Vergleichen Sie die Laufzeiten beider Verfahren bei gleicher Suchtiefe.

Kombinatorische Spiele mit Zufallselementen

engl.: nondeterministic games

Zufall in Spielen bspw. durch Würfeln (Backgammon) oder durch Mischen und unbekannter Kartenverteilung (Skat).

Im Gegensatz zu rein kombinatorischen Spielen kann eine gute Stellung durch ungünstiges Würfeln oder eine schlechte Kartenverteilung nutzlos sein.



Können wir das Minimax-Prinzip auf solche Spiele übertragen?

Was ändert sich durch das Einbringen von Zufallselementen?

- Ein Spieler kann nicht mehr von einem garantierten Nutzen einer Strategie ausgehen.

  Die angewendete Strategie ist oft gut, kann aber im Einzelfall, also z.B. bei ungünstigem Würfeln fehlschlagen.

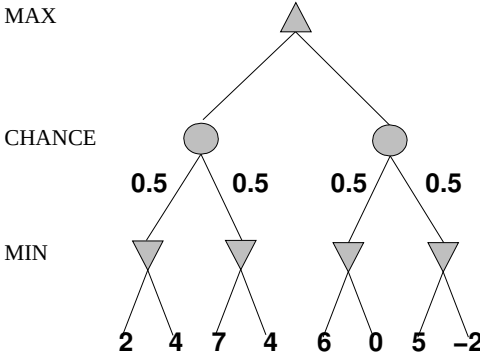
Wie gehen wir damit um?

- Bewerte eine Strategie oder auch einen einzelnen Zug mit dem durchschnittlichen Nutzen: Erwartungswert
- Ein Zufallsereignis $\mathcal{X}$ kann zu den Gewinnen $x_1, \ldots, x_n$ führen.
- $P(\mathcal{X} = x_i)$: Wahrscheinlichkeit, mit der Gewinn x_i auftritt.
- Erwartungswert $E(\mathcal{X})$ des Zufallsereignisses $\mathcal{X}$:

$$E(\mathcal{X}) = \sum_{i=1}^{n} x_i \cdot P(\mathcal{X} = x_i)$$

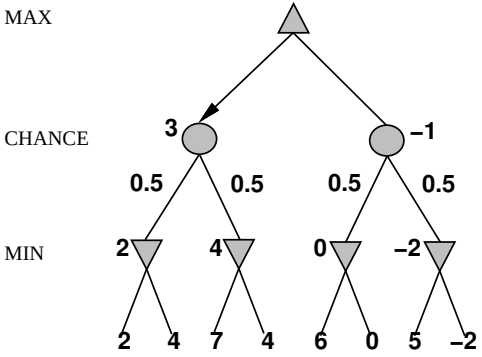
Neben den möglichen Zügen müssen wir nun im Spielbaum für die Bewertung eines Zuges die Zufallskomponente berücksichtigen:

- Zufallsereignisse werden als Knoten repräsentiert.
- Von einem Zufallsknoten ausgehende Kanten sind mit der Wahrscheinlichkeit markiert, mit der der jeweilige Nachfolgezustand erreicht wird.

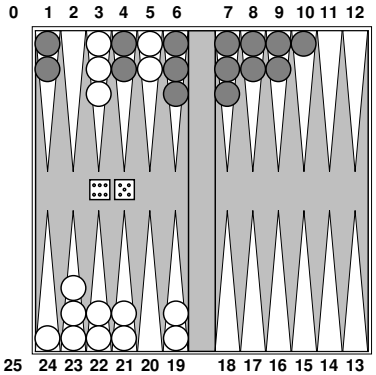


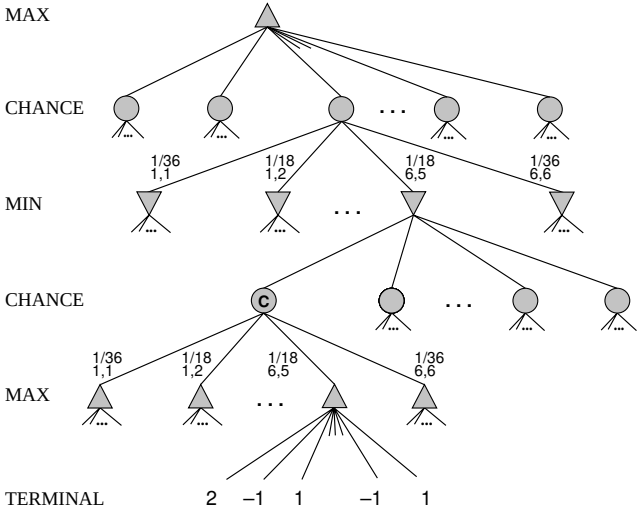
An Zufallsknoten berechnen wir den zugehörigen Erwartungswert:

- Die x_i entsprechen den Bewertungen der Kinder,
- die Wahrscheinlichkeiten $P(\mathcal{X} = x_i)$ ergeben sich durch die Markierungen der Kanten.



Kombinatorische Spiele mit Zufallselementen





Übersicht

Übersicht

Unter einer *Aussage* versteht man ein atomares sprachliches Gebilde, das einen der beiden *Wahrheitswerte* wahr oder falsch hat.

Wahre Aussagen:

- Krefeld liegt am Rhein.
- Es gibt unendlich viele Primzahlen.
- $3 + 4 = 7$

Falsche Aussagen:

- Duisburg liegt an der Elbe.
- Die Erde ist eine Scheibe.
- $3 + 4 = 8$

Aussagen, von denen wir zum jetzigen Zeitpunkt nicht wissen, ob sie wahr oder falsch sind:

- Es gibt außerirdisches Leben.
- $P \neq NP$

Es wird letztlich ignoriert, welche inhaltlichen Bedeutungen vorliegen, unser Interesse wird auf die Wahrheitswerte reduziert:

- A: Otto wird krank
- B: der Arzt verschreibt Otto eine Medizin

Da wir in der Umgangssprache oft einen zeitlichen und kausalen Ablauf voraussetzen, unterscheiden wir umgangssprachlich:

A *und* B „Otto wird krank *und* der Arzt verschreibt Otto eine Medizin“

B *und* A „der Arzt verschreibt Otto eine Medizin *und* Otto wird krank“

Von solchen Feinheiten befreien wir uns im Folgenden.

Gegenstand der Aussagenlogik:

- Untersuche einfache Verknüpfungen wie *und*, *oder* und *nicht*, zwischen Aussagen und
- lege fest, wie sich Wahrheitswerte der atomaren Bestandteile zu Wahrheitswerten von komplizierteren sprachlichen Gebilden fortsetzen lassen.

Inferenz:

- Herleiten von neuem Wissen auf Basis eines Kalküls.
- Definition des Folgerungsbegriffs.
- Übertragen der semantischen Folgerung auf äquivalente syntaktische Umformungen.

Zielrichtungen der Inferenz

- *Prognosen bzw. logische Ableitungen erstellen:*
  Wenn Fakten und Regeln gegeben sind, welche Folgerungen kann man daraus ziehen?
- *Erklärungen finden:*
  Wie lässt sich ein Fakt mittels der Regeln erklären?
- *Hypothesen prüfen:*
  Können aus den Fakten und Regeln die Hypothesen hergeleitet werden?

Es gibt verschiedene Arten von Schlussfolgerungen[32].

Deduktion: Schließe vom allgemeinen Fall auf einen speziellen Fall. Ist immer mit der Vorstellung verbunden, dass es sich um korrekte Schlussfolgerungen handelt, dass das abgeleitete Wissen also stets wahr ist.

Beispiele:

Vögel können fliegen.	Alle Holländer essen gern Käse.
Max ist ein Vogel.	Hans ist Holländer.
Max kann fliegen.	Hans isst gerne Käse.

[32] Schlussfolgerung wird in der Informatik meißt als Inferenz bezeichnet.

Induktion: Man erschließt regelhaftes Wissen aus einzelnen Sachverhalten. Nur so kann man sich weiträumig anwendbare Tatsachen über die wirkliche Welt erschließen. Sonst hätten wir nur Millionen von Einzelerfahrungen.

Allerdings geben wir dabei die Vorstellung auf, das neu abgeleitetes Wissen auch stets wahr ist.

Beispiel:

Bello ist ein Hund, Bello bellt, Bello beißt nicht.
Trixi ist ein Hund, Trixi bellt, Trixi beißt nicht.
Waldi ist ein Hund, Waldi bellt, Waldi beißt nicht.

Hunde, die bellen, beißen nicht. (???)

Abduktion: Wir suchen nach einer Erklärung für unsere Beobachtungen, wir bilden Hypothesen. Alle Diagnoseverfahren in der Medizin oder Technik basieren auf Abduktion.

Auch hier ist das abgeleitete Wissen nicht notwendig wahr.

Beispiel:

Max hat lange Ohren.	Beobachtung
Hasen haben lange Ohren.	Wissen
Max ist ein Hase. (???)	Schlussfolgerung

Arten der Inferenz

Jeder wissenschaftliche Erkenntnisprozess entsteht aus dem Zusammenspiel von Abduktion, Deduktion und Induktion:

- Finde eine Hypothese (Abduktion).
- Leite Voraussagen aus der Hypothese ab (Deduktion).
- Suche nach Fakten, die die Vorannahmen „verifizieren“ (Induktion).
- Werden keine solche Fakten gefunden, zurück zum Anfang.

- Alle Sätze sind Zeichenketten aus Buchstaben eines Alphabets, die nach präzisen Regeln einer Grammatik angeordnet werden.
- Um eine formale Sprache wie das Resolutionskalkül nutzen zu können, müssen wir sowohl deren Syntax als auch deren Semantik kennen.

Lesen Sie laut den Satz: „Die Aussage $A \wedge B$ ist genau dann wahr, wenn die Aussagen A und B wahr sind."

- Eine Trennung der Sprachebenen ist unerlässlich.
- Die formale Aussage $A \wedge B$ soll dadurch erklärt werden, dass auf einer metasprachlichen Ebene über die Aussage A wie auch über die Aussage B geredet wird.

Wir bezeichnen mit A_i eine atomare Formel. Eine *Formel* wird durch folgenden induktiven Prozess formuliert:

- Alle atomaren Formeln sind Formeln.
- *Konjunktion:* Für alle Formeln F und G ist $(F \wedge G)$ eine Formel.
- *Disjunktion:* Für alle Formeln F und G ist $(F \vee G)$ eine Formel.
- *Negation:* Für jede Formel F ist $\neg F$ eine Formel.

Eine Formel F, die als Teil einer Formel G auftritt, heißt *Teilformel*.

Beispiel: Die Teilformeln von $F = \neg((A_5 \wedge A_6) \vee \neg A_3)$ sind:

$$F,\ ((A_5 \wedge A_6) \vee \neg A_3),\ (A_5 \wedge A_6),\ A_5,\ A_6,\ \neg A_3,\ A_3$$

Wir führen abkürzende Schreibweisen ein:

$$(F_1 \to F_2) \quad \text{statt} \quad (\neg F_1 \vee F_2)$$

$$(F_1 \leftrightarrow F_2) \quad \text{statt} \quad ((F_1 \wedge F_2) \vee (\neg F_1 \wedge \neg F_2))$$

$$\left(\bigvee_{i=1}^{n} F_i\right) \quad \text{statt} \quad (\ldots((F_1 \vee F_2) \vee F_3) \vee \cdots \vee F_n)$$

$$\left(\bigwedge_{i=1}^{n} F_i\right) \quad \text{statt} \quad (\ldots((F_1 \wedge F_2) \wedge F_3) \wedge \cdots \wedge F_n)$$

Bisher sind Formeln lediglich Zeichenreihen:

- Sie haben keinen Inhalt, keine Interpretation.
- Wir müssen erst noch die Bedeutung festlegen.

Die Elemente der Menge $\{0, 1\}$ heißen Wahrheitswerte.

Eine *Belegung* ist eine Funktion $\mathcal{A} : \mathbb{D} \to \{0, 1\}$, die den atomaren Formeln $\mathbb{D}$ einen Wahrheitswert 0 oder 1 zuweist.

Wir müssen die Funktion $\mathcal{A}$ so erweitern, dass sie auch für Formeln und Teilformeln gilt:

- $\mathcal{A}((F \wedge G)) = \begin{cases} 1, & \text{falls } \mathcal{A}(F) = 1 \text{ und } \mathcal{A}(G) = 1 \\ 0, & \text{sonst} \end{cases}$
- $\mathcal{A}((F \vee G)) = \begin{cases} 1, & \text{falls } \mathcal{A}(F) = 1 \text{ oder } \mathcal{A}(G) = 1 \\ 0, & \text{sonst} \end{cases}$
- $\mathcal{A}(\neg F) = \begin{cases} 1, & \text{falls } \mathcal{A}(F) = 0 \\ 0, & \text{sonst} \end{cases}$

Beispiel: Es sei $\mathcal{A}(A) = 1$, $\mathcal{A}(B) = 1$ und $\mathcal{A}(C) = 0$, dann gilt:

$$
\begin{aligned}
&\mathcal{A}(\neg((A \wedge B) \vee C)) \\
&= \begin{cases} 1, & \text{falls } \mathcal{A}(((A \wedge B) \vee C)) = 0 \\ 0, & \text{sonst} \end{cases} \\
&= \begin{cases} 0, & \text{falls } \mathcal{A}(((A \wedge B) \vee C)) = 1 \\ 1, & \text{sonst} \end{cases} \\
&= \begin{cases} 0, & \text{falls } \mathcal{A}((A \wedge B)) = 1 \text{ oder } \mathcal{A}(C) = 1 \\ 1, & \text{sonst} \end{cases} \\
&= \begin{cases} 0, & \text{falls } \mathcal{A}((A \wedge B)) = 1 \text{ (da } \mathcal{A}(C) = 0) \\ 1, & \text{sonst} \end{cases} \\
&= \begin{cases} 0, & \text{falls } \mathcal{A}(A) = 1 \text{ und } \mathcal{A}(B) = 1) \\ 1, & \text{sonst} \end{cases} \\
&= 0
\end{aligned}
$$

Wir können die Wirkung der Operatoren $\wedge$, $\vee$, $\neg$ auch durch Verknüpfungstafeln darstellen:

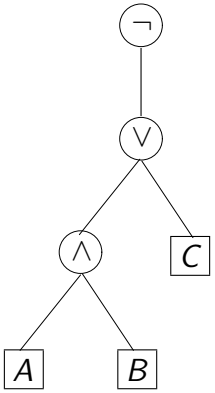
$\mathcal{A}(X)$	$\mathcal{A}(Y)$	$\mathcal{A}(\neg X)$	$\mathcal{A}(X \wedge Y)$	$\mathcal{A}(X \vee Y)$
0	0	1	0	0
0	1	1	0	1
1	0	0	0	1
1	1	0	1	1

Ebenso die Wirkung der Operatoren $\rightarrow$ und $\leftrightarrow$:

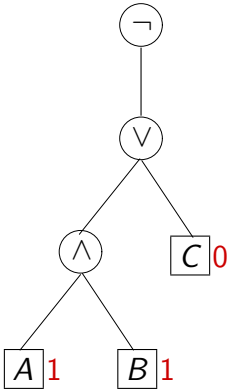
$\mathcal{A}(X)$	$\mathcal{A}(Y)$	$\mathcal{A}(X \rightarrow Y)$	$\mathcal{A}(X \leftrightarrow Y)$
0	0	1	1
0	1	1	0
1	0	0	0
1	1	1	1

Beachte: Aus einer falschen Aussage kann man alles folgern!

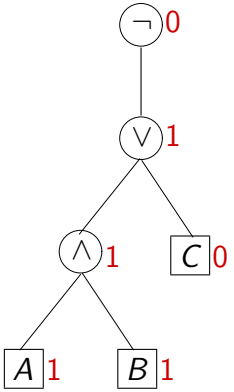
Man kann die Art und Weise, wie Formeln aus einfacheren Teilformeln aufgebaut sind, auch durch eine Baumstruktur darstellen:



Um den Wert der Formel bei einer gegebenen Belegung zu bestimmen, markiere zunächst die Blätter mit den durch die Belegung gegebenen Wahrheitswerten.



Markiere dann von unten nach oben alle Knoten anhand obiger Verknüpfungstafeln.



Definition: Sei F eine Formel und $\mathcal{A}$ eine Belegung.

- Falls $\mathcal{A}$ für alle in F vorkommenden atomaren Formeln definiert ist, heißt $\mathcal{A}$ zu F *passend*.
- Ist $\mathcal{A}$ zu F passend und gilt $\mathcal{A}(F) = 1$, so schreiben wir auch $\mathcal{A} \models F$ und sagen *$\mathcal{A}$ ist ein Modell für F* oder *$\mathcal{A}$ erfüllt F*.
- Falls $\mathcal{A}(F) = 0$, so schreiben wir $\mathcal{A} \not\models F$ und sagen *$\mathcal{A}$ ist kein Modell für F* oder *$\mathcal{A}$ erfüllt F nicht*.

Eine Formel F heißt

- *erfüllbar*, falls F mindestens ein Modell besitzt.
- *unerfüllbar*, falls es kein Modell für F gibt.
- *allgemeingültig* oder *Tautologie*, falls jede Belegung ein Modell für F ist. Wir schreiben dann: $\models F$
- *falsifizierbar*, falls es eine Belegung gibt, die kein Modell für die Formel ist.

Übung 22. Falls eine Formel F aus n atomaren Formeln aufgebaut ist, wie viele verschiedene Belegungen gibt es für F?

Übung 23. Wie viele verschiedene Formeln mit den atomaren Formeln $A_1, \ldots, A_n$ und verschiedenen Wahrheitswerteverläufen gibt es?

Übung 24. Zeigen Sie folgende wichtige Tautologien.

Modus Ponens	$(F \wedge (F \to G)) \to G$
Modus Tollens	$((F \to G) \wedge \neg G) \to \neg F$
Und-Elimination	$(F \wedge G) \to F$
Oder-Introduktion	$F \to (F \vee G)$
Resolutionsregel	$((F \to G) \wedge (\neg F \to H)) \to (G \vee H)$

Da wir in wissensbasierten Systemen nicht nur eine, sondern viele Regeln haben, müssen wir obige Begriffe auf Formelmengen erweitern. Sei $\mathcal{F}$ eine Menge von Formeln.

- Dann ist $\mathcal{A}$ ein *Modell* für $\mathcal{F}$, falls für alle $F \in \mathcal{F}$ gilt $\mathcal{A} \models F$, d.h. $\mathcal{A}$ erfüllt jede Formel in $\mathcal{F}$.

  Insbesondere ist dann $\mathcal{A}$ passend für jede Formel $F \in \mathcal{F}$.
- $\mathcal{F}$ heißt *erfüllbar*, falls $\mathcal{F}$ mindestens ein Modell besitzt, andernfalls heißt $\mathcal{F}$ *unerfüllbar*.

Eine Formel G heißt *Folgerung* der Formeln $F_1, \ldots, F_k$, falls für jede passende Belegung $\mathcal{A}$ für $F_1, \ldots, F_k$ und G gilt:

$$\mathcal{A} \text{ ist Modell für } \{F_1, \ldots, F_k\} \;\Rightarrow\; \mathcal{A} \text{ ist Modell für } G$$

Übung 25. Wir wollen uns ein Haustier anschaffen und machen einige Überlegungen:

- Es soll nur ein Hund (H), eine Katze (K) oder ein Hamster (M) sein.
- Besitzer wertvoller Möbel (W) sollten keine Katze anschaffen, da diese die Möbel zerkratzt.
- Ein Hund erfordert ein freistehendes Haus (F), damit kein Nachbar durch das Bellen gestört wird.

Wir vermuten: Für einen Besitzer wertvoller Möbel ohne freistehendes Haus kommt nur ein Hamster in Frage.

- Wie lauten die Aussagen als aussagenlogische Formeln?
- Wie lautet die Hypothese als aussagenlogische Formel?
- Folgt die Hypothese aus der Formelmenge?

Übung 26. Inspektor Columbo hat drei Personen in Verdacht, die eine Straftat (evtl. gemeinsam) begangen haben. Ihm stehen folgende Informationen zur Verfügung:

- Wenn A schuldig ist und B unschuldig, dann ist C schuldig.
- C arbeitet niemals allein.
- A arbeitet niemals mit C.
- Nur A, B oder C kommen als Täter in Frage.

Inspektor Columbo vermutet, dass B einer der Täter ist. Helfen Sie ihm, indem Sie die Sachverhalte als aussagenlogische Formeln darstellen. Folgt die Hypothese aus der Formelmenge?

Hatte B einen Komplizen? Begründen Sie Ihre Vermutung.

Übung 27. Geben Sie eine Formelmenge $\mathcal{F}$ an, so dass jede echte Teilmenge von $\mathcal{F}$ erfüllbar ist, $\mathcal{F}$ selbst jedoch nicht.

Übung 28. Ist folgende unendliche Formelmenge $\mathcal{F}$ erfüllbar?

$$\mathcal{F} = \{A_1 \vee A_2,\ \neg A_2 \vee \neg A_3,\ A_3 \vee A_4,\ \neg A_4 \vee \neg A_5,\ \ldots\}$$

Übung 29. Zeigen Sie, dass folgende Behauptungen äquivalent sind:

- G ist eine Folgerung von $F_1, \ldots, F_k$
- $((\bigwedge_{i=1}^{k} F_i) \rightarrow G)$ ist eine Tautologie
- $((\bigwedge_{i=1}^{k} F_i) \wedge \neg G)$ ist unerfüllbar

Anmerkungen:

- Wir sehen an unseren Beispielen (Haustier, Inspektor Columbo), dass das Testen von Belegungen ineffizient ist.
- Für maschinelle Inferenz wollen wir Techniken nutzen, die allein auf der Syntax der Formeln beruhen.
- Wir suchen eine Folge von syntaktischen Umformungen, die die Hypothese beweisen.

Daher beschäftigen wir uns im Folgenden mit Äquivalenz und Normalformen.

Übersicht

Syntaktisch unterschiedliche Formeln können dasselbe aussagen:

- $(F \wedge G)$ und $(G \wedge F)$ bedeuten dasselbe, obwohl es syntaktisch zwei verschiedene Objekte sind.
- Auch $\neg(F \vee G)$ und $\neg F \wedge \neg G$ sagen dasselbe aus, wie man an den Wahrheitstafeln erkennen kann.

F	G	$\neg(F \vee G)$	$\neg F \wedge \neg G$
0	0	1	1
0	1	0	0
1	0	0	0
1	1	0	0

Im Weiteren: Um eine Hypothese zu beweisen, wollen wir nicht mehr alle Belegungen testen, sondern eine Folge von syntaktischen Umformungen durchführen.

Definition: Zwei Formeln F und G heißen *(semantisch) äquivalent*, falls für alle Belegungen $\mathcal{A}$, die für F und auch für G passend sind, $\mathcal{A}(F) = \mathcal{A}(G)$ gilt. Wir schreiben dann $F \equiv G$.

$$
\begin{array}{rcll}
(F \wedge F) & \equiv & F & \text{Idempotenz} \\
(F \vee F) & \equiv & F & \\
(F \wedge G) & \equiv & (G \wedge F) & \text{Kommutativität} \\
(F \vee G) & \equiv & (G \vee F) & \\
((F \wedge G) \wedge H) & \equiv & (F \wedge (G \wedge H)) & \text{Assoziativität} \\
((F \vee G) \vee H) & \equiv & (F \vee (G \vee H)) & \\
(F \wedge (F \vee G)) & \equiv & F & \text{Absorption} \\
(F \vee (F \wedge G)) & \equiv & F &
\end{array}
$$

Äquivalenz

$(F \wedge (G \vee H))$	$\equiv$	$((F \wedge G) \vee (F \wedge H))$	Distributivität
$(F \vee (G \wedge H))$	$\equiv$	$((F \vee G) \wedge (F \vee H))$	
$\neg\neg F$	$\equiv$	F	Doppelnegation
$\neg(F \wedge G)$	$\equiv$	$(\neg F \vee \neg G)$	de-Morgansche-
$\neg(F \vee G)$	$\equiv$	$(\neg F \wedge \neg G)$	Regeln
$(F \vee G)$	$\equiv$	F, falls F Tautologie	Tautologieregeln
$(F \wedge G)$	$\equiv$	G, falls F Tautologie	
$(F \vee G)$	$\equiv$	G, falls F unerfüllbar	Unerfüllbarkeits-
$(F \wedge G)$	$\equiv$	F, falls F unerfüllbar	regeln

Beweis: Alle obigen Äquivalenzen können leicht mittels Wahrheitstafeln nachgeprüft werden.

Beispiel Absorption: $F \wedge (F \vee G)) \equiv F$

$\mathcal{A}(F)$	$\mathcal{A}(G)$	$\mathcal{A}((F \vee G))$	$\mathcal{A}((F \wedge (F \vee G)))$
0	0	0	0
0	1	1	0
1	0	1	1
1	1	1	1

Da die erste und die letzte Spalte übereinstimmen, ist die Äquivalenz der beiden Formeln gezeigt.

Übung 30. Zeigen Sie sowohl durch Wahrheitstafeln als auch durch Anwendung obiger Umformungsregeln, dass

$$((A \vee \neg(B \wedge A)) \wedge (C \vee (D \vee C)))$$

äquivalent ist zu $(C \vee D)$.

Übung 31. Zeigen Sie, dass es zu jeder Formel F eine äquivalente Formel G gibt, die nur die Operatoren $\neg$ und $\rightarrow$ enthält.

Übung 32. Zeigen Sie sowohl durch Wahrheitstafeln als auch durch Anwendung obiger Umformungsregeln, dass die beiden folgenden Formeln äquivalent sind.

- $(\neg P \vee Q) \leftrightarrow R$
- $(\neg P \wedge R) \vee (Q \wedge R) \vee (P \wedge \neg Q \wedge \neg R)$

Übung 33. Beweisen Sie die folgenden Verallgemeinerungen der de-Morganschen-Gesetze:

$$\neg \left(\bigvee_{i=1}^{n} F_i \right) \equiv \left(\bigwedge_{i=1}^{n} \neg F_i \right) \qquad \text{und} \qquad \neg \left(\bigwedge_{i=1}^{n} F_i \right) \equiv \left(\bigvee_{i=1}^{n} \neg F_i \right)$$

Übung 34. Beweisen Sie die folgenden Verallgemeinerungen der Distributivgesetze:

$$\left(\left(\bigvee_{i=1}^{m} F_i\right) \wedge \left(\bigvee_{j=1}^{n} G_j\right)\right) \equiv \left(\bigvee_{i=1}^{m}\left(\bigvee_{j=1}^{n}(F_i \wedge G_j)\right)\right)$$

$$\left(\left(\bigwedge_{i=1}^{m} F_i\right) \vee \left(\bigwedge_{j=1}^{n} G_j\right)\right) \equiv \left(\bigwedge_{i=1}^{m}\left(\bigwedge_{j=1}^{n}(F_i \vee G_j)\right)\right)$$

Im Folgenden: Jede, auch noch so kompliziert aussehende Formel kann in eine gewisse Normalform überführt werden. Obige Umformungsregeln sind dafür ausreichend.

- Ein *Literal* ist eine atomare Formel oder die Negation einer atomaren Formel.
- Eine Formel F ist in *konjunktiver Normalform* (KNF), falls sie eine Konjunktion von Disjunktionen von Literalen ist:

$$F = \left(\bigwedge_{i=1}^{n} \left(\bigvee_{j=1}^{m_i} L_{i,j} \right) \right)$$

- Eine Formel F ist in *disjunktiver Normalform* (DNF), falls sie eine Disjunktion von Konjunktionen von Literalen ist:

$$F = \left(\bigvee_{i=1}^{n} \left(\bigwedge_{j=1}^{m_i} L_{i,j} \right) \right)$$

- Die $L_{i,j}$ sind Literale mit $L_{i,j} \in \{A_1, A_2, \ldots\} \cup \{\neg A_1, \neg A_2, \ldots\}$.

Satz: Für jede Formel F gibt es eine äquivalente Formel in KNF und eine äquivalente Formel in DNF. (Beweis durch Induktion über den Formelaufbau von F, wird hier nicht dargestellt.)

Umformungsmethode zur Herstellung der KNF:

- Ersetze in F jedes Vorkommen von

$$\begin{array}{rcl} \neg\neg G & \text{durch} & G \\ \neg(G \wedge H) & \text{durch} & (\neg G \vee \neg H) \\ \neg(G \vee H) & \text{durch} & (\neg G \wedge \neg H) \end{array}$$

  bis keine derartige Teilformel mehr vorkommt.
- Ersetze in F jedes Vorkommen von

$$\begin{array}{rcl} (F \vee (G \wedge H)) & \text{durch} & ((F \vee G) \wedge (F \vee H)) \\ ((F \wedge G) \vee H) & \text{durch} & ((F \vee H) \wedge (G \vee H)) \end{array}$$

  bis keine derartige Teilformel mehr vorkommt.

Oder wir erstellen die Wahrheitstafel zur Formel F und lesen direkt die Teilformeln der disjunktiven Normalform ab:

A	B	C	F		
0	0	0	1	$\rightarrow$	$\neg A \wedge \neg B \wedge \neg C$
0	0	1	0		
0	1	0	0		
0	1	1	1	$\rightarrow$	$\neg A \wedge B \wedge C$
1	0	0	1	$\rightarrow$	$A \wedge \neg B \wedge \neg C$
1	0	1	1	$\rightarrow$	$A \wedge \neg B \wedge C$
1	1	0	0		
1	1	1	0		

Die abgelesenen Teilformeln werden mittels $\vee$ verknüpft:

$$(\neg A \wedge \neg B \wedge \neg C) \ \vee \ (\neg A \wedge B \wedge C) \ \vee \ (A \wedge \neg B \wedge \neg C) \ \vee \ (A \wedge \neg B \wedge C)$$

Die Korrektheit dieses Verfahrens ist intuitiv klar.

Analog können wir aus der Wahrheitstafel einer Formel F direkt die Teilformeln der konjunktiven Normalform ablesen:

A	B	C	F		
0	0	0	1		
0	0	1	0	$\rightarrow$	$A \vee B \vee \neg C$
0	1	0	0	$\rightarrow$	$A \vee \neg B \vee C$
0	1	1	1		
1	0	0	1		
1	0	1	1		
1	1	0	0	$\rightarrow$	$\neg A \vee \neg B \vee C$
1	1	1	0	$\rightarrow$	$\neg A \vee \neg B \vee \neg C$

Die abgelesenen Teilformeln werden mittels $\wedge$ verknüpft:

$$(A \vee B \vee \neg C) \ \wedge \ (A \vee \neg B \vee C) \ \wedge \ (\neg A \vee \neg B \vee C) \ \wedge \ (\neg A \vee \neg B \vee \neg C)$$

Dieses Verfahren ist intuitiv vielleicht nicht sofort klar, aber ...

... wir können hier die Darstellung in DNF benutzen, um uns die Korrektheit des Verfahrens klar zu machen:

A	B	C	F				
0	0	0	1				
0	0	1	0	$\rightarrow$	$A \vee B \vee \neg C$	$\equiv$	$\neg(\neg A \wedge \neg B \wedge C)$
0	1	0	0	$\rightarrow$	$A \vee \neg B \vee C$	$\equiv$	$\neg(\neg A \wedge B \wedge \neg C)$
0	1	1	1				
1	0	0	1				
1	0	1	1				
1	1	0	0	$\rightarrow$	$\neg A \vee \neg B \vee C$	$\equiv$	$\neg(A \wedge B \wedge \neg C)$
1	1	1	0	$\rightarrow$	$\neg A \vee \neg B \vee \neg C$	$\equiv$	$\neg(A \wedge B \wedge C)$

Wir erstellen die KNF für die negierte Formel und erhalten:

$$\neg\big((\neg A \wedge \neg B \wedge C) \vee (\neg A \wedge B \wedge \neg C) \vee (A \wedge B \wedge \neg C) \vee (A \wedge B \wedge C)\big)$$

Wenn wir jetzt die Negation nach den de-Morganschen-Regeln in die Klammer ziehen, erhalten wir genau das gewünschte.

Anmerkung: Die so erzeugten KNF- bzw. DNF-Formeln sind nicht notwendigerweise die kürzesten. Z.B. kann man

$$(A \vee B \vee \neg C) \ \wedge \ (A \vee \neg B \vee C) \ \wedge \ (\neg A \vee \neg B \vee C) \ \wedge \ (\neg A \vee \neg B \vee \neg C)$$

vereinfachen zu

$$(A \vee B \vee \neg C) \ \wedge \ (A \vee \neg B \vee C) \ \wedge \ (\neg A \vee \neg B)$$

Übung 35. Erzeugen Sie mittels Umformung und mittels einer Wahrheitstafel zu der Formel

$$F = ((\neg A \rightarrow B) \wedge ((A \wedge \neg C) \leftrightarrow B))$$

eine äquivalente DNF und KNF.

Übersicht

Definition: Eine Formel F ist eine *Hornformel*, falls F in KNF ist, und jedes Disjunktionsglied in F höchstens ein positives Literal enthält. Wurde benannt nach dem Logiker Alfred Horn.

Beispiel:

$$F = (A \vee \neg B) \wedge (\neg C \vee \neg A \vee D) \wedge (\neg A \vee \neg B) \wedge D \wedge \neg E$$

Hornformeln können anschaulicher geschrieben werden:

$$F = (B \to A) \wedge (C \wedge A \to D) \wedge (A \wedge B \to 0) \wedge (1 \to D) \wedge (E \to 0)$$

Dabei steht 0 für eine beliebige unerfüllbare Formel und 1 für eine beliebige Tautologie.

Algorithmischer Test für die Erfüllbarkeit oder Unerfüllbarkeit einer gegebenen Hornformel F:

initial: für alle Teilformeln $1 \to A_i$ aus F markiere A_i

 # A_i muss mit 1 belegt sein, um F zu erfüllen

wiederhole

- markiere y, falls es eine Teilformel $(A_{i_1} \wedge \ldots \wedge A_{i_k}) \to y$ gibt, bei der $A_{i_1}, \ldots, A_{i_k}$ bereits markiert sind

  # y muss mit 1 belegt sein, um F zu erfüllen
- falls es eine Teilformel $(A_{i_1} \wedge \ldots \wedge A_{i_k}) \to 0$ gibt, bei der $A_{i_1}, \ldots, A_{i_k}$ bereits markiert sind, gib *unerfüllbar* aus und stoppe

gib *erfüllbar* aus und stoppe

Die Korrektheit des Algorithmus ergibt sich anhand der obigen Bemerkungen.

Übung 36. Wenden Sie den Markierungsalgorithmus auf die Formel

$$(\neg A \vee \neg B \vee \neg D) \wedge \neg E \wedge (\neg C \vee A) \wedge C \wedge B \wedge (\neg G \vee D) \wedge G$$

an. Die Wahrheitstafel hätte für diese Formel $2^6 = 64$ Zeilen!

Übung 37. Geben Sie eine Formel an, zu der es keine äquivalente Hornformel gibt und begründen Sie, warum das so ist.

Übung 38. Angenommen, wir könnten die folgenden chemischen Reaktionen durchführen:

$$\begin{aligned} MgO + H_2 &\rightarrow Mg + H_2O \\ C + O_2 &\rightarrow CO_2 \\ H_2O + CO_2 &\rightarrow H_2CO_3 \end{aligned}$$

Ist es dann möglich, H_2CO_3 aus den Grundstoffen MgO, H_2, O_2 und C herzustellen?

Übersicht

Aufgabenstellung

Ziel unserer Bemühungen in diesem ganzen Kapitel ist: Wir wollen die *Unerfüllbarkeit* einer gegebenen Formelmenge nachweisen!

Frage: Warum können wir uns auf Unerfüllbarkeitstests beschränken?

Antwort: Um zu testen,

- ob eine Formel F eine Tautologie ist, testen wir einfach $\neg F$ auf Unerfüllbarkeit.
- ob eine Formel G eine Folgerung einer Formelmenge $\mathcal{F} = \{F_1, F_2, \ldots, F_k\}$ ist, testen wir die Unerfüllbarkeit von $F_1 \wedge F_2 \wedge \ldots \wedge F_k \wedge \neg G$.

Resolution

- Resolution ist eine einfach anzuwendende syntaktische Umformungsregel.
- Generiere aus zwei Formeln eine dritte Formel, die dann wieder als Eingabe in weitere Resolutionsschritte dienen kann.
- Voraussetzung für die Anwendung der Resolution: Die Formeln müssen in konjunktiver Normalform vorliegen.
- Eine Kollektion rein mechanisch anzuwendender syntaktischer Umformungsregeln nennen wir *Kalkül*.

Kalkül

Ein Kalkül ergänzt die Aussagenlogik oder die Prädikatenlogik um den syntaktischen Begriff des Ableitens, der dem semantischen Folgerungsbegriff entspricht.

- Die Begriffe „wahr“ und „falsch“ kommen in diesem System zunächst gar nicht vor.
- Stattdessen werden Axiome gesetzt, die einfach als Zeichenketten angesehen werden.
- Aus diesen Zeichenketten können weitere ableitbare Zeichenketten aufgrund von bestimmten Schlussregeln (Inferenzregeln) hergeleitet werden.

- In dem formalen System sollen nur Zeichenketten (Sätze) hergeleitet werden können, die bei einer plausiblen Interpretation auch wahr sind. → *Korrektheit*
- Alle Sätze, die als „wahr“ interpretierbar sind, sollen auch hergeleitet werden können. → *Vollständigkeit*

Klassische Aussagenlogik:

- Die Symbole $\wedge$, $\vee$ und $\neg$ mit den bekannten Interpretationen bilden die Grundlage eines Kalküls (nach George Boole).
- Kalküle sind korrekt als auch vollständig.

Komplexere logische Systeme wie bspw. die Mengenlehre:

- Es ist unmöglich, einen vollständigen Kalkül aufzustellen, der auch korrekt ist,
- siehe Gödelscher Unvollständigkeitssatz.

Die Definition eines Kalküls macht nur dann Sinn, wenn dessen Korrektheit und Vollständigkeit in Bezug zur Aufgabenstellung nachgewiesen werden kann:

- *Aufgabenstellung:* Wir wollen die Unerfüllbarkeit einer gegebenen Formelmenge nachweisen.
- *Korrektheit:* Keine erfüllbare Formelmenge wird durch den Kalkül als vermeintlich unerfüllbar nachgewiesen.
- *Vollständigkeit:* Jede unerfüllbare Formelmenge wird durch den Kalkül als solche nachgewiesen.

Klausel

Seien $A_1, A_2, A_3, \ldots$ aussagenlogische Variablen. Eine endliche Menge $K \subset \{A_1, A_2, A_3, \ldots\} \cup \{\neg A_1, \neg A_2, \neg A_3, \ldots\}$ heißt eine *Klausel*.

Die Disjunktionsglieder einer konjunktiven Normalform können über Klauseln dargestellt werden.

Beispiel: Die Formel $(A \vee B) \wedge (\neg A \vee C)$ führt zu den Klauseln $\{A, B\}$ und $\{\neg A, C\}$.

Die Menge der Klauseln zu den Disjunktionsgliedern heißt *Klauselmenge der aussagenlogischen Formel*.

Beispiel: Die Formel $(A \vee B) \wedge (\neg A \vee C)$ führt zur Klauselmenge $\{\{A, B\},\ \{\neg A, C\}\}$.

Seien K_1, K_2 und R Klauseln. R heißt *Resolvente* von K_1 und K_2, falls es ein Literal L gibt mit $L \in K_1$ und $\overline{L} \in K_2$ und R die Form hat:

$$R = (K_1 - \{L\}) \cup (K_2 - \{\overline{L}\})$$

Beispiel:

$$\begin{array}{ccc} \{A, \neg B, C\} & & \{B, C, D\} \\ & \searrow \quad \swarrow & \\ & \{A, C, D\} & \end{array}$$

Übung 39. Kann bei der Resolution zweier Hornformeln eine Formel entstehen, die keine Hornformel ist? Begründen Sie Ihre Meinung.

Übung 40. Dürfen die Klauseln $\{A, B, C\}$ und $\{A, \neg B, \neg C\}$ zu $\{A\}$ resolviert werden?

Sei $\mathcal{K}$ eine Klauselmenge. Dann definieren wir:

$$Res(\mathcal{K}) := \mathcal{K} \cup \{R : R \text{ ist Resolvente zweier Klauseln von } \mathcal{K}\}$$

Außerdem setzen wir:

$$\begin{aligned} Res^0(\mathcal{K}) &:= \mathcal{K} \\ Res^{n+1}(\mathcal{K}) &:= Res(Res^n(\mathcal{K})) \\ Res^*(\mathcal{K}) &:= \bigcup_{n=1}^{\infty} Res^n(\mathcal{K}) \end{aligned}$$

Übung 41. Bestimmen Sie für folgende Klauselmenge $\mathcal{K}$ die Mengen $Res^n(\mathcal{K})$ für $n = 0, 1, 2$.

$$\mathcal{K} = \{\{A, \neg B, C\},\ \{B, C\},\ \{\neg A, C\},\ \{B, \neg C\},\ \{\neg C\}\}$$

Übung 42. Geben Sie sämtliche Resolventen an, die aus den Klauseln der folgenden Klauselmenge gewonnen werden können:

$$\{\{A, \neg B, E\}, \{A, B, C\}, \{\neg A, \neg D, E\}, \{A, \neg C\}\}$$

Übung 43. Beweisen Sie, dass es für jede endliche Klauselmenge $\mathcal{K}$ ein $k \geq 0$ gibt mit

$$Res^k(\mathcal{K}) = Res^{k+1}(\mathcal{K}) = \ldots = Res^*(\mathcal{K}).$$

Resolutions-Lemma

Sei F eine Formel in konjunktiver Normalform, dargestellt als Klauselmenge. Ferner sei R eine Resolvente zweier Klauseln K_1 und K_2 aus F. Dann sind F und $F \cup \{R\}$ äquivalent.

Beweis:

- Falls F unerfüllbar ist: Wenn wir F mit einer weiteren Klausel $\wedge$-verknüpfen, ist auch die neue Formel unerfüllbar.
- Falls F mit einer Variablenbelegung $\mathcal{A}$ erfüllbar ist:
  - Beide Eingangsklauseln K_1 und K_2 sind mit $\mathcal{A}$ erfüllt.
  - Das für die Resolution benutzte Literal L kommt einmal negiert und einmal nicht-negiert in K_1 bzw. K_2 vor, also ist entweder $\mathcal{A}(L) = 0$ oder $\mathcal{A}(\overline{L}) = 0$.
  - Die restliche Klausel beim nicht-erfüllten Auftreten muss erfüllt sein, damit die Klausel und damit auch F erfüllt ist.
  - Auch die Resolvente ist erfüllt, da in jedem Disjunktionsglied nur ein einziges Literal erfüllt sein muss.

Satz: Eine Formel in konjunktiver Normalform mit Klauselmenge $\mathcal{K}$ ist genau dann unerfüllbar, falls mittels Resolution die leere Menge abgeleitet werden kann, also $\emptyset \in Res^*(\mathcal{K})$.

Wir zeigen zunächst: $\mathcal{K}$ unerfüllbar $\Leftarrow \emptyset \in Res^*(\mathcal{K})$

Sei also $\emptyset \in Res^k(\mathcal{K})$ und k minimal.

- In $Res^{k-1}(\mathcal{K})$ gibt es zwei Klauseln vom Typ $\{A\}$ und $\{\neg A\}$.
- $A \wedge \neg A$ ist unerfüllbar, also ist $Res^{k-1}(\mathcal{K})$ unerfüllbar.
- Aufgrund des Resolutions-Lemmas gilt:

$$\mathcal{K} \equiv Res^1(\mathcal{K}) \equiv Res^2(\mathcal{K}) \equiv \ldots \equiv Res^n(\mathcal{K}) \equiv \ldots$$

- Somit ist auch $\mathcal{K}$ unerfüllbar.

Noch zu zeigen: $\mathcal{K}$ unerfüllbar $\Rightarrow \emptyset \in Res^*(\mathcal{K})$

Beweis durch Induktion über die Anzahl der Variablen n einer Klauselmenge $\mathcal{K}$.

- Induktionsanfang für $n = 1$:

  $\mathcal{K} = \{\{A\}, \{\neg A\}\}$ ist die einzige unerfüllbare Klauselmenge (falls die Variable A genannt wird) und $\emptyset$ entsteht bei der Resolution.
- Induktionsannahme:

  Eine fortgesetzte Resolution liefert für jede unerfüllbare Klauselmenge mit n Variablen $A_1, A_2, \ldots, A_n$ u.a. die leere Menge.
- Induktionsschritt: Zeige dies für $n + 1$ Variablen.

Induktionsschritt: Sei $\mathcal{K}$ eine unerfüllbare Klauselmenge mit $n+1$ Variablen $A_1, A_2, \ldots, A_{n+1}$.

- $\mathcal{K}_0$ entstehe aus $\mathcal{K}$ durch Weglassen von A_{n+1} aus allen Klauseln und Weglassen aller Klauseln mit $\neg A_{n+1}$.
  $\rightarrow$ belege A_{n+1} mit 0
- $\mathcal{K}_1$ entstehe aus $\mathcal{K}$ durch Weglassen von $\neg A_{n+1}$ aus allen Klauseln und Weglassen aller Klauseln mit A_{n+1}.
  $\rightarrow$ belege A_{n+1} mit 1

Gegeben sei eine beliebige Belegung $\mathcal{A}$ der Variablen. Da $\mathcal{K}$ unerfüllbar ist, gilt:

- $\mathcal{K}_0$ ist unerfüllbar, falls $\mathcal{A}(A_{n+1}) = 0$.
- $\mathcal{K}_1$ ist unerfüllbar, falls $\mathcal{A}(A_{n+1}) = 1$.

Nach Induktionsannahme kann man also in beiden Fällen mittels Resolution $\emptyset$ erzeugen:

$$\emptyset \in Res^*(\mathcal{K}_0) \quad \text{und} \quad \emptyset \in Res^*(\mathcal{K}_1)$$

Fügt man A_{n+1} und $\neg A_{n+1}$ wieder hinzu, so liefert die Resolution für das erweiterte $\mathcal{K}_0$ eine Menge, in der $\{A_{n+1}\}$ enthalten ist, für das erweiterte $\mathcal{K}_1$ ist $\{\neg A_{n+1}\}$ enthalten. Also

$$\{A_{n+1}\} \in Res^*(\mathcal{K}_0) \quad \text{und} \quad \{\neg A_{n+1}\} \in Res^*(\mathcal{K}_1)$$

Die Resolvente von beiden Mengen ist $\emptyset$, also $\emptyset \in Res^*(\mathcal{K})$.

Übung 44. Zeigen Sie mittels Resolution die Unerfüllbarkeit der Klauselmenge

$$\{\{A, B, \neg C\},\ \{\neg A\},\ \{A, B, C\},\ \{A, \neg B\}\},$$

also der Formel

$$(A \vee B \vee \neg C)\ \wedge\ (\neg A)\ \wedge\ (A \vee B \vee C)\ \wedge\ (A \vee \neg B).$$

Übung 45. Leiten Sie die Hypothese „für einen Besitzer wertvoller Möbel ohne freistehendes Haus kommt nur ein Hamster in Frage“ aus Übung 25 mittels Resolution her.

Übung 46. Diätplan: „Das ist ja unmöglich, mit dieser Baukasten-Diät“, seufzt Robert.

- Wenn ich einen Apfel esse, muss ich auch einen Salzhering nehmen. Wegen der Mineralien.
- Esse ich eine Banane, muss ich eine Scheibe Knäckebrot hinterher essen - oder ich lasse den Salzhering liegen.
- Aber wenn ich ein Tortenstück esse, darf ich kein Knäckebrot nehmen - zu viele Kohlehydrate.
- Esse ich einen Apfel, soll ich auch ein Tortenstück nehmen.
- Wenn ich keinen Milchreis esse, soll ich eine Banane nehmen.
- Ich esse auf jeden Fall einen Apfel, aber keinen Milchreis - den vertrage ich nicht.

„Pack den Diätplan weg“, sagt Sabine, „er ist für dich wirklich unmöglich!“.

Frage: Hat Sabine Recht?

Grenzen der Aussagenlogik

Wir können keine Aussagen über ganze Klassen von Objekten machen, so dass Schlussfolgerungen für individuelle Objekte möglich sind.

Beispiel: Gegeben seien folgende Fakten bzw. Regeln:

- Martin ist Informatiker.
- Peter ist Informatiker.
- Jeder Informatiker kann programmieren.

Folgerungen:

- Martin kann programmieren.
- Peter kann programmieren.

Übersicht

Übersicht

Die Prädikatenlogik ist eine Erweiterung der Aussagenlogik. Hinzu kommen der Allquantor $\forall$, der Existenzquantor $\exists$ sowie Funktions- und Prädikatsymbole.

Analog zur Aussagenlogik kann auch für die Prädikatenlogik deren Syntax und Semantik mathematisch präzise definiert werden.

Syntax der Prädikatenlogik:

- Eine *Variable* hat die Form x_i mit $i = 1, 2, 3, \ldots$
- Ein *Prädikatsymbol* hat die Form P_i^k mit $i = 1, 2, 3, \ldots$ und $k = 0, 1, 2, \ldots$
- Ein *Funktionssymbol* hat die Form f_i^k mit $i = 1, 2, 3, \ldots$ und $k = 0, 1, 2, \ldots$
- Man nennt i den *Unterscheidungsindex* und k die *Stellenzahl* oder auch *Stelligkeit*.

Der Begriff *Term* wird durch einen induktiven Prozess definiert:

- Jede Variable ist ein Term.
- Falls f ein Funktionssymbol ist mit Stellenzahl k, und falls $t_1, \ldots, t_k$ Terme sind, so ist auch $f(t_1, \ldots, t_k)$ ein Term.

Auch Funktionssymbole der Stellenzahl 0 sollen eingeschlossen sein, in einem solchen Fall fallen die Klammern weg.

Nullstellige Funktionssymbole heißen auch *Konstanten*.

Auch die Formeln der Prädikatenlogik können wir induktiv definieren:

- Falls P ein Prädikatsymbol der Stelligkeit k ist, und falls $t_1, \ldots, t_k$ Terme sind, dann ist $P(t_1, \ldots, t_k)$ eine Formel.

  Solche Formeln nennen wir auch *atomare Formeln*.
- Für jede Formel F ist auch $\neg F$ eine Formel.
- Für alle Formeln F und G sind auch $(F \wedge G)$ und $(F \vee G)$ Formeln.
- Falls x eine Variable ist und F eine Formel, so sind auch $\exists x\ F$ und $\forall x\ F$ Formeln.

Falls F eine Formel ist und F als Teil einer Formel G auftritt, so heißt F *Teilformel* von G.

Alle Vorkommen von Variablen in einer Formel werden in *freie* und *gebundene* Vorkommen unterteilt:

- Ein Vorkommen der Variablen x in der Formel F heißt *gebunden*, falls x in einer Teilformel der Form $\exists x\ G$ oder $\forall x\ G$ vorkommt,
- andernfalls heißt dieses Vorkommen von x *frei*.

$\rightarrow$ Dieselbe Variable kann also in einer Formel an verschiedenen Stellen sowohl frei als auch gebunden vorkommen.

Eine Formel ohne Vorkommen einer freien Variablen heißt *geschlossen* oder auch eine *Aussage*.

Beispiel:

$$F = (\exists x_1\ P_5^2(x_1, f_2^1(x_2)) \vee \neg\forall x_2\ P_4^2(x_2, f_7^2(f_4^0, f_5^1(x_3))))$$

ist eine Formel. Sämtliche Teilformeln von F sind:

- F
- $\exists x_1\ P_5^2(x_1, f_2^1(x_2))$
- $P_5^2(x_1, f_2^1(x_2))$
- $\neg\forall x_2\ P_4^2(x_2, f_7^2(f_4^0, f_5^1(x_3)))$
- $\forall x_2\ P_4^2(x_2, f_7^2(f_4^0, f_5^1(x_3)))$
- $P_4^2(x_2, f_7^2(f_4^0, f_5^1(x_3)))$

Weiterhin gilt:

- Alle Vorkommen von x_1 in F sind gebunden.
- Erstes Vorkommen von x_2 ist frei, zweites ist gebunden.
- x_3 kommt in F frei vor.

$\rightarrow$ Die Formel F ist nicht geschlossen bzw. keine Aussage.

Alle in F vorkommenden Terme sind:

- x_1
- x_2
- $f_2^1(x_2)$
- $f_7^2(f_4^0, f_5^1(x_3))$
- f_4^0
- $f_5^1(x_3))$
- x_3

Der Term f_4^0 stellt eine Konstante dar.

Wir vereinbaren wieder die vereinfachenden Schreibweisen wie in der Aussagenlogik. Außerdem legen wir fest:

- u, v, w, x, y, z stehen für Variablen.
- a, b, c stehen für Konstanten.
- f, g, h stehen für Funktionssymbole, die Stelligkeit geht immer aus dem Kontext hervor.
- P, Q, R stehen für Prädikatsymbole, auch hier geht die Stelligkeit aus dem Kontext hervor.

Übung 47. Geben Sie sämtliche Teilformeln und Terme an, die in der folgenden Formel enthalten sind:

$$F = ((Q(x) \vee \exists x \forall y\ (P(f(x), z) \wedge Q(a))) \vee \forall z\ R(x, z, g(x)))$$

Welche Teilformeln sind Aussagen? Bestimmen Sie für jedes Vorkommen einer Variablen, ob die Variable frei oder gebunden ist.

Semantik der Prädikatenlogik: Eine *Struktur* ist ein Paar $\mathcal{A} = (U_\mathcal{A}, I_\mathcal{A})$, wobei $U_\mathcal{A}$ eine beliebige aber nicht leere Menge ist, die die *Grundmenge* von $\mathcal{A}$, der *Individuenbereich* oder auch *Universum* genannt wird.

Ferner ist $I_\mathcal{A}$ eine Abbildung, die

- jedem im Definitionsbereich von $I_\mathcal{A}$ liegenden k-stelligen Prädikatsymbol P ein k-stelliges Prädikat über $U_\mathcal{A}$ zuordnet,
- jedem im Definitionsbereich von $I_\mathcal{A}$ liegenden k-stelligen Funktionssymbol f eine k-stellige Funktion auf $U_\mathcal{A}$ zuordnet,
- jeder Variablen x (sofern $I_\mathcal{A}$ auf x definiert ist) ein Element der Grundmenge $U_\mathcal{A}$ zuordnet.

Wir schreiben abkürzend statt $I_\mathcal{A}(P)$ einfach $P^\mathcal{A}$, statt $I_\mathcal{A}(f)$ einfach $f^\mathcal{A}$ und statt $I_\mathcal{A}(x)$ einfach $x^\mathcal{A}$.

Sei F eine Formel und $\mathcal{A} = (U_\mathcal{A}, I_\mathcal{A})$ eine Struktur. $\mathcal{A}$ heißt zu F *passend*, falls $I_\mathcal{A}$ für alle in F vorkommenden Prädikatsymbole, Funktionssymbole und freien Variablen definiert ist.

Beispiel: $F = \forall x\ P(x, f(x)) \wedge Q(g(a, z))$ ist eine Formel. Eine zu F passende Struktur ist bspw. $\mathcal{A} = (U_\mathcal{A}, I_\mathcal{A})$ mit

$$
\begin{aligned}
U_\mathcal{A} &= \{0, 1, 2, 3, \ldots\} = \mathbb{N} \\
I_\mathcal{A}(P) &= P^\mathcal{A} = \{(m, n) \mid m, n \in U_\mathcal{A} \text{ und } m < n\} \\
I_\mathcal{A}(Q) &= Q^\mathcal{A} = \{n \in U_\mathcal{A} \mid n \text{ ist Primzahl}\} \\
I_\mathcal{A}(f) &= f^\mathcal{A} \text{ mit } f^\mathcal{A}(n) := n + 1 \\
I_\mathcal{A}(g) &= g^\mathcal{A} \text{ mit } g^\mathcal{A}(m, n) := m + n \\
I_\mathcal{A}(a) &= a^\mathcal{A} = 2 \\
I_\mathcal{A}(z) &= z^\mathcal{A} = 3
\end{aligned}
$$

Mit anderen Worten: Wir haben Interpretationen der Prädikate und Funktionen angegeben und eine Belegung der freien Variablen und der Konstanten festgelegt. Wir wissen jetzt,

- was die Prädikate P und Q bedeuten,
- wie die Funktionen f und g definiert sind,
- das die Konstante a den Wert 2 hat,
- das die Variable x den Wert 3 hat
- und auf welcher Grundmenge das alles definiert ist.

Intuitiv:

- Jede natürliche Zahl ist kleiner als ihr Nachfolger und die Summe von 2 und 3 ist eine Primzahl.

$\rightarrow$ In dieser Struktur gilt die Formel F.

- F ist nicht notwendigerweise in jeder Struktur gültig!

Wir wollen den Begriff *gültig* nun formal definieren.

Sei F eine Formel und $\mathcal{A}$ eine zu F passende Struktur. Für jeden Term t aus Variablen und Funktionssymbolen von F definieren wir induktiv den *Wert* von t in der Struktur $\mathcal{A}$, den wir mit $\mathcal{A}(t)$ bezeichnen:

- Falls t eine Variable ist (also $t = x$), so ist $\mathcal{A}(t) = x^{\mathcal{A}}$.
- Falls t die Form $t = f(t_1, \ldots, t_k)$ hat, wobei $t_1, \ldots, t_k$ Terme und f ein k-stelliges Funktionssymbol ist, so ist

$$\mathcal{A}(t) = f^{\mathcal{A}}(\mathcal{A}(t_1), \ldots, \mathcal{A}(t_k)).$$

  Dieser Fall schließt auch die Möglichkeit ein, dass f nullstellig ist, also t die Form $t = a$ hat. Dann ist $\mathcal{A}(t) = a^{\mathcal{A}}$.

Analog definieren wir induktiv den (Wahrheits-)*Wert* der Formel F unter der Struktur $\mathcal{A}$, wobei wir ebenfalls die Bezeichnung $\mathcal{A}(F)$ verwenden.

- Falls F die Form $F = P(t_1, \ldots, t_k)$ hat mit den Termen $t_1, \ldots, t_k$ und einem k-stelligen Prädikatsymbol P, so gilt:

$$\mathcal{A}(F) = \begin{cases} 1, & \text{falls } (\mathcal{A}(t_1), \ldots, \mathcal{A}(t_k)) \in P^{\mathcal{A}} \\ 0, & \text{sonst} \end{cases}$$

- Falls F die Form $F = \neg G$ hat, so gilt:

$$\mathcal{A}(F) = \begin{cases} 1, & \text{falls } \mathcal{A}(G) = 0 \\ 0, & \text{sonst} \end{cases}$$

- Falls F die Form $F = (G \wedge H)$ hat, so gilt:

$$\mathcal{A}(F) = \begin{cases} 1, & \text{falls } \mathcal{A}(G) = 1 \text{ und } \mathcal{A}(H) = 1 \\ 0, & \text{sonst} \end{cases}$$

- Falls F die Form $F = (G \vee H)$ hat, so gilt:

$$\mathcal{A}(F) = \begin{cases} 1, & \text{falls } \mathcal{A}(G) = 1 \text{ oder } \mathcal{A}(H) = 1 \\ 0, & \text{sonst} \end{cases}$$

- Falls F die Form $F = \forall x\ G$ hat, so gilt:

$$\mathcal{A}(F) = \begin{cases} 1, & \text{falls für alle } d \in U_{\mathcal{A}} \text{ gilt: } \mathcal{A}_{[x/d]}(G) = 1 \\ 0, & \text{sonst} \end{cases}$$

- Falls F die Form $F = \exists x\ G$ hat, so gilt:

$$\mathcal{A}(F) = \begin{cases} 1, & \text{falls es ein } d \in U_{\mathcal{A}} \text{ gibt mit: } \mathcal{A}_{[x/d]}(G) = 1 \\ 0, & \text{sonst} \end{cases}$$

$\mathcal{A}_{[x/d]}$ bezeichne die Struktur $\mathcal{A}'$, die mit $\mathcal{A}$ identisch ist, bis auf die Definition von $x^{\mathcal{A}'}$: Es sei $x^{\mathcal{A}'} = d$, wobei $d \in U_{\mathcal{A}} = U_{\mathcal{A}'}$.

Man könnte auch sagen: x wird mit dem Wert d belegt.

Analog zur Aussagenlogik definieren wir:

- Falls $\mathcal{A}(F) = 1$ für eine Formel F und einer zu F passenden Struktur $\mathcal{A}$ gilt, so schreiben wir $\mathcal{A} \models F$ und sagen: F *gilt* in $\mathcal{A}$ oder auch $\mathcal{A}$ ist *Modell* für F.
- Falls jede zu F passende Struktur ein Modell für F ist, so schreiben wir $\models F$ und sagen F ist (allgemein-)*gültig*.
- Falls es ein Modell für die Formel F gibt, so heißt F *erfüllbar*, andernfalls *unerfüllbar*.
- Weitere Begriffe wie *Folgerung* und *Äquivalenz* können aus der Aussagenlogik übertragen werden.

Analog zum aussagenlogischen Fall lässt sich zeigen, dass für jede Formel F gilt:

F ist genau dann (allgemein-)gültig, wenn $\neg F$ unerfüllbar ist.

Wir schieben für einen Augenblick die präzise mathematische Definition beiseite, damit vielleicht klarer wird, was wir eigentlich gerade tun.

Ersetzt man in einer Aussage A eine Konstante durch eine Variable x, so entsteht die *Aussageform* $A(x)$, auch *Prädikat* $A(x)$ genannt.

- Krefeld liegt am Rhein $\rightarrow$ x liegt am Rhein
- 5 ist eine natürliche Zahl $\rightarrow$ y ist eine natürliche Zahl

Häufig hat man weitere Variablen und zusätzlich auch Terme, die über Variablen gebildet werden, z.B.

- 7 ist kleiner als 12 $\rightarrow$ x ist kleiner als y

$\rightarrow$ Wir erhalten ein Prädikat der Form $A(x, y)$.

- 7 plus 3 ist kleiner als 12 $\rightarrow$ $x + y$ ist kleiner als z

$\rightarrow$ Wir erhalten ein Prädikat der Form $A(t(x, y), z)$.

- Eine Aussageform/ein Prädikat hat im Allgemeinen keinen bestimmten Wahrheitswert.
- Erst wenn die Variable durch einen konkreten Wert ersetzt wird, entsteht eine Aussage, die einen Wahrheitswert hat.

Aus der Aussagenlogik wird durch die Aussageformen die *Prädikatenlogik*. Im Folgenden einige Beispiele dazu.

Aussageform: $A(x) := x^2 > 30$

- $A(x)$ wird für $x = 6$ zur wahren Aussage.
- Für $x = 5$ ist $A(x)$ falsch.
- $\neg A(x)$ lautet $x^2 \leq 30$.

Aussageform: $A(x) := x$ ist eine Ampelfarbe

- Für $x \in \{\text{rot, gelb, grün}\}$ wird $A(x)$ wahr.
- Für $x = \text{blau}$ wird $A(x)$ zu einer falschen Aussage.

Allquantor

Der *Allquantor* $\forall$ steht für den Text „für alle“.

- $\forall x \in E : A(x)$ ist die Aussage, die in Textform lautet:
  „Für alle Elemente x von E gilt: Die Aussageform $A(x)$ wird eine wahre Aussage.“
  Oder anders formuliert: $A(x)$ wird für jedes $x \in E$ wahr.

Beispiele für wahre Aussagen:

- $\forall x \in \{\text{rot, gelb, grün}\} : x$ ist eine Ampelfarbe
- $\forall x \in \{-3, -2, -1, 0, 1, 2, 3\} : (x^2 = 4 \iff x = 2 \vee x = -2)$

Existenzquantor

Der *Existenzquantor* $\exists$ steht für den Text „es existiert“.

- $\exists x \in E : A(x)$ ist die Aussage, die in Textform lautet:

  „Es existiert (mindestens) ein Element x von E, so dass $A(x)$ für dieses Element x wahr wird.“

  Oder anders formuliert: $A(x)$ wird für (mindestens) ein $x \in E$ wahr.

Beispiele für wahre Aussagen:

- $\exists x \in \{\text{blau, gelb, grün}\} : x$ ist eine Ampelfarbe
- $\exists x \in \{1, 2, 3\} : x^2 = 4$

Übung 48. Das Prädikat $mutter(a, b)$ bedeute, dass a Mutter von b ist. Beschreiben Sie umgangssprachlich die folgenden Formeln:

- $\forall y\ \exists x\ mutter(x, y)$
- $\exists x\ \forall y\ mutter(x, y)$

Übung 49. Drücken Sie folgende Sachverhalte mittels Prädikaten aus:

- Alle Studenten sind arm.
- Alle grünen Pilze sind giftig.
- Es gibt einen armen Studenten.
- Wer schneller als Anna ist, ist auch schneller als Klaus.
- Niemand ist schneller als Harald.
- Claudia ist die schnellste.

Übung 50. Formulieren Sie das Kannibalen und Missionare-Problem mittels Prädikatenlogik:

- 3 Missionare und 3 Kannibalen wollen einen Fluss überqueren.
- Es steht nur ein Ruderboot für die Überfahrt zur Verfügung.
- Das Boot kann maximal 2 Personen aufnehmen.
- Es dürfen nie mehr Kannibalen als Missionare an einem Ort sein, sonst gibt es ein großes Fressen.

Übersicht

Zwei prädikatenlogische Formeln F und G heißen *äquivalent*, falls für alle sowohl zu F als auch zu G passenden Strukturen $\mathcal{A}$ gilt: $\mathcal{A}(F) = \mathcal{A}(G)$

Alle in der Aussagenlogik gezeigten Äquivalenzen gelten auch in der Prädikatenlogik:

- Idempotenz
- Kommutativität
- Assoziativität
- Absorption
- Distributivität
- Doppelnegation
- deMorgansche Regeln
- usw.

Um auch prädikatenlogische Formeln in Normalformen umformen zu können, benötigen wir noch Äquivalenzen, die Quantoren mit einbeziehen.

Seien F und G beliebige Formeln. Dann gilt:

1.a $\neg\forall x\ F \ \equiv\ \exists x\ \neg F$

Beispiel: „Nicht alle Fußballspiele der 1. Bundesliga am vergangenen Spieltag endeten unentschieden.“

Ist äquivalent zu: „Es gab am vergangenen Spieltag der 1. Bundesliga (mindestens) ein Fußballspiel, das nicht unentschieden endete.“

Beispiel: „Nicht alle Außerirdischen haben grüne Haut.“

Ist äquivalent zu: „Es gibt Außerirdische, die *keine* grüne Haut haben.“

1.b $\neg\exists x\ F \ \equiv\ \forall x\ \neg F$

Beispiel: „Es gab kein Fußballspiel der 1. Bundesliga, das am vergangenen Spieltag unentschieden endete.“

Ist äquivalent zu: „Alle Fußballspiele der 1. Bundesliga des vergangenen Spieltags endeten nicht unentschieden.“

2. Falls x in G nicht frei vorkommt, gilt:

$$\begin{aligned}(\forall x\ F \wedge G) &\equiv \forall x\ (F \wedge G)\\ (\forall x\ F \vee G) &\equiv \forall x\ (F \vee G)\\ (\exists x\ F \wedge G) &\equiv \exists x\ (F \wedge G)\\ (\exists x\ F \vee G) &\equiv \exists x\ (F \vee G)\end{aligned}$$

3. $(\forall x\ F \wedge \forall x\ G) \equiv \forall x\ (F \wedge G)$
   $(\exists x\ F \vee \exists x\ G) \equiv \exists x\ (F \vee G)$
4. $\forall x \forall y\ F \equiv \forall y \forall x\ F$
   $\exists x \exists y\ F \equiv \exists y \exists x\ F$

Übung 51. Formulieren Sie Beispiele für obige Aussagen 2 bis 4.

Übung 52. Machen Sie sich klar, dass obige zweite Aussage auch einen Fall wie $G = \forall y\ \exists z\ Q(y, z)$ einschließt und welche Folgen das hat.

Exemplarisch beweisen wir: $(\forall x\ F \wedge G) \ \equiv\ \forall x\ (F \wedge G)$

$\mathcal{A}(\forall x\ F \wedge G) = 1$

$\Longleftrightarrow$ $\mathcal{A}(\forall x\ F) = 1$ und $\mathcal{A}(G) = 1$

$\Longleftrightarrow$ Für alle $d \in U_{\mathcal{A}}$ gilt $\mathcal{A}_{[x/d]}(F) = 1$ und $\mathcal{A}_{[x/d]}(G) = 1$, denn weil x in G nicht frei vorkommt, gilt $\mathcal{A}(G) = \mathcal{A}_{[x/d]}(G)$.

$\Longleftrightarrow$ Für alle $d \in U_{\mathcal{A}}$ gilt: $\mathcal{A}_{[x/d]}((F \wedge G)) = 1$

$\Longleftrightarrow$ $\mathcal{A}(\forall x\ (F \wedge G) = 1$

Übung 53. Zeigen Sie, dass folgende Formelpaare *nicht* äquivalent sind, indem Sie Gegenbeispiele angeben:

$$(\forall x\ F\ \vee\ \forall x\ G)\ \not\equiv\ \forall x\ (F \vee G)$$
$$(\exists x\ F\ \wedge\ \exists x\ G)\ \not\equiv\ \exists x\ (F \wedge G)$$

Übung 54. Bei verschiedenen Quantoren ist die Reihenfolge extrem wichtig. Es gilt folgende *Nicht-Äquivalenz*:

$$\forall x\ \exists y\ F\ \not\equiv\ \exists y\ \forall x\ F$$

Formulieren Sie ein Beispiel, das den Sachverhalt deutlich macht.

Übung 55. Zeigen Sie, dass $\forall x\ \exists y\ P(x, y)$ eine Folgerung von $\exists y\ \forall x\ P(x, y)$ ist, aber nicht umgekehrt.

Durch obige Äquivalenzumformungen können wir die in einer Formel evtl. vorhandenen Quantoren an den Anfang der Formel verschieben:

$$
\begin{aligned}
&(\neg(\exists x\ P(x,y) \vee \forall z\ Q(z)) \wedge \exists w\ P(f(a,w))) \\
&\quad\equiv\ ((\neg\exists x\ P(x,y) \wedge \neg\forall z\ Q(z)) \wedge \exists w\ P(f(a,w))) && \text{deMorgan} \\
&\quad\equiv\ ((\forall x\ \neg P(x,y) \wedge \exists z\ \neg Q(z)) \wedge \exists w\ P(f(a,w))) && \text{wegen 1.} \\
&\quad\equiv\ (\exists w\ P(f(a,w)) \wedge (\forall x\ \neg P(x,y) \wedge \exists z\ \neg Q(z))) && \text{kommutativ} \\
&\quad\equiv\ \exists w\ (P(f(a,w)) \wedge \forall x\ (\neg P(x,y) \wedge \exists z\ \neg Q(z))) && \text{wegen 2.} \\
&\quad\equiv\ \exists w\ (\forall x\ (\exists z\ \neg Q(z) \wedge \neg P(x,y)) \wedge P(f(a,w))) && \text{kommutativ} \\
&\quad\equiv\ \exists w\ (\forall x\ \exists z\ (\neg Q(z) \wedge \neg P(x,y)) \wedge P(f(a,w))) && \text{wegen 2.} \\
&\quad\equiv\ \exists w\ \forall x\ \exists z\ ((\neg Q(z) \wedge \neg P(x,y)) \wedge P(f(a,w))) && \text{wegen 2.}
\end{aligned}
$$

Punkt 2 des obigen Satzes kann nicht immer angewendet werden:

$$\forall x\ P(x,y)\ \wedge\ Q(x)\ \not\equiv\ \forall x\ \big(P(x,y)\ \wedge\ Q(x)\big)$$

Um die Regel 2 dennoch anwenden zu können, benennen wir zunächst die freie Variable x um:

$$\begin{aligned}\forall x\ P(x,y)\ \wedge\ Q(x)\ &\equiv\ \forall x\ P(x,y)\ \wedge\ Q(z)\\ &\equiv\ \forall x\ \big(P(x,y)\ \wedge\ Q(z)\big)\end{aligned}$$

Ein solches Umbenennen von Variablen heißt *Substitution*. Sei F eine Formel, x eine Variable und t ein Term.

> Dann bezeichnet $F[x/t]$ diejenige Formel, die man aus F erhält, indem jedes freie Vorkommen der Variablen x in F durch den Term t ersetzt wird.

Anmerkung: Wir müssen zwischen $\mathcal{A}_{[x/d]}$ und $F[x/t]$ deutlich unterscheiden. Im ersten Fall erhalten wir eine neue Struktur, bei der x den Wert d erhält (Semantik), im zweiten Fall erhalten wir eine neue Formel, bei der x durch t ersetzt ist (Syntax).

Lemma: (gebundene Umbenennung)

Falls die Variable y nicht in der Formel G vorkommt, gilt:

$$\begin{aligned} \forall x\ G &\equiv \forall y\ G[x/y] \\ \exists x\ G &\equiv \exists y\ G[x/y] \end{aligned}$$

Beispiel: Aus

$$\forall x\ P(x) \ \wedge\ Q(x, y)$$

wird durch gebundene Umbenennung die äquivalente Formel

$$\forall z\ P(z) \ \wedge\ Q(x, y).$$

Durch systematisches Ausführen von gebundenen Umbenennungen, wobei immer neue, noch nicht vorkommende Variablen verwendet werden, kann man das folgende Lemma zeigen.

Lemma: Zu jeder Formel F gibt es eine äquivalente Formel G in bereinigter Form.

Eine Formel heißt *bereinigt*, sofern es keine Variable gibt, die in der Formel sowohl frei als auch gebunden vorkommt, und sofern hinter allen vorkommenden Quantoren verschiedene Variablen stehen.

Beispiel:

$$\begin{aligned} & \forall x\ \exists y\ P(x, f(y)) \ \wedge\ \forall y\ (Q(x, y) \vee R(x)) \\ \equiv\ & \forall u\ \exists y\ P(u, f(y)) \ \wedge\ \forall y\ (Q(x, y) \vee R(x)) \\ \equiv\ & \forall u\ \exists v\ P(u, f(v)) \ \wedge\ \forall y\ (Q(x, y) \vee R(x)) \end{aligned}$$

Freie Variablen: x

Gebundene Variablen: u, v, y

Eine Formel heißt *pränex* oder in *Pränexform*, falls sie die Bauart

$$Q_1 y_1 \; Q_2 y_2 \; \ldots Q_n y_n \; F$$

hat, wobei $Q_i \in \{\exists, \forall\}$, $n \geq 0$, und die y_i Variablen sind. Ferner kommt kein Quantor in F vor.

Satz: Für jede Formel F gibt es eine äquivalente und bereinigte Formel G in Pränexform.

Beispiel:

$$\begin{aligned}
& \forall x \; \exists y \; [P(x, g(y, f(x))) \; \vee \; \neg Q(z)] \; \vee \; \neg \forall x \; R(x, y) \\
\equiv \; & \forall x \; \exists y \; [P(x, g(y, f(x))) \; \vee \; \neg Q(z)] \; \vee \; \exists x \neg \; R(x, y) \\
\equiv \; & \forall x \; \exists y \; [P(x, g(y, f(x))) \; \vee \; \neg Q(z)] \; \vee \; \exists v \neg \; R(v, y) \\
\equiv \; & \forall x \; \exists u \; [P(x, g(u, f(x))) \; \vee \; \neg Q(z)] \; \vee \; \exists v \neg \; R(v, y) \\
\equiv \; & \forall x \; \exists u \; \exists v \; [P(x, g(u, f(x))) \; \vee \; \neg Q(z) \; \vee \; \neg \; R(v, y)]
\end{aligned}$$

Beweis: Induktion über Formelaufbau.

Wenn F eine atomare Formel ist, so liegt F bereits in Pränexform. Für den Induktionsschluss unterscheiden wir verschiedene Fälle.

- F hat die Form $\neg F_1$ und $G_1 = Q_1 y_1 \ Q_2 y_2 \ \ldots Q_n y_n \ G'$ sei die zu F_1 äquivalente Formel, die nach Induktionsvoraussetzung existiert. Dann gilt

$$F \equiv \overline{Q_1} y_1 \ \overline{Q_2} y_2 \ \ldots \overline{Q_n} y_n \ \neg G'$$

  mit $\overline{Q_i} = \exists$, falls $Q_i = \forall$, und $\overline{Q_i} = \forall$, falls $Q_i = \exists$.
- F hat die Form $(F_1 \circ F_2)$ mit $\circ \in \{\wedge, \vee\}$. Seien G_1 und G_2 die zu F_1 und F_2 äquivalenten Formeln in Pränexform. Durch gebundenes Umbenennen erhalten wir

$$G_1 = Q_1 y_1 \ Q_2 y_2 \ \ldots Q_n y_n \ G_1' \quad \text{und} \quad G_2 = Q_1' z_1 \ Q_2' z_2 \ \ldots Q_m' z_m \ G_2'$$

  Formeln, deren gebundene Variablen disjunkt sind mit $Q_i, Q_j' \in \{\exists, \forall\}$. Dann gilt:

$$F \equiv Q_1 y_1 \ Q_2 y_2 \ \ldots Q_n y_n \ Q_1' z_1 \ Q_2' z_2 \ \ldots Q_m' z_m \ (G_1' \circ G_2')$$

- F hat die Form $Qx \ F_1$ mit $Q \in \{\exists, \forall\}$ und $F_1 = Q_1 y_1 \ Q_2 y_2 \ \ldots Q_n y_n \ F_1'$ und x ist disjunkt zu $y_1, \ldots, y_n$. Dann gilt:

$$F \equiv Qx \ Q_1 y_1 \ Q_2 y_2 \ \ldots Q_n y_n \ F_1'$$

Für die prädikatenlogische Variante der Resolution müssen wir die Formeln noch weiter vereinfachen: Zunächst werden freie Variablen durch Einführen von Existenzquantoren gebunden: $\forall x\ P(x, y)$ ist erfüllbarkeitsäquivalent zu $\forall x\ \exists y\ P(x, y)$.

Schließlich werden Existenzquantoren durch *Skolemisierung* eliminiert.

- Tritt die Form $\exists x$ nicht im Geltungsbereich eines Allquantors auf, wird $\exists x$ einfach weggelassen und alle Auftreten von x werden durch eine neue Konstante c ersetzt.

  Beispiel: Aus $\exists x\ P(x)$ wird $P(c)$.

  Beispiel: Aus $\exists x\ \exists z\ \forall y\ P(y, x, z)$ wird $\forall y\ P(y, c_1, c_2)$.
- Tritt dagegen $\exists x$ im Geltungsbereich der allquantifizierten Variablen $y_1, \ldots, y_k$ auf, so werden alle Auftreten von x durch den Term $f(y_1, \ldots, y_k)$ ersetzt, wobei f ein neues Funktionssymbol sein muss.

  Beispiel: Aus $\forall y\ \exists x\ P(x, y)$ wird $\forall y\ P(f(y), y)$.

  Beispiel: Aus $\exists x\ \forall y\ \exists z\ P(y, x, z)$ wird $\forall y\ P(y, c, f(y))$.

Die vollständige Skolemisierung einer Formel F entfernt alle Existenzquantoren aus F und liefert eine Formel F', die erfüllbarkeitsäquivalent zu F ist, d.h. F ist genau dann erfüllbar, wenn F' erfüllbar ist.

Übung 56. Geben Sie die Skolemform der folgenden Formel an:

$$F = \forall x\ \exists y\ \forall z\ \exists w\ (\neg P(a, w) \vee Q(f(x), y))$$

Übung 57. Wenden Sie alle vorgestellten Umformungsschritte (bereinigen, Pränexform, Skolemform) an auf die Formel:

$$F = \forall z\ \exists y\ (P(x, g(y), z) \vee \neg\forall x\ Q(x)) \wedge \neg\forall z\ \exists x\ \neg R(f(x, z), z)$$

Zusammenfassung

Gegeben sei eine prädikatenlogische Formel F.

- Bereinige F durch systematisches Umbenennen der gebundenen Variablen.
  $\rightarrow$ Es entsteht eine zu F äquivalente Formel F_1.
- Binde die in F_1 vorkommenden freien Variablen $y_1, \ldots, y_k$:
  Ersetze F_1 durch $F_2 = \exists y_1\ \exists y_2 \ldots \exists y_k\ F_1$.
  $\rightarrow$ Dann ist F_2 erfüllbarkeitsäquivalent zu F_1.
- Stelle eine zu F_2 äquivalente Formel F_3 in Pränexform her.
  $\rightarrow$ F_3 ist erfüllbarkeitsäquivalent zu F.
- Eliminiere Existenzquantoren aus F_3 durch Skolemisierung und erhalte Formel F_4.
  $\rightarrow$ F_4 ist erfüllbarkeitsäquivalent zu F_3 und damit auch zu F.
- Da alle auftretenden Variablen jetzt allquantifiziert sind, können (zur Darstellung) alle Allquantoren weggelassen werden.
- Die so erhaltene Formel kann durch entsprechende Umformungen in disjunktive oder konjunktive Normalform überführt werden und in *Klauselform* dargestellt werden.

Übersicht

Wir suchen einen algorithmischen Test für die Erfüllbarkeit oder Gültigkeit von prädikatenlogischen Formeln.

Gültigkeitsproblem der Prädikatenlogik:

gegeben: Eine prädikatenlogische Formel F.
gefragt: Ist F eine (allgemein-)gültige Formel?

Erfüllbarkeitsproblem der Prädikatenlogik:

gegeben: Eine prädikatenlogische Formel F.
gefragt: Ist F erfüllbar?

Leider können solche Algorithmen aus prinzipiellen Gründen nicht existieren, wie wir im Folgenden sehen werden.

Postsches Korrespondenzproblem:

gegeben: Eine endliche Menge von Wortpaaren (x_1, y_1), $(x_2, y_2), \ldots, (x_k, y_k)$, wobei $x_i, y_i \in \Sigma^+$.

gefragt: Gibt es eine Folge von Indizes $i_1, i_2, \ldots, i_n \in \{1, 2, \ldots, k\}$, $n \geq 1$, mit $x_{i_1} x_{i_2} \ldots x_{i_n} = y_{i_1} y_{i_2} \ldots y_{i_n}$?

Beispiel: $K = ((1, 101), (10, 00), (011, 11))$ besitzt die Lösung (1,3,2,3), denn es gilt

$$\underbrace{1}_{x_1}\underbrace{011}_{x_3}\underbrace{10}_{x_2}\underbrace{011}_{x_3} = \underbrace{101}_{y_1}\underbrace{11}_{y_3}\underbrace{00}_{y_2}\underbrace{11}_{y_3}$$

Aus der Berechenbarkeitstheorie wissen wir, dass das Postsche Korrespondenzproblem nicht entscheidbar ist.

Mittels Reduktion kann man zeigen, dass das Gültigkeitsproblem für prädikatenlogische Formeln nicht entscheidbar ist.

Als Folgerung ergibt sich, dass auch das Erfüllbarkeitsproblem für prädikatenlogische Formeln nicht entscheidbar ist. (Eine Formel F ist gültig, genau dann wenn $\neg F$ unerfüllbar ist.)

Monadische Prädikatenlogik: Die Formeln enthalten keine Funktionssymbole und alle Prädikatsymbole sind einstellig.

- Für eine erfüllbare Aussage F der monadischen Prädikatenlogik mit den einstelligen Prädikatensymbolen $P_1, \ldots, P_n$ existiert bereits ein Modell der Mächtigkeit 2^n.
- Das Erfüllbarkeits- und das Gültigkeitsproblem für monadische prädikatenlogische Formeln ist entscheidbar.

Übersicht

Problem im Umgang mit prädikatenlogischen Formeln:

- Die Definition von Strukturen $\mathcal{A} = (U_\mathcal{A}, I_\mathcal{A})$ lässt es zu, dass $U_\mathcal{A}$ beliebige Mengen sein können.
- Es scheint keine Möglichkeit zu geben, Aussagen über deren Mächtigkeit, geschweige denn über den Aufbau von $I_\mathcal{A}$ zu machen.

$\rightarrow$ Wie sollen wir alle potenziellen Strukturen für eine gegebene Formel auf systematische Art und Weise aufzählen, um sie daraufhin auf ihre Modelleigenschaft zu untersuchen?

Arbeiten, die im wesentlichen auf Jacques Herbrand, Kurt Gödel und Thoralf Skolem zurückgehen, zeigen: Die algorithmische Suche nach potenziellen Modellen für eine Formel kann auf eine gewisse kanonische Art erfolgen bzw. beschränkt bleiben.

Sei F eine geschlossene Formel in Skolemform. Das *Herbrand-Universum* $D(F)$ wird wie folgt induktiv definiert:

- $D(F)$ enthält alle in F vorkommenden Konstanten. Falls F keine Konstanten enthält, so ist a in $D(F)$.
- Für jedes in F vorkommende n-stellige Funktionssymbol f und Terme $t_1, t_2, \ldots, t_n$ in $D(F)$ ist auch der Term $f(t_1, t_2, \ldots, t_n)$ in $D(F)$.

Also: Das Herbrand-Universum besteht aus allen variablenfreien Termen, die aus den Konstanten und den Funktionssymbolen der Formel F gebildet werden können.

Beispiel: Das Herbrand-Universum zur Formel $F = \forall x\ \forall y\ \forall z\ P(x, f(y), g(z, x))$ ist:

$$\begin{aligned} D(F) \quad = \quad & \{a, f(a), g(a,a), f(f(a)), f(g(a,a)), \\ & g(f(a), a), g(a, f(a)), g(f(a), f(a)), \ldots\} \end{aligned}$$

Übung 58. Geben Sie für die Formel $G = \forall x\ \forall y\ Q(c, f(x), h(y, b))$ die ersten Elemente des Herbrand-Universums an.

Wir werden im Folgenden sehen, dass es ausreichend ist, $D(F)$ als Grundbereich zu nutzen, um nach möglichen Modellen für die Formel F zu suchen.

Sei F eine Aussage in Skolemform. Dann heißt jede zu F passende Struktur $\mathcal{A} = (U_\mathcal{A}, I_\mathcal{A})$ eine *Herbrand-Struktur* für F, falls gilt:

- $U_\mathcal{A} = D(F)$
- Für jedes in F vorkommende n-stellige Funktionssymbol f und $t_1, t_2, \ldots, t_n \in D(F)$ ist $f^\mathcal{A}(t_1, t_2, \ldots, t_n) = f(t_1, t_2, \ldots, t_n)$.

Beispiel: Eine Herbrand-Struktur $\mathcal{A} = (U_\mathcal{A}, I_\mathcal{A})$ für die Formel $F = \forall x\ \forall y\ \forall z\ P(x, f(y), g(z, x))$ muss folgende Bedingungen erfüllen:

$$U_\mathcal{A} = D(F) = \{a, f(a), g(a,a), f(f(a)), f(g(a,a)), \ldots\}$$

und

$$\begin{array}{rcl} f^\mathcal{A}(a) & = & f(a) \\ f^\mathcal{A}(f(a)) & = & f(f(a)) \\ f^\mathcal{A}(g(a,a)) & = & f(g(a,a)) \end{array}$$

usw.

$$\begin{array}{rcl} g^\mathcal{A}(a,a) & = & g(a,a) \\ g^\mathcal{A}(f(a),a) & = & g(f(a),a) \\ g^\mathcal{A}(g(a,a)) & = & g(g(a,a)) \end{array}$$

usw.

Die Wahl von $P^\mathcal{A}$ ist noch offen. Wir könnten festlegen:

$$(t_1, t_2, t_3) \in P^\mathcal{A} \ :\Longleftrightarrow\ g(t_1, t_2) = g(t_2, f(t_3))$$

Ist die so definierte Struktur $\mathcal{A}$ ein Modell für F?

Anmerkungen:

- Bei Herbrand-Strukturen sind der Grundbereich und die Interpretation der Funktionssymbole festgelegt. Nur die Interpretation der Prädikatsymbole kann frei gewählt werden.
- Syntax und Semantik von Termen wird „gleichgeschaltet“. Terme werden „durch sich selbst“ interpretiert.
- Eine Herbrand-Struktur $\mathcal{A}$ für eine Formel F nennen wir *Herband-Modell* für F, falls $\mathcal{A}$ ein Modell für F ist.

Satz: Sei F eine Aussage in Skolemform. Dann ist F genau dann erfüllbar, wenn F ein Herbrand-Modell besitzt. (ohne Beweis)

Folgerung: Jede erfüllbare Formel der Prädikatenlogik besitzt bereits ein abzählbares Modell, also ein Modell mit abzählbarer Grundmenge.

Sei $F = \forall y_1\ \forall y_2 \ldots \forall y_n\ F^*$ eine Aussage in Skolemform. Dann ist $E(F)$, die *Herbrand-Expansion* von F, definiert als

$$E(F) = \{F^*[y_1/t_1][y_2/t_2]\cdots[y_n/t_n] \mid t_1, t_2, \ldots, t_n \in D(F)\}.$$

Die Formeln in $E(F)$ entstehen also, indem die Terme in $D(F)$ in jeder möglichen Weise für die Variablen in F^* substituiert werden.

Beispiel: Für die obige Formel $F = \forall x\ \forall y\ \forall z\ P(x, f(y), g(z, x))$ sind die einfachsten Formeln in $E(F)$ die folgenden:

$$\begin{array}{rcl}
P(a, f(a), g(a, a)) & \text{mit} & [x/a][y/a][z/a] \\
P(f(a), f(a), g(a, f(a))) & \text{mit} & [x/f(a)][y/a][z/a] \\
P(a, f(f(a)), g(a, a)) & \text{mit} & [x/a][y/f(a)][z/a] \\
P(a, f(a), g(f(a), a)) & \text{mit} & [x/a][y/a][z/f(a)] \\
P(g(a, a), f(a), g(a, g(a, a))) & \text{mit} & [x/g(a, a)][y/a][z/a]
\end{array}$$

usw.

Beachte: Die Formeln in $E(F)$ können letztlich als aussagenlogische Formeln behandelt werden, da sie keine Variablen enthalten.

Satz: Für jede Aussage F in Skolemform gilt: F ist erfüllbar genau dann, wenn die Formelmenge $E(F)$ im aussagenlogischen Sinn erfüllbar ist. (ohne Beweis)

Zusammenfassend kann man sagen:

- Eine prädikatenlogische Formel kann durch i.Allg. unendlich viele aussagenlogische Formeln approximiert werden.
- Diese aussagenlogischen Formeln können systematisch aufgezählt werden.

Um daraus ein Semi-Entscheidungsverfahren für die Unerfüllbarkeit von Formeln der Prädikatenlogik zu formulieren, benötigen wir noch den Endlichkeitssatz der Aussagenlogik.

Endlichkeitssatz: (Aussagenlogik)

Eine Menge $\mathcal{F}$ von aussagenlogischen Formeln ist genau dann erfüllbar, wenn jede endliche Teilmenge von $\mathcal{F}$ (also auch $\mathcal{F}$ selbst) erfüllbar ist. (ohne Beweis)

Oder anders formuliert: Eine evtl. unendliche Formelmenge $\mathcal{F}$ ist genau dann unerfüllbar, wenn bereits eine endliche Teilmenge $\mathcal{F}'$ von $\mathcal{F}$ unerfüllbar ist.

Satz von Herbrand:

Eine Aussage F in Skolemform ist genau dann unerfüllbar, wenn es eine endliche Teilmenge von $E(F)$ gibt, die im aussagenlogischen Sinn unerfüllbar ist. (direkte Folgerung aus obigen Sätzen)

Algorithmus von Gilmore: Semi-Entscheidungsverfahren für die Unerfüllbarkeit von Formeln der Prädikatenlogik.

- Gegeben: Eine prädikatenlogische Aussage F in Skolemform.
- Sei $E(F) = \{F_1, F_2, F_3, \ldots\}$ eine beliebige aber fixierte Aufzählung der Herbrand-Expansion von F.
- Algorithmus:
  - $n := 0$;
  - repeat $n := n + 1$;
  - until $(F_1 \wedge F_2 \wedge \ldots \wedge F_n)$ ist unerfüllbar

Anmerkungen:

- semi-entscheidbar $\hat{=}$ rekursiv aufzählbar
- Für unerfüllbare Formeln stoppt obiger Algorithmus, für erfüllbare Formeln nicht.
- Durch Eingabe von $\neg F$ anstelle von F wird aus dem Unerfüllbarkeitstest ein Gültigkeitstest.

Übung 59. Zeigen Sie mittels des Algorithmus von Gilmore, dass folgende Formeln unerfüllbar sind:

- $F = \exists x \; \forall y \; (\neg P(f(f(x))) \wedge P(f(y)))$
- $G = \forall x \; (P(f(x)) \wedge \neg P(x))$

Anmerkung: Der Algorithmus von Gilmore funktioniert zwar, ist aber in der Praxis unbrauchbar, weil er zu viele Formeln erzeugt und nicht zielgerichtet arbeitet.

Daher werden wir uns im Folgenden ansehen, wie wir das Prinzip der Resolution auch in der Prädikatenlogik anwenden können.

Übersicht

Die Tests auf Unerfüllbarkeit können auch per aussagenlogischer Resolution erledigt werden.

Beispiel: Gegeben sei folgende unerfüllbare Formel in Skolemform:

$$F = \forall x \ (P(x) \wedge \neg P(f(x)))$$

Damit erhalten wir

- als Klauselform: $\{\{P(x)\},\ \{\neg P(f(x))\}\}$
- als Herbrand-Universum: $D(F) = \{a,\ f(a),\ f(f(a)), \ldots\}$
- als Herbrand-Expansion:
  $E(F) = \{(P(a) \wedge \neg P(f(a))),\ (P(f(a)) \wedge \neg P(f(f(a)))), \ldots\}$, oder in Klauselform
  $\{\{P(a)\},\ \{\neg P(f(a))\},\ \{P(f(a))\},\ \{\neg P(f(f(a)))\}, \ldots\}$

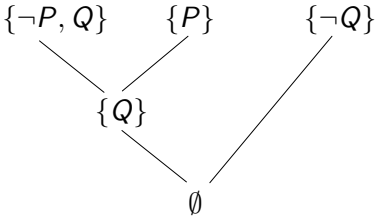
Bereits die ersten beiden Elemente von $E(F)$ reichen aus, um die Unerfüllbarkeit von F zu zeigen:

$$\{P(a)\} \qquad \{\neg P(f(a))\} \qquad \{P(f(a))\} \qquad \{\neg P(f(f(a)))\}$$
$$\searrow \qquad \swarrow$$
$$\emptyset$$

Um besser zu verstehen, was wir hier eigentlich tun, werfen wir kurz einen Blick zurück auf die Aussagenlogik.

Aussagenlogik:

- Betrachten wir die Klauselmenge $\{\{\neg P, Q\}, \{P\}, \{\neg Q\}\}$.
- Resolutionsprinzip: Resolviere Klauseln mit komplementären Literalen.



- Führt der Resolutionsgraph auf die leere Klausel, dann ist die Formel unerfüllbar.

Aber was passiert, wenn wir aus den Variablen P und Q der obigen Formel Prädikate machen?

- Betrachten wir bspw. folgende Formel:

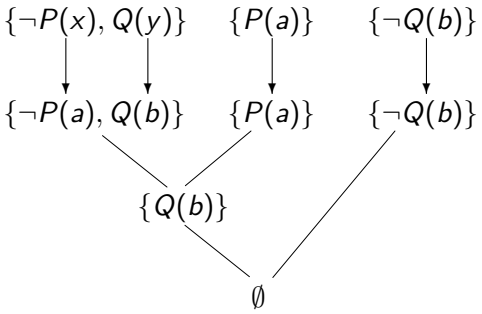
$$\forall x \, \forall y \, (P(x) \rightarrow Q(y)) \wedge P(a) \wedge \neg Q(b)$$

- Als Klauselmenge: $\{\{\neg P(x), Q(y)\}, \{P(a)\}, \{\neg Q(b)\}\}$

Sind $\neg P(x)$ und $P(a)$ komplementär?

- In einem gewissen Sinn: Ja!
- Denn obige Formel soll für alle x gelten, also insbesondere auch für $x := a$.

Im Prinzip suchen wir also Substitutionen (hier $[x/a]$ und $[y/b]$), die komplementäre „Literale“ erzeugen.



Die Herbrand-Theorie zeigt formal die Korrektheit des obigen Vorgehens und erweitert das obige Vorgehen auch auf Funktionen bzw. Terme.

Wir werden später sehen: Es ist nicht notwendig, alle Substitutionen gleich zu Beginn durchzuführen.

Übung 60. Zeigen Sie mittels Resolution, dass $\exists y\ \mathit{irren}(y)$ eine Folgerung ist von:

$$(\forall x\ \mathit{Mensch}(x) \rightarrow \mathit{irren}(x) \wedge \mathit{Mensch}(\mathit{Max}))$$

Oder in Worten: Alle Menschen irren sich und Max ist ein Mensch, also gibt es jemanden, der sich irrt.

Problem:

Wie sollen wir effizient die Substitutionen finden, die bei einer Resolutionsherleitung zur leeren Klausel führen? Ein systematisches Durchprobieren aller Substitutionen scheint sehr aufwendig und wenig zielgerichtet zu sein.

Problem: Es müssen Entscheidungen für einige Substitutionen „vorausschauend“ getroffen werden, um Resolutionen, die im Diagramm weiter unten liegen, zu ermöglichen.

Wir wollen daher nur insoweit Substitutionen durchführen, wie es für den direkt nachfolgenden Resolutionsschritt notwendig ist.

Beispiel: Bei $\{P(x), \neg Q(g(x))\}$ und $\{\neg P(f(y))\}$ reicht die Substitution $[x/f(y)]$, um die Resolvente $\{\neg Q(g(f(y)))\}$ zu erhalten. Es ist nicht nötig, hier bereits y zu substituieren.

Wir suchen also Substitutionen, die zwei Literale (oder mehr) *unifiziert*, also identisch macht. Im obigen Beispiel unifiziert $[x/f(y)]$ die beiden Literale $P(x)$ und $P(f(y))$.

Auch die Substitution $[x/f(a)][y/a]$ würde die beiden Literale unifizieren, aber dabei wird mehr substituiert als notwendig.

Eine Substitution σ ist ein *Unifikator* einer (endlichen) Menge von Literalen $\mathcal{L} = \{L_1, L_2, \ldots, L_k\}$, falls $L_1\sigma = L_2\sigma = \ldots = L_k\sigma$. Wir sagen dann: $\mathcal{L}$ ist *unifizierbar*.

Ein Unifikator σ einer Literalmenge $\mathcal{L}$ heißt *allgemeinster Unifikator* von $\mathcal{L}$, falls für jeden Unifikator μ von $\mathcal{L}$ gilt, dass es eine Substitution s gibt mit $\mu = \sigma s$.

Beispiel: Betrachten wir die Terme $f(x, g(a, y))$ und $f(b, z)$. Dann sind σ und μ zwei Unifikatoren.

$$\mu = [x/b][y/a][z/g(a,a)] \qquad \sigma = [x/b][z/g(a,y)]$$

σ ist allgemeinster Unifikator, μ setzt sich dagegen aus σ und einer anderen Substitution zusammen:

$$\mu = \underbrace{[x/b][z/g(a,a)]}_{\sigma}[y/a]$$

Beispiele: Der allgemeinste Unifikator (most general unifier) von

- $\{P(x), P(f(y))\}$ ist $[x/f(y)]$.
- $\{Q(x, f(x)), Q(y, g(y))\}$ existiert nicht.
- $\{P(x), P(f(x))\}$ existiert nicht.
- $\{Q(x, f(y)), Q(g(z), f(x))\}$ ist $[x/g(z)][y/g(z)]$.

Anmerkungen:

- Der allgemeinste Unifikator zweier Terme ist eindeutig bestimmt (bis auf Variablenumbenennungen).
- Eine Menge $\mathcal{L}$ von (mehr als zwei) Literalen kann nicht unifizierbar sein, auch wenn alle Paare von Literalen in $\mathcal{L}$ unifizierbar sind.
  Beispiel: $\{Q(x, f(y)), Q(f(y), z), Q(z, f(x))\}$

Unifikationsalgorithmus:

Eingabe: eine Literalmenge $\mathcal{L}$
$\sigma := []$; (leere Substitution)
while $|\mathcal{L}\sigma| > 1$ *do*
    Suche die erste Position in $\mathcal{L}\sigma$, an der sich zwei Literale
        L_1, L_2 unterscheiden
    if keines der beiden Zeichen ist eine Variable
    then stoppe mit „nicht unifizierbar“
    else Sei x die Variable und t der Term in anderen Literal
        (möglicherweise auch eine Variable)
        if x kommt in t vor
        then stoppe mit „nicht unifizierbar“
        else $\sigma := \sigma[x/t]$
Ausgabe: σ

Anmerkung: Die Anzahl der Resolutionsmöglichkeiten ist im Allgemeinen sehr hoch, weil

- es viele Möglichkeiten gibt, zwei Klauseln zu finden, die resolvierbar sind, um so neue Resolventen zu erzeugen.
- Resolventen bei einer Resolutionsherleitung länger werden können als ihre Ausgangsklauseln.

Für die Resolution der leeren Klausel eignen sich aber eventuell nur sehr wenige Paare von Klauseln. Wie finden wir also die „richtigen“ Klauseln zum Resolvieren?

Bei der logischen Programmierung beschränken wir uns auf Hornklauseln:

- Jede Programmklausel (also eine Regel oder ein Fakt) enthält genau ein positives Literal (definite Klausel).
- Eine Zielklausel (Anfrage an das System) enthält nur negative Literale.

Das hat weitreichende Folgen für die Ableitungsmöglichkeiten und die dabei erzielbare Effizienz:

- Resolvieren zweier Programmklauseln kann niemals zur leeren Klausel führen.
- Die Resolution einer Zielklausel und einer Programmklausel führt immer wieder zu einer neuen (evtl. leeren) Zielklausel.

$\rightarrow$ SLD-Resolution: Selection rule driven Linear resolution for Definite clauses

Linear resolution bedeutet, dass mit der gerade hergeleiteten Klausel weitergearbeitet wird.

Beispiel: Das Prädikat $P(x, y)$ bedeute „x is parent of y“. Außerdem gilt für Großeltern (englisch grand parent) folgende Regel:

$$\begin{aligned} & \forall x\ \forall y\ \forall z\ (P(x,z) \wedge P(z,y)) \rightarrow GP(x,y) \\ \equiv\ & \forall x\ \forall y\ \forall z\ (\neg(P(x,z) \wedge P(z,y)) \vee GP(x,y)) \\ \equiv\ & \forall x\ \forall y\ \forall z\ (\neg P(x,z) \vee \neg P(z,y) \vee GP(x,y)) \end{aligned}$$

Weiterhin gelten folgende Fakten: $P(ben, lea)$, $P(pia, otto)$ und $P(lea, tim)$. Die Regeln und die Fakten ergeben zusammen die Programmklauseln.

$$\{P(ben, lea)\}, \{P(pia, otto)\}, \{P(lea, tim)\}, \{\neg P(x,z), \neg P(z,y), GP(x,y)\}$$

Wir fragen uns, ob $GP(ben, tim)$ aus den Programmklauseln folgt? Dazu müssen wir zeigen, dass die Regel und die Fakten und die negierte Anfrage unerfüllbar sind.

$$\begin{aligned} \{P(ben, lea)\}, \{\neg P(x,z), \neg P(z,y), GP(x,y)\} &\overset{[x/ben][z/lea]}{\rightarrow} \{\neg P(lea, y), GP(ben, y)\} \\ \{\neg P(lea, y), GP(ben, y)\}, \{P(lea, tim)\} &\overset{[y/tim]}{\rightarrow} \{GP(ben, tim)\} \\ \{GP(ben, tim)\}, \{\neg GP(ben, tim)\} &\rightarrow \emptyset \end{aligned}$$

Anmerkungen zum Schluss

Die hier vorgestellte Prädikatenlogik nennt man *Prädikatenlogik erster Stufe* oder auch *Prädikatenlogik erster Ordnung*.

- PL1 erweitert die Aussagenlogik um Quantoren, Prädikate und Terme.
- Leider lassen sich mit PL1 viele wichtige Formeln nicht ausdrücken, bspw. das Induktionsaxiom der Peano-Axiome:

$$\forall P\ (P(0) \wedge (\forall x\ P(x) \rightarrow P(x+1))) \rightarrow \forall x\ P(x)$$

- Die *Prädikatenlogik zweiter Ordnung* erlaubt Quantifizierungen auch über Funktions- und Prädikatsymbole.
- Das macht die Sache nicht leichter: Viele Eigenschaften, die in PL1 entscheidbar oder semi-entscheidbar sind, sind in PL2 unentscheidbar.

Übersicht

Übersicht

Algorithm = Logic + Control

- C, C++ oder Java: Angabe eines Lösungsalgorithmus.
  → WIE ist ein Problem zu lösen?
- Logikprogramme: Nur Angabe des Problems.
  → WAS ist zu lösen - nicht wie. (Idealfall)
- Dieser Idealfall wird bei existierenden Prolog-Systemen mit depth-first Auswertungsstrategie nicht erreicht.
- Die breadth-first Auswertungsstrategie ist zwar vollständig, aber hoffnungslos ineffizient.

→ Wir müssen einen Kompromiss finden zwischen dem Idealfall (vollständige Trennung von Logik- und Kontrollkomponente) und Effizienz.

- Programmiersprache für Logik-Programme
- kennt zusätzlich Meta-Befehle, Ein-/Ausgabe und Arithmetik
- Im Gegensatz zu reinen Logik-Programmen wird das WIE durch die Verarbeitungsreihenfolge bestimmt.
- David H.D. Warren:
  "Prolog ist [...] ein Werkzeug für das Denken."
- Basiert auf Ideen von Kowalski und Colmerauer.
- Erste Implementierung 1972 in Marseille.
- Aufschwung durch 5th Generation-Projekt.
- Neben LISP die wichtigste Sprache der KI.
- Einsatzgebiete: Expertensysteme, Rule Engines
- Warren Abstract Machine: Virtuelle Machine, die compilierten Prolog-Code ausführen kann (z.B. GNU Prolog).
- 1995 Standardisierung: ISO/IEC 13211-1, vorher viele Dialekte

Literatur

- W.F. Clocksin, C.S. Mellish:
  Programming in Prolog.
  Springer, New York, 1981
- L. Sterling, E. Shapiro:
  Prolog - Fortgeschrittene Programmiertechniken.
  Addison-Wesley, Bonn, 1988
- Ivan Bratko:
  Prolog – Programming for Artificial Intelligence.
  3rd edition, Addison-Wesley, 2000.

Aspekte bei Prolog

Es gibt syntaktische Konventionen, um verschiedene Programmbestandteile zu unterscheiden:

- Variablen: Großschreibung
- Prädikate und Funktionssymbole: Kleinschreibung
- Jede Klausel wird mit einem Punkt abgeschlossen.

In einer konkret einsetzbaren Sprache brauchen wir Konzepte, um Daten einzulesen, auszugeben oder zu bearbeiten: System stellt Prädikate wie `read` oder `write` bereit.

- Können vom Benutzer nicht verändert werden.
- Haben vom logischen Gesichtspunkt keine Bedeutung.
- Werden bei Bearbeitung durch Prologs Auswertemechanismus einfach übergangen.
- Erzeugen aber Seiteneffekte wie schreiben auf den Bildschirm oder in eine Datei.

Man muss sich über die Abfolge der Auswertung eines Prolog- Programms bewusst sein:

- Eine Zielklausel wie

  ```
  ?- read(X), compute(X,Y), write(Y).
  ```

  kann sinnvoll nur von links nach rechts ausgewertet werden.
- Im Gegensatz zu den theoretischen Betrachtungen: Nur Klauseln ohne Seiteneffekte können auch in einer anderen Reihenfolge ausgewertet werden.

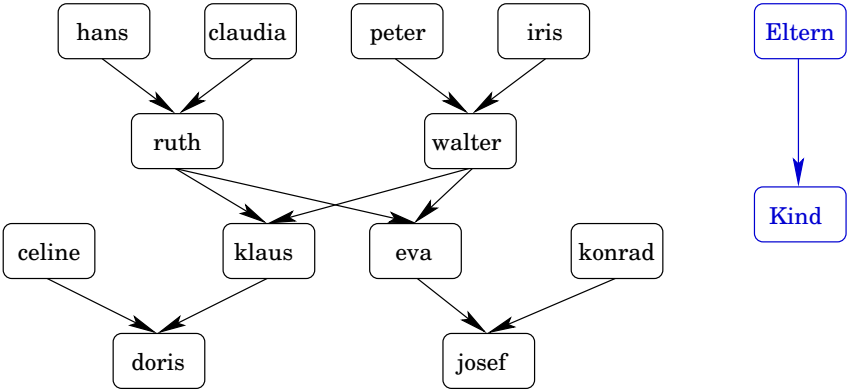
Andere Prädikate erzwingen Variableninstanziierungen:

- Trifft das Auswertesystem bspw. auf `is(X, Y*5)` (oder in Infix-Notation `X is Y*5`), wobei `Y` bereits mit der Konstanten 7 belegt ist, so wird `X` mit 35 belegt.

→ Auf diese Art können arithmetische Berechnungen in Prolog realisiert werden.

Übersicht

Wir wollen folgende Verwandschaftsbeziehung in Prolog darstellen:



Dazu definieren wir folgende Fakten:

```
parent(hans, ruth).
parent(claudia, ruth).
parent(peter, walter).
parent(iris, walter).
parent(ruth, klaus).
parent(walter, klaus).
parent(ruth, eva).
parent(walter, eva).
parent(celine, doris).
parent(klaus, doris).
parent(eva, josef).
parent(konrad, josef).
```

Dabei bezeichnet `parent(x, y)` den Sachverhalt, das `y` das Kind von `x` ist. Wir lesen `parent(x, y)` also als `x is parent of y`.

Fakten sind Prädikate ohne Variablen.

Sie finden im Internet frei verfügbare Prolog-Interpreter z.B. unter:

- `http://www.gprolog.org/`
- `https://www.swi-prolog.org/`
- `http://jlogic.sourceforge.net/`

Das Manual zu GNU-Prolog finden Sie unter

`http://www.gprolog.org/manual/gprolog.html`

Freie Prolog-Tutorials finden Sie bspw. unter:

- `http://www.learnprolognow.org/`
- `https://www.cpp.edu/~jrfisher/www/prolog_tutorial/contents.html`
- `https://ktiml.mff.cuni.cz/~bartak/prolog/`

Die Erläuterungen in diesem Kapitel beziehen sich auf den Prolog-Interpreter von GNU, gelten mit Einschränkungen aber auch für SWI-Prolog.

Nachdem wir obige Fakten als Programm in der Datei `intro.pl` gespeichert haben, können wir es am Kommando-Prompt des Interpreters einlesen:

```
| ?- [intro].
```

Falls das Programm syntaktisch korrekt ist, sollte so etwas ausgegeben werden wie:

```
compiling intro.pl for byte code...
intro.pl compiled, 12 lines read

yes
| ?-
```

Wenn Sie stattdessen das Programm mittels der Tastatur direkt eingeben wollen:

- Nach der Eingabe von `[user].` kann der Benutzer Regeln und Fakten definieren.
- Mittels `Ctrl-D` wird die Eingabe beendet und automatisch der Übersetzungsvorgang gestartet.

Um jederzeit feststellen zu können, welches Programm gerade aktiv ist, geben wir folgendes ein:

```
| ?- listing.
```

Was kann man nun mit diesem Programm machen?

- Wir können fragen, ob Ruth ein Kind von Walter ist:

```
| ?- parent(walter, ruth).
no
```

- Da Prolog annimmt, seine Fakten würden die ganze Welt komplett beschreiben, ist wahr, was gefunden wird und falsch, was nicht vorhanden ist.

Wie funktioniert das?

- Prolog vergleicht die Anfrage mit den Fakten und Regeln der Wissensbasis und prüft, ob der Funktor mit gleicher Stelligkeit vorhanden ist.
- Der Term `walter` wird mit denen der Fakten verglichen, bei Übereinstimmung wird dasselbe mit `ruth` durchgeführt.

→ Diesen Vorgang nennt man *matchen*.

Was kann man sonst noch mit diesem Programm machen?

- Wir können die Kinder von Walter auflisten:

```
| ?- parent(walter, X).
```

- Antwort:

```
X = klaus ? ;  % mit dem Semikolon wird
X = eva        % die nächste Lösung angefordert
```

- Die gleiche Anfrage mit SQL auf die Tabelle `Eltern`:

```
 Parent  Child
---------------
 ruth    klaus
 walter  klaus     Select Child from Eltern
 ruth    eva        where Parent = 'walter';
 walter  eva
 ...     ...
```

Wie funktioniert das?

- Prolog durchsucht die Fakten (von oben nach unten) und testet, ob `walter` als erster Term vorkommt (matchen).
- Anschließend wird `X` mit `klaus` *instanziiert*. (besser: belegt)
- Diesen Vorgang nennt man *Unifikation* oder *Instanziierung*.

  Ersetzen der Variablen durch Terme, so dass die Terme als Zeichenfolge gleich sind.
- Die gefundene Belegung wird ausgegeben.
- Die Lösungssuche endet noch nicht, da noch ungeprüfte Klauseln existieren.
- Die Lösungssuche wird bei der letzten Alternative fortgesetzt; alle Variablen, die ab diesem Punkt instanziiert wurden, werden wieder frei.
- Diesen Vorgang nennt man *Backtracking*.

Was kann man noch mit diesem Programm machen?

- Wir können die Eltern von Walter auflisten:

```
| ?- parent(X, walter).
```

- Bedeutung der Anfrage: Für welche `X` lässt sich das Ziel `parent(X, walter)` aus den Fakten folgern?
- Antwort:

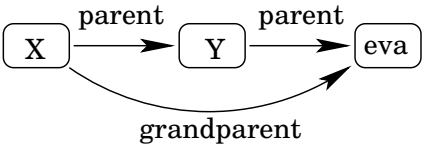
```
X = peter ? a  % mit dem 'a' werden alle
X = iris       % restlichen Lösung angefordert
no
```

Man kann „neues Wissen“ mit dem Programm generieren:

- Wir können die Großeltern von Eva ausgeben:

```
| ?- parent(Y, eva), parent(X, Y).
```

- Zur Veranschaulichung:



- Das Komma ist als logisches UND zu lesen. Ausgabe:

```
X = hans     Y = ruth ? ;
X = claudia  Y = ruth ? ;
X = peter    Y = walter ? ;
X = iris     Y = walter ? ;
no
```

Wie funktioniert das?

- Zielliste: `parent(Y, eva), parent(X, Y).`
  - Das Programm wird von oben nach unten durchsucht, und der passende Fakt 7 gefunden: `parent(ruth,eva)`
  - Also wird `Y=ruth` instanziiert, das erste Literal ist `true` und wird aus der Zielliste entfernt.
- Zielliste: `parent(X,ruth).`
  - Wieder wird das Programm von oben nach unten durchsucht und Fakt 1 gefunden: `parent(hans,ruth)`
  - Also wird `X=hans` instanziiert, das erste Literal der aktuellen Zielliste ist `true` und wird aus der Zielliste entfernt.
- Die Zielliste ist leer, damit ist die Anfrage erfüllt und die Instanziierung `X=hans, Y=ruth` wird ausgegeben.

Wird eine weitere Lösung angefordert, wird die Instanziierung von `X` wieder rückgängig gemacht und die Suche wird vor der letzten Instanziierung fortgesetzt. → Backtracking

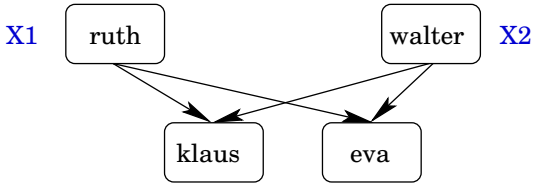
- Zielliste: `parent(X,ruth).`
  - Im Programm wird Fakt 2 gefunden: `parent(claudia,ruth)`
  - `X=claudia` wird instanziiert, das erste Literal der aktuellen Zielliste ist `true` und wird aus der Zielliste entfernt.
- Die Zielliste ist wieder leer, die nächste Lösung `X=claudia, Y=ruth` wird ausgegeben usw.

Wie können wir testen, ob zwei Personen Geschwister sind?

- Wenn Eva und Klaus Geschwister sind, haben sie einen gemeinsamen Elternteil `X`:

  ```
  | ?- parent(X, eva), parent(X, klaus).
  ```

- Zur Veranschaulichung:



- Ausgabe:

```
X = ruth ? ;   % mit dem Semikolon wird
X = walter ? ; % die nächste Lösung angefordert
no
```

Wie können wir zwischen Mutter und Vater unterscheiden?

- Zur Unterscheidung männlich/weiblich fügen wir dem Programm zwei Relationen hinzu:

```
male(hans).          female(claudia).
male(peter).         female(iris).
male(walter).        female(ruth).
male(klaus).         female(celine).
male(konrad).        female(eva).
male(josef).         female(doris).
```

- Damit können wir z.B. Walters Vater und Mutter unterscheiden:

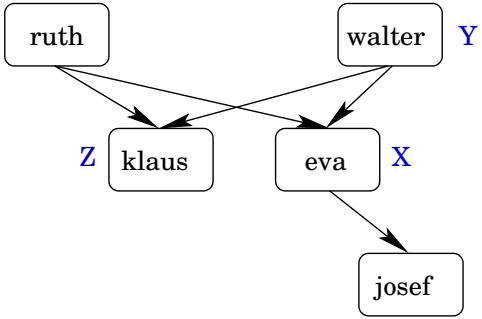
  Vater: `| ?- parent(X, walter), male(X).`
  Mutter: `| ?- parent(X, walter), female(X).`

Wie können wir Onkel und Tanten von einer Person bestimmen?

- Josefs Onkel und Tante sind Geschwister seiner Eltern:

```
| ?- parent(X,josef), parent(Y,X), parent(Y,Z), X \== Z.
```

- Zur Veranschaulichung:



- Hier muss zwingend `X` ungleich `Z` gelten! In Prolog: `X \== Z`

Wie können wir einem Programm *Regeln* hinzufügen?

- *Regeln:*
  ```
  mother(X,Y) :- parent(X,Y), female(X).
  father(X,Y) :- parent(X,Y), male(X).
  ```
- *Semantik:*
  $\forall$ `X` $\forall$ `Y` gilt: `parent(X,Y)` $\wedge$ `female(X)` $\Rightarrow$ `mother(X,Y)`
- *Sprechweise:*
  `X` is the mother of `Y`, if `X` is a parent of `Y` and `X` is female.

Im Folgenden definieren wir Regeln für

- Großeltern, -vater und -mutter.
- Geschwister, Schwester, Bruder.
- Onkel und Tante.
- eine allgemeine Vorfahr-Beziehung.

Großeltern, Großvater und Großmutter:

- *Regeln:*

```
grandparent(X,Y) :- parent(X,Z), parent(Z,Y).
grandfather(X,Y) :- grandparent(X,Y), male(X).
grandmother(X,Y) :- grandparent(X,Y), female(X).
```

- *Semantik:* $\forall$ X $\forall$ Y $\forall$ Z gilt:
  parent(X,Z) $\wedge$ parent(Z,Y) $\Rightarrow$ grandparent(X,Y)
- *Sprechweise:*
  X is grandparent of Y, if X is parent of Z and Z is parent of Y.
- Beispiel:

```
| ?- grandparent(X,josef).
X = ruth ? ;
X = walter ? ;
no
```

Wie funktioniert das?

- Zielliste: `grandparent(X,josef).`
  - Das Programm wird von oben nach unten durchsucht und die erste passende Regel gefunden:
    `grandparent(X,Y) :- parent(X,Z), parent(Z,Y).`
  - Also wird `Y=josef` instanziiert und der Eintrag der Zielliste wird ersetzt durch die rechte Seite der Regel.
- Zielliste: `parent(X,Z), parent(Z,josef).`
  - Es wird eine passende Regel für den ersten Eintrag der Zielliste gesucht und die Regel `parent(hans,ruth)` gefunden.
  - Also wird `X=hans` und `Z=ruth` instanziiert und der erste Eintrag der Zielliste durch `true` ersetzt, also entfernt.
- Zielliste: `parent(ruth,josef).`
  - Dieser Fakt wird nicht im Programm gefunden, das Ziel ist mit dieser Instanziierung nicht erfüllt.
  - Die letzte Belegung wird zurückgenommen und als Zielliste ergibt sich wieder: `parent(X,Z), parent(Z,josef).`

Wie funktioniert das? (Fortsetzung)

- Zielliste: `parent(X,Z), parent(Z,josef).`
  - Es wird der nächste Fakt `parent(claudia, ruth)` untersucht und `X=claudia` und `Z=ruth` instanziiert.
  - Der erste Eintrag der Zielliste wird durch `true` ersetzt, also entfernt.
- Zielliste: `parent(ruth,josef).`
  - Wieder wird kein übereinstimmender Fakt gefunden und ein Backtracking durchgeführt.
- Zielliste: `parent(X,Z), parent(Z,josef).`
  - Dieser Vorgang wiederholt sich so lange, bis der Fakt `parent(ruth,eva)` gefunden wird, also `X=ruth` und `Z=eva`.
- Zielliste: `parent(eva,josef)`
  - Dieses Ziel wird als Fakt gefunden und die Instanziierung `X=ruth` ausgegeben. Der Wert von `Z` wird nicht ausgegeben, da er nicht in der ursprünglichen Zielliste auftaucht.

Geschwister haben einen gemeinsamen Elternteil:

- *Regeln:*

```
sibling(X,Y) :- parent(Z,X), parent(Z,Y), X \== Y.
sister(X,Y) :- sibling(X,Y), female(X).
brother(X,Y) :- sibling(X,Y), male(X).
```

- *Semantik:* $\forall$ `X` $\forall$ `Y` $\forall$ `Z` gilt:
  `parent(Z,X)` $\wedge$ `parent(Z,Y)` $\wedge$ `X` $\neq$ `Y` $\Rightarrow$ `sibling(X,Y)`
- *Sprechweise:*
  `X` is sibling of `Y`, if `Z` is parent of `X` and `Z` is parent of `Y` and `X` is not equal to `Y`.

- Onkel und Tante

```
aunt(X,Y) :- sibling(X,Z), parent(Z,Y), female(X).
uncle(X,Y) :- sibling(X,Z), parent(Z,Y), male(X).
```

- eine allgemeine Vorfahr-Beziehung *mittels Rekursion*

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
```

Übersicht

Die bekannten arithmetischen Operationen sind definiert:

`+` Addition	`/` Division (floating point)
`-` Subtraktion	`//` Division (integer)
`*` Multiplikation	`mod` Rest bei ganzzahliger Division

Allerdings wird die Anfrage

```
| ?- X = 1 + 2.
```

nicht mit `X = 3`, sondern mit `X = 1 + 2` beantwortet!

Der Ausdruck `1 + 2` ist in Prolog ein Term, wobei + die Funktion und `1` und `2` die Operanden sind. Die Addition wird nur dann durchgeführt, wenn der Operator `is` verwendet wird:

```
| ?- X is 1 + 2.
```

Werden numerische Werte verglichen, erfolgt auch automatisch eine Auswertung der arithmetischen Ausdrücke:

```
| ?- 7 * 3 > 20.          | ?- 7 * 3 < 20.
yes                       no
```

Dies gilt für die folgenden Operatoren:

`X > Y`	`X` is greater than `Y`
`X < Y`	`X` is less than `Y`
`X >= Y`	`X` is greater than or equal to `Y`
`X =< Y`	`X` is less than or equal to `Y`
`X =:= Y`	the values of `X` and `Y` are equal
`X =\= Y`	the values of `X` and `Y` are not equal

Beachten Sie den Unterschied zwischen = und =:= sowie den Unterschied zwischen \== und =\=.

Wichtig: Machen Sie sich den Unterschied zwischen den beiden Operatoren = und =:= klar!

- Bei `X = Y` versucht Prolog, die Objekte `X` und `Y` zu unifizieren (gleich zu machen) und ggf. mit Werten zu belegen.

```
| ?- 1 + 2 = 2 + 1.
no
```

```
| ?- 1 + X = Y + 3.
X = 3
Y = 1
yes
```

- Der Operator =:= wertet den arithmetischen Ausdruck aus, belegt die Variablen aber nicht mit Werten.

```
| ?- 1 + 2 =:= 2 + 1.
yes
```

```
| ?- 1 + X =:= Y + 3.
uncaught exception: ....
```

Im zweiten Beispiel hätten `X` und `Y` vorher mit Werten belegt werden müssen, z.B. `X=4, Y=2, 1+X =:= Y+3.`

Test auf Gleichheit:

`=`	unifizierbar
`==`	identisch mit
`=:=`	numerisch gleich

Test auf Ungleichheit:

`\=`	nicht unifizierbar
`\==`	nicht identisch mit
`=\=`	numerisch ungleich

Beispiel:

```
| ?-  X=1+3, Y=2+2, X=:=Y.
yes
```

Beim Vergleich `X=:=Y` werden die Terme `1+3` und `2+2` zunächst ausgewertet und dann verglichen.

```
| ?- X=1+3, Y=2+2, X==Y.
no
```

Bei dem Vergleich `X==Y` wird geprüft, ob die Inhalte von `X` und `Y` identisch sind (ohne eine arithmetische Auswertung).

Beispiel:

```
| ?-  1+X = 1+Y.
Y = X
yes
```

Die Variablen `X` und `Y` können unifiziert werden. Hier sieht man, das Prolog einen allgemeinsten Unifikator sucht. Es werden nicht alle Möglichkeiten aufgezählt wie `X=1,Y=1` und `X=2,Y=2` usw., stattdessen wird nur die „minimale“ Unifikation erzeugt.

```
| ?-  1+X == 1+Y.
no
```

Prolog versucht nicht, die Terme gleich zu machen, es findet hier keine Unifikation statt, es werden lediglich zwei Zeichenketten miteinander verglichen.

Übung 61. Schreiben Sie ein Prädikat `maximum(X,Y,Z,Erg)`, das den maximalen Wert der drei Variablen `X`, `Y` und `Z` bestimmt und in `Erg` bereit stellt.

Übung 62. Schreiben Sie ein Prädikat `fib(N,Erg)`, das die N-te Fibonacci-Zahl F_N in `Erg` berechnet.

Übung 63. Schreiben Sie ein Prädikat `ggT(X,Y,Erg)` zur Berechnung des größten gemeinsamen Teilers zweier ganzer Zahlen `X` und `Y`.

Zur Erinnerung:

$$ggT(x,y) = \begin{cases} x, & \text{falls } y = 0 \\ ggT(y, x \bmod y), & \text{sonst} \end{cases}$$

In Prolog können wir zur Laufzeit Fakten und Regeln der Datenbasis hinzufügen.

- `asserta(term)` fügt `term` am Anfang der Datenbasis ein.
- `assertz(term)` fügt `term` am Ende der Datenbasis an.
- `retract(term)` entfernt `term` wieder aus der Datenbasis.

Damit das funktioniert, muss der Term entweder unbekannt sein, oder das Prädikat muss als `dynamic` vereinbart sein.

Um die Laufzeit des `fib`-Prädikats zur Berechnung der Fibonacci-Zahlen zu verbessern, vermeiden wir rekursive Aufrufe durch memorieren bereits berechneter Werte.

```
:- dynamic(fib/2).

fib(0,0).
fib(1,1).
fib(N, Erg) :-
    N > 0,
    N1 is N-1,
    N2 is N-2,
    fib(N1,E1),
    fib(N2,E2),
    Erg is E1+E2,
    asserta(fib(N, Erg)).
```

Vergleichen Sie die Laufzeiten dieser und der zuvor erstellten Lösung. Wie groß darf die Eingabe `N` werden?

Ändert sich die Laufzeit, wenn `assertz` anstelle von `asserta` verwendet wird? Lassen Sie sich das Programm mittels `listing` anzeigen.

Übersicht

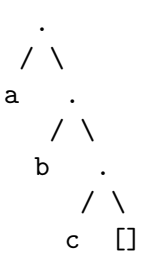
Listen

Eine Liste ist eine Aufzählung von Elementen und wird in Prolog in eckigen Klammern `[ ... ]` geschrieben:

```
[adam, eva, heidi, peter, bonnie, clyde]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Wir können eine Liste `[a,b,c]` auch in Paar-Schreibweise angeben:
`.(a,.(b,.(c,[])))`

Listen werden in Prolog als entartete Binärbäume gespeichert:



Daraus ergibt sich die dritte Darstellung: Aufzählung mit Restliste

- Eine leere Liste wird geschrieben als `[]`.
- Eine nicht-leere Liste besteht immer
  - aus dem ersten Element, Kopf (engl.: head) der Liste genannt,
  - und dem Rest, Schwanz (engl.: tail) der Liste genannt.

Der senkrechte Strich teilt den Kopf vom Schwanz der Liste:

```
L = [a, b, c, d]
L = [a | Tail]
```

Unifizieren wir `[X|Rest]` mit `[1,2,3,4]`, dann erhalten wir:

```
X = 1
Rest = [2,3,4]
```

Listen können auch ineinander verschachtelt werden:

```
| ?- Hobbies1 = [rennen, tanzen],
     Hobbies2 = [schlafen, essen],
     L = [heidi, Hobbies1, peter, Hobbies2].

Hobbies1 = [rennen,tanzen]
Hobbies2 = [schlafen,essen]
L = [heidi,[rennen,tanzen],peter,[schlafen,essen]]
```

Wir betrachten im Weiteren folgende Operationen:

- `member(X, L)` testet, ob das Objekt `X` in der Liste `L` enthalten ist.
- `sublist(L1, L)` testet, ob `L1` eine Teilliste von `L` ist.
- `permutation(L, P)` erzeugt in `P` alle Permutationen der in `L` enthaltenen Objekte.

`member(X, L)` testet, ob das Objekt `X` in der Liste `L` enthalten ist:

```
| ?- member(b, [a,b,c]).
true

| ?- member(b, [a,[b,c],d]).
no

| ?- member([b,c], [a,[b,c],d]).
true
```

`X` ist Element der Liste `L`, wenn

- `X` der Kopf der Liste ist, also
  `member(X, [X|Tail]).`
- oder `X` im Rest der Liste enthalten ist, also
  `member(X, [Head|Tail]) :- member(X, Tail).`

Übung 64. Schreiben Sie eine Funktion `conc(L1, L2, L)`, die die Listen `L1` und `L2` aneinander hängt. In der Ergebnisliste `L` stehen also zunächst alle Elemente von `L1`, dann alle Elemente von `L2`.

Übung 65. Schreiben Sie eine Funktion `add(X,L1,L)`, die ein Element `X` am Anfang der Liste `L1` einfügt, Liste `L` ist die Ergebnisliste.

Übung 66. Schreiben Sie eine Funktion `del(X,L1,L)`, die ein Element `X` aus der Liste `L1` entfernt und Liste `L` liefert.

Was liefert der Aufruf `del(1, L, [2,3,4]).`?

Übung 67. Schreiben Sie eine Funktion `insert(X,L1,L)`, die ein Element `X` an einer beliebigen Stelle der Liste `L1` einfügt und Liste `L` liefert.

Debugging: Mittels `trace` aktivieren Sie den Debugger, mittels `notrace` wird der Debugger ausgeschaltet.

```
| ?- trace.
| ?- insert(1, [4,3,2], L).
      1    1  Call: insert(1,[4,3,2],_22) ?
      1    1  Exit: insert(1,[4,3,2],[1,4,3,2]) ?

L = [1,4,3,2] ? ;
      1    1  Redo: insert(1,[4,3,2],[1,4,3,2]) ?
      2    2  Call: insert(1,[3,2],_55) ?
      2    2  Exit: insert(1,[3,2],[1,3,2]) ?
      1    1  Exit: insert(1,[4,3,2],[4,1,3,2]) ?
| ?- notrace.
```

`sublist(L1, L)` testet, ob `L1` eine Teilliste von `L` ist:

```
| ?- sublist([c,d,e], [a,b,c,d,e,f]).
true

| ?- sublist([c,e], [a,b,c,d,e,f]).
no
```

`S` ist eine Teilliste von `L`, wenn

- `L` in zwei Listen `L1` und `L2` aufgeteilt werden kann und
- die Liste `L2` wiederum in die Liste `S` und eine Liste `L3` aufgeteilt werden kann.

  `sublist(S,L) :- conc(L1,L2,L), conc(S,L3,L2).`
- Stellen Sie diesen Sachverhalt graphisch dar.
- Was liefert der Aufruf `sublist(S, [1,2,3]).`?

Übung 68. Die in gprolog eingebaute Funktion `sublist` verhält sich etwas anders:

```
| ?- sublist([c,e], [a,b,c,d,e,f]).
true

| ?- sublist([e,c], [a,b,c,d,e,f]).
no
```

Wie könnte diese Funktion implementiert sein?

`permutation(L, P)` erzeugt in `P` alle Permutationen der in `L` enthaltenen Objekte

```
| ?- permutation([a,b,c], P).
P = [a,b,c];
P = [a,c,b];
P = [b,a,c];
....
```

Mögliche Implementierung:

- Abbruchbedingung: `permutation([], []).`
- Falls die erste Liste die Form `[X|L]` hat: Erstelle zunächst Liste `L1` als Permutation von `L`, füge dann `X` an jeder Stelle in `L1` ein.

```
permutation([X|L], P) :- permutation(L, L1),
                         insert(X,L1,P).
```

Übung 69. Geben Sie eine mögliche Implementierung der in gprolog eingebauten Relation `reverse` an.

Beispiel: `reverse([a,b,c,d], P).` liefert `P = [d,c,b,a]`

Übung 70. Definieren Sie eine Relation `shift(L1, L2)`, so dass `L2` aus `L1` hervorgeht, indem die Elemente zyklisch nach links verschoben werden.

Beispiel:

```
| ?- shift([a,b,c,d,e], L1),
     shift(L1, L2).
```

liefert

```
L1 = [b,c,d,e,a]
L2 = [c,d,e,a,b]
```

Übung 71. Schreiben Sie ein Prädikat `lastElem(L,X)`, das das letzte Listenelement der Liste `L` in der Variablen `X` liefert.

Übung 72. Schreiben Sie Prädikate, mit denen man feststellen kann, ob eine Liste Präfix (bzw. Suffix) einer anderen Liste ist.

Anonyme Variablen: Häufig benötigt man einen Wert nicht weiter. In einem solchen Fall kann man eine anonyme Variable verwenden, die mit einem Unterstrich beginnt. Ihr Wert wird nicht gespeichert:

```
erstesElement(X, [X|_Rest]).
```

Übersicht

Der Cut

Damit Prolog für praktische Probleme zugänglicher wird, gibt es den Cut-Operator '!', mit dem das Backtracking durch Teilbäume des Regelwerks unterbunden werden kann.

Die Lösungsmenge ist dann zwar nicht mehr in jedem Fall identisch, aber Programme können unter Performance-Aspekten optimiert werden.

Steht in einer Klausel ein Cut und wird diese Stelle bei der Auswertung erreicht, so werden alle Belegungen, die beim Matching der vorangehenden Prädikate getroffen wurden eingefroren. Für sie findet kein Backtracking mehr statt, es werden auch keine Klauseln der gleichen Prozedur mehr ausprobiert.

Scheitert der Regelteil hinter dem Cut, so scheitert die gesamte Prozedur.

Der Cut: grüne und rote Cuts

Ein Cut kann eingesetzt werden, um die Berechnung zu beschleunigen, ohne dass Lösungen verlorengehen oder hinzukommen. Solche Cuts heißen *grüne Cuts*.

```
minimum(X,Y,X) :- X < Y, !.
minimum(X,Y,Y) :- X >= Y, !.
```

Der zweite Cut kann weggelassen werden, da die Prozedur `minimum` dann abgeschlossen ist. (Eine Zusammenfassung von „Regeln gleichen Namens“ nennt man auch Prozedur.)

Ohne Cuts hat das Programm die gleiche Lösungsmenge.

Der Cut: grüne und rote Cuts

Alternativ:

```
minimum(X,Y,Z) :- X < Y, !, Z=X.
minimum(X,Y,Y).
```

Hier berücksichtigt man die SLD-Strategie: Auswerten der Klauseln von oben nach unten und innerhalb einer Klausel von links nach rechts.

- Aufruf `minimum(2,3,3)` liefert bei Verwendung des Cuts `no`.
  - Zunächst wird `X < Y` als true bewertet, der Cut wird als true interpretiert, die Unifikation `Z=X` schlägt fehl.
  - Deshalb geht der Prolog-Interpreter die Literale der Klausel zurück, um ein Backtracking zu ermöglichen.
  - Dabei trifft er auf den Cut, das Backtracking wird - auf dieser Ebene - abgebrochen, damit ist insgesamt die Auswertung fehlgeschlagen.
- Der Aufruf `minimum(2,3,3)` liefert ohne Verwendung des Cuts `yes`.
- Dies ist ein *roter Cut*.

Übersicht

Bubblesort

Idee:

- Tausche zwei benachbarte Zahlen X und Y, falls $X > Y$.
- Sortiere den Rest.

```
% Wenn ein Wertepaar getauscht wurde, also swap erfolgreich
% war, darf die nächste Regel NICHT ausgeführt werden!
% Roter Cut!
bubblesort(List, SortedList) :-
    swap(List, List2), !,
    bubblesort(List2, SortedList).

bubblesort(Sorted, Sorted).

swap([X,Y|Rest], [Y,X|Rest]) :- X > Y.
swap([Z|Rest], [Z|Rest2]) :- swap(Rest, Rest2).
```

Insertionsort

Idee: Sortiere nicht-leere Liste `L=[X|Tail]` wie folgt:

- Sortiere den Rest `Tail` der Liste `L` zu `SortedTail`.
- Füge `X` an der richtigen Stelle in `SortedTail` ein.

```
insertionsort([], []).
insertionsort([X|Tail], SortedList) :-
    insertionsort(Tail, SortedTail),
    insert(X, SortedTail, SortedList).

% Wenn diese Regel angewendet wird, darf die
% nächste Regel NICHT angewendet werden. Roter Cut!
insert(X, [Y|Sorted], [Y|Sorted2]) :-
    X > Y, !,
    insert(X, Sorted, Sorted2).
insert(X, Sorted, [X|Sorted]).
```

Quicksort

Idee: Sortiere eine Liste `L` wie folgt:

- Wähle das erste Element `X` der Liste `L` als Pivot-Element.
- Teile mittels `split` die Restliste auf in eine Liste mit kleineren Werten `Small` und eine Liste mit größeren Werten `Big` als `X`.
- Sortiere `Small` rekursiv zu `SortedSmall`.
- Sortiere `Big` rekursiv zu `SortedBig`.
- Füge die Listen mittels `conc` zusammen zur sortierten Liste `SortedList`.

```
quicksort([], []).
quicksort([X|Tail], SortedList) :-
    split(X, Tail, Small, Big),
    quicksort(Small, SortedSmall),
    quicksort(Big, SortedBig),
    conc(SortedSmall, [X|SortedBig], SortedList).
```

Quicksort

```
%%% helper functions for quicksort
split(X, [], [], []).

split(X, [Y|Tail], [Y|SmallItems], BigItems) :-
    X > Y, !,       % Roter Cut!
    split(X, Tail, SmallItems, BigItems).

split(X, [Y|Tail], SmallItems, [Y|BigItems]) :-
    split(X, Tail, SmallItems, BigItems).

conc([], L, L).
conc([X|L1], L2, [X|L3]) :- conc(L1, L2, L3).
```

Übersicht

In Prolog gibt es build-in predicates zur Interaktion mit dem Anwender:

- `write(X)` gibt den Inhalt des Terms `X` auf dem Bildschirm aus.
  - `nl` steht für new line.
  - `tab(N)` fügt `N` Leerzeichen in die Ausgabe ein.
- `read(X)` liest den nächsten Term von der Tastatur und belegt `X` mit dem gelesenen Wert.
  Die Eingabe muss mit einem Punkt abgeschlossen werden.

Beispiel:

```
cube :-
    write('next value, please: '),
    read(N),
    C is N*N*N,
    write(C).
```

Übung 73. Schreiben Sie ein Prädikat `writelist(L)`, das die einzelnen Listen einer Liste von Listen `L` jeweils in einer Zeile ausgibt.

Ausgabe von `writelist([[1,2,3], [3,4,5,6]])`:

```
1,2,3
3,4,5,6
```

Übung 74. Schreiben Sie ein Prädikat `bars(L)`, das zu jeder Zahl der Liste jeweils eine Zeile mit entsprechend vielen Sternchen ausgibt.

Ausgabe von `bars([3,6,5])`:

```
***
******
*****
```

Dateioperationen:

- `tell(File)` leitet die Ausgabe der Prädikate `write`, `put`, `nl` usw. in die Datei `File` um.
- `told` beendet die Umleitung und bewirkt, dass spätere Ausgaben wieder auf dem Bildschirm erscheinen.
- `telling(File)` gibt an, in welche Datei die Ausgabe zur Zeit erfolgt, dabei steht `user` für die Tastatur.

```
| ?- telling(Anfang),
     tell('datei.txt'),      % öffnen der Datei
     telling(Mitte),
     write('Hallo, '), nl,   % erste Zeile schreiben
     write('Welt!'), nl,     % zweite Zeile schreiben
     told,                   % schließen der Datei
     telling(Ende).
```

Dateioperationen:

- `see(File)` nimmt die Eingabe für die Prädikate `read`, `get` usw. aus der Datei `File`.
- `seen` beendet die Umleitung und bewirkt, dass spätere Eingaben wieder vom Benutzer abgefragt werden.
- `seeing(File)` gibt an, aus welcher Datei die Eingabe zur Zeit genommen wird, dabei steht `user` für die Tastatur.

```
| ?- seeing(Anfang),
     see('datei.txt'),     % öffnen der Datei
     seeing(Mitte),
     get0(X), put(X),      % erstes Zeichen lesen
     get0(Y), put(Y),      % zweites Zeichen lesen
     seen,                 % schließen der Datei
     seeing(Ende).
```

Das Prädikat `read` kann nur dann zum Lesen verwendet werden, wenn Terme gelesen werden, also bspw. Zahlen, Fakten oder Regeln, die allesamt mit einem Punkt abgeschlossen sind.

```
test :-
    tell('datei.txt'),      % öffnen der Datei
    write('sunshine.'), nl,
    write('raining.'), nl,
    write('funny :- sunshine, raining.'), nl,
    write('nice :- sunshine, \\+raining.'),
    told.                   % schließen der Datei
```

In GNU-Prolog wird `not` als `\+` dargestellt und muss innerhalb einer Zeichenkette natürlich maskiert werden.

Die in einer Datei abgelegten Fakten und Regeln können wie folgt eingelesen und der Datenbasis hinzugefügt werden:

```
add_to_database(File) :-
    see(File),          /* open file */
    repeat,
    read(Data),         /* read from File */
    process(Data),
    seen,               /* close File */
    !.                  /* stop reading */

process(end_of_file) :- !.
process(Data) :-  asserta(Data), nl, fail.
```

Das Prädikat `repeat` ist bei jeder Auswertung wahr und kann verwendet werden, um Schleifen zu programmieren.

Wir wollen zunächst ein Rätsel der folgenden Form lösen:

```
                                  MEHR
  SEND       AAL       BILL     + MATHE      GERALD
+ MORE     + AAL     + IRMA     + MACHT    + DONALD
-------    ------    -------    --------   ---------
 MONEY      FANG      LIEBE      FREUDE      ROBERT
```

Eine naive Implementierung ohne Listen könnte zunächst die einzelnen Ziffern definieren:

```
digit(1).        digit(6).
digit(2).        digit(7).
digit(3).        digit(8).
digit(4).        digit(9).
digit(5).        digit(0).
```

Die einzelnen Ziffern müssen paarweise verschieden sein, die ersten Ziffern müssen ungleich 0 sein.

```
solve :-                    % SEND + MORE = MONEY
  digit(S), S\=0,
  digit(E), E\=S,
  digit(N), N\=E, N\=S,
  digit(D), D\=N, D\=E, D\=S,
  digit(M), M\=D, M\=N, M\=E, M\=S, M\=0,
  digit(O), O\=M, O\=D, O\=N, O\=E, O\=S,
  digit(R), R\=O, R\=M, R\=D, R\=N, R\=E, R\=S,
  digit(Y), Y\=R, Y\=O, Y\=M, Y\=D, Y\=N, Y\=E, Y\=S,
  sum(S,E,N,D,M,O,R,Y),
  write('S = '), write(S), nl,
  ....
  write('Y = '), write(Y), nl.
```

Abschließend müssen wir noch prüfen, ob die gegebene Bedingung SEND + MORE = MONEY ist:

```
sum(S,E,N,D,M,O,R,Y) :-
        N1 is            1000*S + 100*E + 10*N + D,
        N2 is            1000*M + 100*O + 10*R + E,
        N3 is 10000*M + 1000*O + 100*N + 10*E + Y,
        N3 =:= N1 + N2.
```

Fragen:

- Wie können wir die Laufzeit des Programms verbessern?
- Können wir Listen zur Lösung des Problems nutzen, um das Programm kürzer zu gestalten?

Naives Generate-And-Test-Verfahren:

- Erzeuge eine Permutation für die Belegung der Variablen,

```
gen(S,E,N,D,M,O,R,Y) :-
    permutation([S,E,N,D,M,O,R,Y,_,_],
                [0,1,2,3,4,5,6,7,8,9]).
```

- teste, ob die gegebene Gleichung für die Belegung erfüllt ist

```
gl(S,E,N,D,M,O,R,Y) :- M \= 0, S \= 0,
                (S*1000 + E*100 + N*10 + D) +
                (M*1000 + O*100 + R*10 + E) =:=
     (M*10000 + O*1000 + N*100 + E*10 + Y).
```

- und gib die Belegung aus.

```
solve :- gen(S,E,N,D,M,O,R,Y),
         gl(S,E,N,D,M,O,R,Y),
         write('S = '), write(S), nl,
         write('E = '), write(E), nl,   ....
```

Wir können es natürlich auch einmal mit dem Einsatz von Wissen versuchen! Hier können wir Wissen über die Addition einbringen:

```
    S  E  N  D          10 * C1 + Y =:= D + E
  + M  O  R  E          10 * C2 + E =:= N + R + C1
  + C3 C2 C1            10 * C3 + N =:= E + O + C2
 ---------------        10 *  M + O =:= S + M + C3
  M  O  N  E  Y         M = 1   (Übertrag)
```

Wir formen die obigen Gleichungen um und prüfen nach jeder Belegung einer Variablen, ob die Teilbelegung noch richtig ist:

```
solve :-
   M = 1,                       % M muss 1 sein
   L = [2,3,4,5,6,7,8,9,0],     % restliche Ziffern
   select(S, L, L1),            % wähle Ziffer für S
   S > 0,                       % S muss ungleich 0 sein
```

Zahlenrätsel

```
(C3 = 0; C3 = 1),          % Übertrag ist 0 oder 1
O is S + M + C3 - 10*M,    % O ist jetzt eindeutig
select(O, L1, L2),         % entferne O aus Rest

select(E, L2, L3),         % wähle Ziffer für E
(C2 = 0; C2 = 1),          % Übertrag ist 0 oder 1
N is E + O + C2 - 10*C3,   % N ist jetzt eindeutig
select(N, L3, L4),         % entferne N aus Rest

(C1 = 0; C1 = 1),          % Übertrag ist 0 oder 1
R is E + 10*C2 - N - C1,   % R ist jetzt eindeutig
select(R, L4, L5),         % entferne R aus Rest
```

```
    select(D, L5, L6),          % wähle Ziffer für D
    Y is D + E - 10*C1,         % Y ist jetzt eindeutig
    select(Y, L6, _),           % entferne Y aus Rest

    % und dann noch die Ausgabe
    write('S = '), write(S), nl,
    write('E = '), write(E), nl,
    ....
    write('Y = '), write(Y), nl.
```

Laufzeitvergleich?

Wir wollen nun ein Programm schreiben, das obige Idee aufgreift, aber Zahlenrätsel mittels Listen löst.

Wenn wir zwei Zahlen addieren, gehen wir spaltenweise von rechts nach links vor. In einer Spalte addieren wir einen Übertrag `C1` von rechts und zwei Ziffern `D1` und `D2`. Wir erhalten als Ergebnis eine Ziffer `D` und einen Übertrag `C` nach links.

```
  . . . D1 . . .
  . . . D2 . . .
        C1     0
----------------
0     C
  . . .  D . . .
```

Das Carry-Bit in der rechten Spalte ist 0, ebenso darf links vom Ergebnis kein Übertrag-Bit auftreten. Außerdem gilt:

- `D = (D1 + D2 + C1) mod 10`
- `C = (D1 + D2 + C1) div 10`

Wir definieren ein Prädikat `sum(N1, N2, N)`, das die zwei Zahlen `N1` und `N2`, repräsentiert als Liste, addiert und das Ergebnis als Liste `N` speichert: `sum([0,A,A,L], [0,A,A,L], [F,A,N,G]).`

Das verwendete Prädikat `sum1(N1,N2,N,C1,C2,Digs1,Digs2)` wertet zusätzliche Informationen aus:

- `C1`: Übertrag-Bit vor der Summation
- `C2`: Übertrag-Bit nach der Summation
- `Digs1`: Menge der verfügbaren Ziffern vor der Summation
- `Digs2`: Menge der verbleibenden Ziffern nach der Summation

```
sum(N1, N2, N) :-
    sum1(N1, N2, N, 0, 0, [0,1,2,3,4,5,6,7,8,9], _).

sum1([], [], [], 0, 0, Digits, Digits).
sum1([D1|N1], [D2|N2], [D|N], C1, C, Digs1, Digs) :-
    sum1(N1, N2, N, C1, C2, Digs1, Digs2),
    digitsum(D1, D2, C2, D, C, Digs2, Digs).
```

Das Prädikat `digitsum(D1,D2,C1,D,C,Digs1,Digs)`

- wählt aus den noch verfügbaren Ziffern in `Digs1` die Ziffern `D1`, `D2` und `D` aus, so dass nach der Summation der Spalte noch die Ziffern in `Digs` zur Verfügung stehen.
- berechnet den Wert `D = (D1+D2+C1) mod 10` sowie den Wert `C = (D1+D2+C1) div 10`.

```
digitsum(D1, D2, C1, D, C, Digs1, Digs) :-
    del(D1, Digs1, Digs2),
    del(D2, Digs2, Digs3),
    del(D, Digs3, Digs),
    S is D1 + D2 + C1,
    D is S mod 10,
    C is S // 10.
```

Wir müssen darauf achten, dass eine bereits instanziierte Variable `D1`, `D2` oder `D` nicht nochmals aus der Liste `Digs1` entfernt wird.

```
% do nothing if A is already instantiated
del(A, L, L) :- nonvar(A), !.
del(A, [A|L], L).
del(A, [B|L], [B|L1]) :- del(A, L, L1).

puzzle1([O,S,E,N,D], [O,M,O,R,E], [M,O,N,E,Y]).
puzzle2([O,B,I,L,L], [O,I,R,M,A], [L,I,E,B,E]).
puzzle3([D,O,N,A,L,D], [G,E,R,A,L,D], [R,O,B,E,R,T]).
puzzle4([O,A,A,L], [O,A,A,L], [F,A,N,G]).
```

Aufrufbeispiel: `puzzle1(N1,N2,N3), sum(N1,N2,N3).`

Es gibt natürlich auch kompliziertere Zahlenrätsel:

```
  ABB -  CD =  EED          ACE + DAC = JFD
   -     -      *            -     +     -
   FD +  EF =   CE          AAA - HFC =  GI
   =     =      =            =     =     =
  EGD *  FH = ????          AH + III = JBJ
```

Ein naives Generate-And-Test-Verfahren testet einfach wieder alle möglichen Belegungen:

```
gen(A,B,C,D,E,F,G,H) :-
    permutation([A,B,C,D,E,F,G,H,_,_],
                [0,1,2,3,4,5,6,7,8,9]).
```

Anschließend testen wir, ob alle Gleichungen erfüllt sind:

```
gl1(A,B,C,D,E) :- (E*100 + E*10 + D) =:=
    ((A*100 + B*10 + B) - (C*10 + D)).
gl2(C,D,E,F) :- (C*10 + E) =:=
    ((F*10 + D) + (E*10 + F)).
gl3(A,B,D,E,F,G) :- (E*100 + G*10 + D) =:=
    ((A*100 + B*10 + B) - (F*10 + D)).
gl4(C,D,E,F,H) :- (F*10 + H) =:=
    ((C*10 + D) - (E*10 + F)).

gl5(C,D,E,F,G,H,X) :-
    ((E*100 + E*10 + D) * (C*10 + E)) =:=
        ((E*100 + G*10 + D) * (F*10 + H)),
    X is ((E*100 + G*10 + D) * (F*10 + H)).
```

Und nun alles zusammen packen:

```
solve :- gen(A,B,C,D,E,F,G,H),
         gl1(A,B,C,D,E),
         gl2(C,D,E,F),
         gl3(A,B,D,E,F,G),
         gl4(C,D,E,F,H),
         gl5(C,D,E,F,G,H,X),
         write('A = '), write(A), nl,
         write('B = '), write(B), nl,
         ....
         write('H = '), write(H), nl,
         write('X = '), write(X).
```

Übersicht

Erinnern wir uns:

- 3 Missionare und 3 Kannibalen wollen einen Fluss überqueren.
- Es steht nur ein Ruderboot für die Überfahrt zur Verfügung.
- Das Boot kann maximal 2 Personen aufnehmen.
- Es dürfen nie mehr Kannibalen als Missionare an einem Ort sein, sonst gibt es ein großes Fressen.

Zunächst definieren wir einen gültigen Zustand:

```
% keine Missionare --> beliebig viele Kannibalen
zust( 0, Kl, _,  _, Kr) :- Kl >= 0, Kr >= 0.
zust( _, Kl, _,  0, Kr) :- Kl >= 0, Kr >= 0.

% sonst: Anzahl Missionare >= Anzahl Kannibalen
zust(Ml, Kl, _, Mr, Kr) :- Ml >= Kl, Mr >= Kr,
                           Kl >= 0, Kr >= 0.
```

Dann definieren wir die Überfahrten vom linken zum rechten Ufer:

Kannibalen und Missionare

```
% 1 Kannibale von Links nach Rechts
zug(n(Ml,Kl,'L',Mr,Kr), n(Ml,Kl2,'R',Mr,Kr2)) :-
    Kl2 is Kl - 1,
    Kr2 is Kr + 1,
    zust(Ml,Kl2,'R',Mr,Kr2).

% 2 Kannibalen von Links nach Rechts
zug(n(Ml,Kl,'L',Mr,Kr), n(Ml,Kl2,'R',Mr,Kr2)) :-
    Kl2 is Kl - 2,
    Kr2 is Kr + 2,
    zust(Ml,Kl2,'R',Mr,Kr2).

% 1 Kannibale, 1 Missionar von Links nach Rechts
zug(n(Ml,Kl,'L',Mr,Kr), n(Ml2,Kl2,'R',Mr2,Kr2)) :-
    Ml2 is Ml - 1,
    Kl2 is Kl - 1,
    Mr2 is Mr + 1,
    Kr2 is Kr + 1,
    zust(Ml2,Kl2,'R',Mr2,Kr2).
```

```
% 1 Missionar von Links nach Rechts
zug(n(Ml,Kl,'L',Mr,Kr), n(Ml2,Kl,'R',Mr2,Kr)) :-
    Ml2 is Ml - 1,
    Mr2 is Mr + 1,
    zust(Ml2,Kl,'R',Mr2,Kr).

% 2 Missionare von Links nach Rechts
zug(n(Ml,Kl,'L',Mr,Kr), n(Ml2,Kl,'R',Mr2,Kr)) :-
    Ml2 is Ml - 2,
    Mr2 is Mr + 2,
    zust(Ml2,Kl,'R',Mr2,Kr).
```

Dann definieren wir die Überfahrten vom rechten zum linken Ufer:

Kannibalen und Missionare

```
% 1 Kannibale von Rechts nach Links
zug(n(Ml,Kl,'R',Mr,Kr), n(Ml,Kl2,'L',Mr,Kr2)) :-
    Kl2 is Kl + 1,
    Kr2 is Kr - 1,
    zust(Ml,Kl2,'L',Mr,Kr2).

% 2 Kannibalen von Rechts nach Links
zug(n(Ml,Kl,'R',Mr,Kr), n(Ml,Kl2,'L',Mr,Kr2)) :-
    Kl2 is Kl + 2,
    Kr2 is Kr - 2,
    zust(Ml,Kl2,'L',Mr,Kr2).

% 1 Kannibale, 1 Missionar von Rechts nach Links
zug(n(Ml,Kl,'R',Mr,Kr), n(Ml2,Kl2,'L',Mr2,Kr2)) :-
    Ml2 is Ml + 1,
    Kl2 is Kl + 1,
    Mr2 is Mr - 1,
    Kr2 is Kr - 1,
    zust(Ml2,Kl2,'L',Mr2,Kr2).
```

Kannibalen und Missionare

```
% 1 Missionar von Rechts nach Links
zug(n(Ml,Kl,'R',Mr,Kr), n(Ml2,Kl,'L',Mr2,Kr)) :-
    Ml2 is Ml + 1,
    Mr2 is Mr - 1,
    zust(Ml2,Kl,'L',Mr2,Kr).

% 2 Missionare von Rechts nach Links
zug(n(Ml,Kl,'R',Mr,Kr), n(Ml2,Kl,'L',Mr2,Kr)) :-
    Ml2 is Ml + 2,
    Mr2 is Mr - 2,
    zust(Ml2,Kl,'L',Mr2,Kr).
```

Nun definieren wir eine Zug-Folge, einen Pfad. Im einfachsten Fall erkennen wir keine doppelten Zustände, sondern beschränken nur die Suchtiefe, um „Endlosschleifen“ zu verhindern.

```
% Abbruch, wenn die Suchtiefe überschritten wurde:
pfad(n(_,_,_,_,_), n(_,_,_,_,_), Tiefe, _) :-
    Tiefe > 10, !, fail.
```

Um später einen Pfad ausgeben zu können, merken wir uns die gemachten Züge in einer Liste.

```
% Ein einzelner Zug ist ein zulässiger Pfad:
pfad(n(M1,K1,P1,M2,K2), n(M3,K3,P3,M4,K4),
        Tiefe, [n(M3,K3,P3,M4,K4)|[]]) :-
    zug(n(M1,K1,P1,M2,K2), n(M3,K3,P3,M4,K4)).
```

Ein Pfad von A nach B existiert, falls es einen Zug von A nach C gibt und ein Pfad von C nach B existiert. Dabei müssen wir die Tiefe hochzählen:

```
pfad(n(M1,K1,P1,M2,K2), n(M5,K5,P5,M6,K6),
        Tiefe, [n(M3,K3,P3,M4,K4)|L]) :-
    T is Tiefe + 1,
    zug(n(M1,K1,P1,M2,K2), n(M3,K3,P3,M4,K4)),
    pfad(n(M3,K3,P3,M4,K4), n(M5,K5,P5,M6,K6), T, L).
```

Jetzt müssen wir das Ganze nur noch mit einem einfachen Aufruf versehen:

```
solve :- pfad(n(3,3,'L',0,0), n(0,0,'R',3,3), 0, L),
         write([n(3,3,'L',0,0)|L]).
```

Können wir auf den Tiefenzähler verzichten?

- Wir werden die bereits untersuchten Zustände, also die bisherige Zugfolge, in einer Liste speichern.
- Vor der Ausführung eines neuen Zugs wird getestet, ob der Knoten in der Liste enthalten ist, also bereits untersucht wurde.
- Auf diese Weise vermeiden wir doppelte Zustände und verhindern „Endlosschleifen“.

Die Liste der bisherigen Züge wird in die Ergebnisliste kopiert, wenn eine Belegung erfolgreich war. Wir führen also zwei Listen:

- Die erste Liste enthält die Zustände, die bereits untersucht wurden. Wir füllen diese Liste beim rekursiven Abstieg.
- Da diese Informationen beim Rekursionsabbau wieder verloren gehen, kopieren wir die Werte am Rekursionsende in eine zweite Liste.

Am Rekursionsende wird die erste Liste in die zweite kopiert:

```
% Abbruch, wenn beide Zustände gleich sind:
pfad(n(M1,K1,P,M2,K2), n(M1,K1,P,M2,K2), L, L) :- !.
```

Sonst erzeugen wir einen Zug zu einem Zwischenziel,

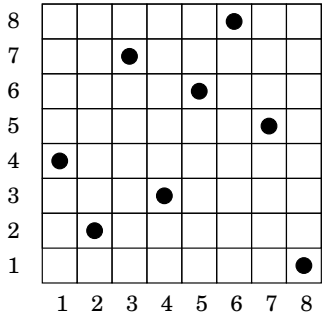
- testen, ob der neue Zustand (das Zwischenziel) noch nicht in der ersten Liste gespeichert ist,
- hängen den neuen Zustand an die alte Liste an und
- versuchen, einen Pfad vom Zwischenziel zum Ziel zu finden.

```
pfad(n(M1,K1,P1,M2,K2), n(M5,K5,P5,M6,K6), L1, L2) :-
   zug(n(M1,K1,P1,M2,K2), n(M3,K3,P3,M4,K4)),
   \+ member(n(M3,K3,P3,M4,K4), L1),
   append(L1, [n(M3,K3,P3,M4,K4)], L),
   pfad(n(M3,K3,P3,M4,K4), n(M5,K5,P5,M6,K6), L, L2).
```

Übersicht

Eine Lösung des Problems soll mittels des unären Prädikats `solution(Pos)` dargestellt werden.

Wir stellen den Zustand als Liste von x/y-Koordinaten dar: `[X1/Y1, X2/Y2, X3/Y3, ..., X8/Y8]`.



Damit keine zwei Damen in der selben Spalte stehen, definieren wir ein Template `template([1/Y1, 2/Y2, 3/Y3, ..., 8/Y8])`.

Sammeln wir einige Eigenschaften von gültigen Lösungen:

- Eine leere Liste von Damen ist eine gültige Lösung, da sich keine Damen schlagen.

→ `solution([]).`

- Für eine nicht-leere Liste der Form `[X/Y, Others]` muss gelten:
  - Die Liste `Others` muss eine gültige Lösung sein.
  - `X` und `Y` müssen Integer-Werte zwischen 1 und 8 sein.
  - Die Dame auf Position `X/Y` darf sich mit keiner Dame der Liste `Others` schlagen.

→
```
solution([X/Y | Others]) :-
    solution(Others),
    member(Y, [1,2,3,4,5,6,7,8]),
    noattack(X/Y, Others).
```

Wie testen wir, ob sich die Dame auf Position `X/Y` mit denen der Liste `Others` schlägt, also ob `noattack(Q, QRest)` erfüllt ist?

- Wenn die Liste `QRest` leer ist, schlagen sich keine Damen.

→
```
noattack(_, []).
```

- Falls `QRest` die Form `[Q1 | QRest1]` hat,
  - darf die Dame `Q` die Dame `Q1` nicht schlagen, und
  - die Dame `Q` darf die Damen in `QRest1` nicht schlagen.

→
```
noattack(X/Y, [X1/Y1 | Others]) :-
    Y =\= Y1,            % Zeilen ungleich
    Y1-Y =\= X1-X,       % Diagonalen ungleich
    Y1-Y =\= X-X1,
    noattack(X/Y, Others).
```

Als Abfrage erhalten wir also: `template(S), solution(S)`.

Übersicht

Zunächst nur eine naive Implementierung:

- Es werden keine Kreise erkannt.
- Vermeide endlose Rekursion durch Tiefenbeschränkung.
- Damit eine Lösung gefunden wird, müssen wir die Tiefe sukzessive erhöhen (iterative deepening).
- Die 0 beschreibt die Position der Lücke.

```
% Die beiden Endzustände:
puzzle(1,2,3,8,0,4,7,6,5,[],Tiefe).
puzzle(0,1,2,3,4,5,6,7,8,[],Tiefe).

% Abbruchbedingung für Tiefensuche:
puzzle(X1,X2,X3,Y1,Y2,Y3,Z1,Z2,Z3,L,Tiefe):-
    Tiefe > 7, !, fail.
```

```
% Zeile 1, Position links
% Lücke nach rechts schieben
puzzle(0,X2,X3,Y1,Y2,Y3,Z1,Z2,Z3,[X2|L],Tiefe) :-
    T is Tiefe+1,
    puzzle(X2,0,X3,Y1,Y2,Y3,Z1,Z2,Z3,L,T).

% Lücke nach unten schieben
puzzle(0,X2,X3,Y1,Y2,Y3,Z1,Z2,Z3,[Y1|L],Tiefe) :-
    T is Tiefe+1,
    puzzle(Y1,X2,X3,0,Y2,Y3,Z1,Z2,Z3,L,T).
```

```
% Zeile 1, Position mitte
% Lücke nach links schieben
puzzle(X1,0,X3,Y1,Y2,Y3,Z1,Z2,Z3,[X1|L],Tiefe) :-
    T is Tiefe+1,
    puzzle(0,X1,X3,Y1,Y2,Y3,Z1,Z2,Z3,L,T).

% Lücke nach rechts schieben
puzzle(X1,0,X3,Y1,Y2,Y3,Z1,Z2,Z3,[X3|L],Tiefe) :-
    T is Tiefe+1,
    puzzle(X1,X3,0,Y1,Y2,Y3,Z1,Z2,Z3,L,T).

% Lücke nach unten schieben
puzzle(X1,0,X3,Y1,Y2,Y3,Z1,Z2,Z3,[Y2|L],Tiefe) :-
    T is Tiefe+1,
    puzzle(X1,Y2,X3,Y1,0,Y3,Z1,Z2,Z3,L,T).
```

```
% Zeile 1, Position rechts
% Lücke nach links schieben
puzzle(X1,X2,0,Y1,Y2,Y3,Z1,Z2,Z3,[X2|L],Tiefe) :-
    T is Tiefe+1,
    puzzle(X1,0,X2,Y1,Y2,Y3,Z1,Z2,Z3,L,T).

% Lücke nach unten schieben
puzzle(X1,X2,0,Y1,Y2,Y3,Z1,Z2,Z3,[Y3|L],Tiefe) :-
    T is Tiefe+1,
    puzzle(X1,X2,Y3,Y1,Y2,0,Z1,Z2,Z3,L,T).
```

```
% Zeile 2, Position links
% Lücke nach rechts schieben
puzzle(X1,X2,X3,0,Y2,Y3,Z1,Z2,Z3,[Y2|L],Tiefe) :-
    T is Tiefe+1,
    puzzle(X1,X2,X3,Y2,0,Y3,Z1,Z2,Z3,L,T).

% Lücke nach oben schieben
puzzle(X1,X2,X3,0,Y2,Y3,Z1,Z2,Z3,[X1|L],Tiefe) :-
    T is Tiefe+1,
    puzzle(0,X2,X3,X1,Y2,Y3,Z1,Z2,Z3,L,T).

% Lücke nach unten schieben
puzzle(X1,X2,X3,0,Y2,Y3,Z1,Z2,Z3,[Z1|L],Tiefe) :-
    T is Tiefe+1,
    puzzle(X1,X2,X3,Z1,Y2,Y3,0,Z2,Z3,L,T).
```

```
% Zeile 2, Position mitte
% Lücke nach links schieben
puzzle(X1,X2,X3,Y1,0,Y3,Z1,Z2,Z3,[Y1|L],Tiefe) :-
    T is Tiefe+1,
    puzzle(X1,X2,X3,0,Y1,Y3,Z1,Z2,Z3,L,T).

% Lücke nach rechts schieben
puzzle(X1,X2,X3,Y1,0,Y3,Z1,Z2,Z3,[Y3|L],Tiefe) :-
    T is Tiefe+1,
    puzzle(X1,X2,X3,Y1,Y3,0,Z1,Z2,Z3,L,T).
```

```
% Zeile 2, Position mitte (Fortsetzung)
% Lücke nach oben schieben
puzzle(X1,X2,X3,Y1,0,Y3,Z1,Z2,Z3,[X2|L],Tiefe) :-
    T is Tiefe+1,
    puzzle(X1,0,X3,Y1,X2,Y3,Z1,Z2,Z3,L,T).

% Lücke nach unten schieben
puzzle(X1,X2,X3,Y1,0,Y3,Z1,Z2,Z3,[Z2|L],Tiefe) :-
    T is Tiefe+1,
    puzzle(X1,X2,X3,Y1,Z2,Y3,Z1,0,Z3,L,T).
```

```
% Zeile 2, Position rechts
% Lücke nach links schieben
puzzle(X1,X2,X3,Y1,Y2,0,Z1,Z2,Z3,[Y2|L],Tiefe) :-
    T is Tiefe+1,
    puzzle(X1,X2,X3,Y1,0,Y2,Z1,Z2,Z3,L,T).

% Lücke nach oben schieben
puzzle(X1,X2,X3,Y1,Y2,0,Z1,Z2,Z3,[X3|L],Tiefe) :-
    T is Tiefe+1,
    puzzle(X1,X2,0,Y1,Y2,X3,Z1,Z2,Z3,L,T).

% Lücke nach unten schieben
puzzle(X1,X2,X3,Y1,Y2,0,Z1,Z2,Z3,[Z3|L],Tiefe) :-
    T is Tiefe+1,
    puzzle(X1,X2,X3,Y1,Y2,Z3,Z1,Z2,0,L,T).
```

```
% Zeile 3, Position links
% Lücke nach rechts schieben
puzzle(X1,X2,X3,Y1,Y2,Y3,0,Z2,Z3,[Z2|L],Tiefe) :-
    T is Tiefe+1,
    puzzle(X1,X2,X3,Y1,Y2,Y3,Z2,0,Z3,L,T).

% Lücke nach oben schieben
puzzle(X1,X2,X3,Y1,Y2,Y3,0,Z2,Z3,[Y1|L],Tiefe) :-
    T is Tiefe+1,
    puzzle(X1,X2,X3,0,Y2,Y3,Y1,Z2,Z3,L,T).
```

```
% Zeile 3, Position mitte
% Lücke nach links schieben
puzzle(X1,X2,X3,Y1,Y2,Y3,Z1,0,Z3,[Z1|L],Tiefe) :-
    T is Tiefe+1,
    puzzle(X1,X2,X3,Y1,Y2,Y3,0,Z1,Z3,L,T).

% Lücke nach rechts schieben
puzzle(X1,X2,X3,Y1,Y2,Y3,Z1,0,Z3,[Z3|L],Tiefe) :-
    T is Tiefe+1,
    puzzle(X1,X2,X3,Y1,Y2,Y3,Z1,Z3,0,L,T).

% Lücke nach oben schieben
puzzle(X1,X2,X3,Y1,Y2,Y3,Z1,0,Z3,[Y2|L],Tiefe) :-
    T is Tiefe+1,
    puzzle(X1,X2,X3,Y1,0,Y3,Z1,Y2,Z3,L,T).
```

```
% Zeile 3, Position rechts
% Lücke nach links schieben
puzzle(X1,X2,X3,Y1,Y2,Y3,Z1,Z2,0,[Z2|L],Tiefe) :-
    T is Tiefe+1,
    puzzle(X1,X2,X3,Y1,Y2,Y3,Z1,0,Z2,L,T).

% Lücke nach oben schieben
puzzle(X1,X2,X3,Y1,Y2,Y3,Z1,Z2,0,[Y3|L],Tiefe) :-
    T is Tiefe+1,
    puzzle(X1,X2,X3,Y1,Y2,0,Z1,Z2,Y3,L,T).
```

Auch hier können wir, wie beim Kannibalen-Missionare-Problem, die Tiefenbeschränkung ersetzen, indem wir den neuen Zustand daraufhin überprüfen, ob er in der Liste der bereits besuchten Zustände enthalten ist.

Allerdings nützt uns das in diesem Fall gar nichts, da bei der Tiefensuche evtl. sehr ungünstige, d.h. sehr lange Wege gefunden werden und die Suchtiefe die Stack-Größe von Prolog überschreitet:

```
Fatal Error: global stack overflow
```

Daher werden wir den A^*-Algorithmus implementieren, um kürzere, bessere Lösungen zu finden. Das hier vorgestellte Programm ist angelehnt an die Lösung unter: `https://www.cpp.edu/~jrfisher/www/prolog_tutorial/contents.html`

Übersicht

Einen Knoten des Baums stellen wir als Struktur `node(State, Depth, F, [Ancestors])` dar:

- State: repräsentierter Zustand
- Depth: Tiefe des Knotens im Baum, also $g(n)$.
- F: Wert der Bewertungsfunktion $f(n) = g(n) + h(n)$.
- Ancestors: Liste der Vorgänger des Knotens.

Um eine Lösung zu berechnen, rufen wir das Haupt-Prädikat des Programms `solve(Start, Sol)` auf. Dabei beschreibt `Start` den Startzustand und in `Sol` wird die Lösung dargestellt:

- Zunächst wird der f-Wert für `Start` berechnet.
- Anschließend wird mittels `search` ein Weg zum Ziel gesucht. Dabei wird die aktualisierte OpenList als Parameter übergeben.
- Schließlich muss die Vorgängerliste umgedreht werden.

```
solve(State,Solution) :-
    f_function(State,0,F),
    search([node(State,0,F,[])],S),
    reverse(S,Solution).
```

Das Prädikat `f_function(State,D,F)` berechnet den Wert `F` für den Zustand `State` in Tiefe `D`.

- Dazu wird ein Prädikat `h_function(State,H)` aufgerufen, das wir für jedes Problem spezifisch implementieren müssen.
- Das Prädikat liefert in `H` den Wert der Schätzfunktion h vom Zustand `State` aus.

```
f_function(State,D,F) :-
    h_function(State,H),
    F is D + H.
```

Die Suche eines Zielknotens mittels `search` benötigt die OpenList:

- Der erste Knoten `B` wird aus der OpenList entfernt, alle Folgeknoten `Children` von `B` werden mittels `expand` bestimmt und mittels `insert_all` in die OpenList eingefügt. Anschließend wird die Suche rekursiv mit der aktualisierten OpenList aufgerufen.
- Die Suche endet, falls ein Ziel gefunden wird. Dazu muss für jedes Problem ein spezifisches Prädikat `goal(State)` bereit gestellt werden, das anzeigt, ob `State` ein gültiges Ziel ist.

```
search([node(State,_,_,Sol)|_], Sol) :- goal(State).
search([B|Tail],S) :-
    expand(B,Children),
    insert_all(Children,Tail,Open),
    search(Open,S).
```

Wenn ein Knoten `node(State, Depth, F, [A])` expandiert wird, passiert folgendes:

- Bestimme alle Kinder des Knotens.
- Jedes Kind hat Tiefe `Depth + 1`.
- Der `F`-Wert wird für jedes Kind berechnet.
- Die Vorgänger-Liste `A` wird vom Kind übernommen und ergänzt.

```
expand(node(State,D,_,S),All_My_Children) :-
    bagof(node(Child,D1,F,[Move|S]),
          ( D1 is D+1,
            move(State,Child,Move),
            f_function(Child,D1,F)),
          All_My_Children).
```

Das Einfügen von Knoten in die OpenList erfolgt mittels einer Insertionsort-Variante. Dabei werden doppelte Zustände entfernt, die Einordnung der Elemente erfolgt anhand des f-Wertes.

```
insert_all([F|R],Open1,Open3) :-
    insert(F,Open1,Open2),
    insert_all(R,Open2,Open3).
insert_all([],Open,Open).

insert(B,Open,Open) :- repeat_node(B,Open), ! .
insert(B,[C|R],[B,C|R]) :- cheaper(B,C), ! .
insert(B,[B1|R],[B1|S]) :- insert(B,R,S), !.
insert(B,[],[B]).

repeat_node(node(P,_,_,_), [node(P,_,_,_)|_]).
cheaper(node(_,_,F1,_), node(_,_,F2,_)) :- F1 < F2.
```

8-Puzzle: Zunächst die Aufzählung aller möglichen Züge.

```
left( A/0/C/D/E/F/H/I/J , 0/A/C/D/E/F/H/I/J ).
left( A/B/C/D/0/F/H/I/J , A/B/C/0/D/F/H/I/J ).
left( A/B/C/D/E/F/H/0/J , A/B/C/D/E/F/0/H/J ).
left( A/B/0/D/E/F/H/I/J , A/0/B/D/E/F/H/I/J ).
left( A/B/C/D/E/0/H/I/J , A/B/C/D/0/E/H/I/J ).
left( A/B/C/D/E/F/H/I/0 , A/B/C/D/E/F/H/0/I ).

up( A/B/C/0/E/F/H/I/J , 0/B/C/A/E/F/H/I/J ).
up( A/B/C/D/0/F/H/I/J , A/0/C/D/B/F/H/I/J ).
up( A/B/C/D/E/0/H/I/J , A/B/0/D/E/C/H/I/J ).
up( A/B/C/D/E/F/0/I/J , A/B/C/0/E/F/D/I/J ).
up( A/B/C/D/E/F/H/0/J , A/B/C/D/0/F/H/E/J ).
up( A/B/C/D/E/F/H/I/0 , A/B/C/D/E/0/H/I/F ).
```

```
right( A/0/C/D/E/F/H/I/J , A/C/0/D/E/F/H/I/J ).
right( A/B/C/D/0/F/H/I/J , A/B/C/D/F/0/H/I/J ).
right( A/B/C/D/E/F/H/0/J , A/B/C/D/E/F/H/J/0 ).
right( 0/B/C/D/E/F/H/I/J , B/0/C/D/E/F/H/I/J ).
right( A/B/C/0/E/F/H/I/J , A/B/C/E/0/F/H/I/J ).
right( A/B/C/D/E/F/0/I/J , A/B/C/D/E/F/I/0/J ).

down( A/B/C/0/E/F/H/I/J , A/B/C/H/E/F/0/I/J ).
down( A/B/C/D/0/F/H/I/J , A/B/C/D/I/F/H/0/J ).
down( A/B/C/D/E/0/H/I/J , A/B/C/D/E/J/H/I/0 ).
down( 0/B/C/D/E/F/H/I/J , D/B/C/0/E/F/H/I/J ).
down( A/0/C/D/E/F/H/I/J , A/E/C/D/0/F/H/I/J ).
down( A/B/0/D/E/F/H/I/J , A/B/F/D/E/0/H/I/J ).
```

Nun müssen wir einige, für das 8-Puzzle spezifische Prädikate definieren. Als Schätzer verwenden wir die Manhatten-Distanz:

```
h_function(Puzz,H) :- p_fcn(Puzz,H).

p_fcn(A/B/C/D/E/F/G/H/I, P) :-
    a(A,Pa), b(B,Pb), c(C,Pc),
    d(D,Pd), e(E,Pe), f(F,Pf),
    g(G,Pg), h(H,Ph), i(I,Pi),
    P is Pa+Pb+Pc+Pd+Pe+Pf+Pg+Ph+Pg+Pi.

a(0,0). a(1,0). a(2,1). a(3,2). a(4,3).
        a(5,4). a(6,3). a(7,2). a(8,1).
b(0,0). b(1,1). b(2,0). b(3,1). b(4,2).
        b(5,3). b(6,2). b(7,3). b(8,2).
```

```
c(0,0). c(1,2). c(2,1). c(3,0). c(4,1).
        c(5,2). c(6,3). c(7,4). c(8,3).
d(0,0). d(1,1). d(2,2). d(3,3). d(4,2).
        d(5,3). d(6,2). d(7,2). d(8,0).
e(0,0). e(1,2). e(2,1). e(3,2). e(4,1).
        e(5,2). e(6,1). e(7,2). e(8,1).
f(0,0). f(1,3). f(2,2). f(3,1). f(4,0).
        f(5,1). f(6,2). f(7,3). f(8,2).
g(0,0). g(1,2). g(2,3). g(3,4). g(4,3).
        g(5,2). g(6,2). g(7,0). g(8,1).
h(0,0). h(1,3). h(2,3). h(3,3). h(4,2).
        h(5,1). h(6,0). h(7,1). h(8,2).
i(0,0). i(1,4). i(2,3). i(3,2). i(4,1).
        i(5,0). i(6,1). i(7,2). i(8,3).
```

$A^\star$-Algorithmus

Das Prädikat `expand` des $A^\star$-Algorithmus benötigt das Prädikat `move`, das wir für das 8-Puzzle wie folgt definieren:

```
move(State, Child, l) :-  left(State, Child).
move(State, Child, u) :-  up(State, Child).
move(State, Child, r) :-  right(State, Child).
move(State, Child, d) :-  down(State, Child).
```

In der Ergebnisliste, die dem Benutzer des Prolog-Programms angezeigt wird, stehen nur die Werte 'l', 'u', 'r' und 'd'.

Als Letztes müssen wir noch das Ziel-Prädikat definieren:

```
goal(1/2/3/8/0/4/7/6/5).
```

Jetzt können wir eine Lösung zu einem Startzustand berechnen lassen:

```
| ?- solve(1/6/2/7/0/3/5/8/4, S).
S = [d,l,u,r,u,r,d,d,l,u] ? ;
S = [d,l,u,r,u,r,d,d,l,u,l,r] ? ;
S = [d,l,u,d,u,r,u,r,d,d,l,u] ? ;
S = [d,l,u,r,u,r,d,d,l,u,l,r,l,r] ? ;
S = [d,l,r,l,u,r,u,r,d,d,l,u] ?
.....
```

Wir nutzen oben das Systemprädikat `bagof(X,P,L)`:

- Liefert die Liste `L`, die alle Objekte `X` enthält, die das Ziel `P` erfüllen.
- Macht in der Regel nur Sinn, falls `X` und `P` gemeinsame Variablen enthalten.
- Beispiel: Klassifizierung von Buchstaben in Vokale und Konsonanten.

```
class(a, vok).      class(d, kon).
class(b, kon).      class(e, vok).
class(c, kon).      class(f, kon).
```

Wir können mittels `bagof` eine Liste aller Vokale erstellen lassen:

```
| ?- bagof(Letter, class(Letter, vok), Letters).
Letters = [a,e]
yes
```

Weitere Systemprädikate:

- `integer(A)` wird einmal wahr, falls A eine ganze Zahl ist.
- `var(A)` wird einmal wahr, wenn A eine (nicht instantiierte) Variable ist.
- `repeat` wird beliebig oft wahr.
- und viele weitere