Vorbereitung für die Vertiefungsprüfung Künstliche Intelligenz, Wissensrepräsentation und Datenbanken

Andreas Wortmann
(andreas.wortmann@rwth-aachen.de)

Basierend auf den Vorbereitungen von

Dominique Ziegelmayer
(dominique.ziegelmayer@rwth-aachen.de)

Inhaltsverzeichnis

1 Einleitung

1.1 Vorwort von mir

Liebe Kommilitioninnen und Kommilitonen, mit diesem Dokument habe ich - basierend auf Dominiques Vorbereitungen - die Liste der häufigsten Prüfungsfragen für die Vertiefungsprüfung in den Gebieten Künstliche Intelligenz, Knowledge Representation und Datenbanken zusammengefasst. Hierzu habe ich unter anderem Protokolle ab 1995 von Markus de Brün, Stefan Schubert, J.W., Joachim Börger, Tobias Vössing, Daniel Beyer, Lutz Ißler, Andreas Strack, Topper Powderfan, Patrick Eggert, Dominik Klein, Thorben Keller und 15 anonymen Kommilitonen zusammengefasst und um ein Punkte aus verschiedenen Zusammenfassungen erweitert. Die Fragen zur Datenbanken-Vorlesung stammen fast vollständig aus „Zusammenfassung Kemper/Eickler Datenbanksysteme 5.Auflage“ von Christian Pöcher, während der Großteil der Materialien aus der Fachschaft I/1[1] oder von S-Inf[2] stammt. Ich übernehme weiterhin keinerlei Garantie für Richtigkeit oder Vollständigkeit der Unterlagen und weise wie Dominique darauf hin, dass dieses Dokument keinesfalls als Vorlesungsersatz diesen soll. Die LaTeX-Dateien und Grafiken zu diesem PDF werden, solange mein RWTH-Webspace existiert auf diesem[3] zur Verfügung stehen und soll frei - im Sinne der GPL2[4] weiterverwendet werden. Ansonsten gelten alle weiteren Hinweise von Dominique.

Zur Prüfung kann ich mich meinen Vorgängern und Vorgängerinnen nur anschließen: Lakemeyer war ein sehr angenehmer Prüfer der mir Zeit zum nachdenken ließ und mir auch mal auf die Sprünge half, wenn es hakte, so daß es dann auch zur 1.3 reichte.

Viel Erfolg,
Andreas Wortmann.

1.2 Vorwort von Dominique

Liebe Kommilitonen, in diesem Dokument habe ich die häufigsten Prüfungsfragen für die Diplomprüfung in den Gebieten KI, KR und Logikprogrammierung zusammengefasst und mich bemüht zu fast allen Fragen ein paar Worte zu schreiben. Ich möchte hier ausdrücklich darauf hinweisen, dass ich keinerlei Garantie auf Vollständigkeit und

[1] http://www.fsmpi.rwth-aachen.de/studium/pruefungsprotokolle/
[2] http://s-inf.de
[3] https://www-users.rwth-aachen.de/andreas.wortmann/vertiefungspruefung.tex.zip
[4] http://www.gnu.org/licenses/gpl-2.0.html

Richtigkeit gebe. Dieses Dokument soll eine Hilfe für all diejenigen sein, die kurz vor Ihrer Prüfung stehen und selbst ihr Wissen überprüfen möchten. Nicht empfehlenswert ist ausschliesslich aus diesem Dokument zu lernen und keinerlei Hintergründe zu kennen. Verbesserungen und Anmerkungen könnt ihr mir gerne schicken, allerdings muss ich natürlich sehen, wieviel Zeit ich habe, diese zu berücksichtigen. Ansonsten allen viel Erfolg, Dominique Ziegelmayer.

2 Künstliche Intelligenz

2.1 Agenten

2.1.1 Woraus besteht ein rationaler Agent?

- **Sensoren**: Nehmen die Umwelt wahr.
- **Effektoren**: Beeinflussen die Umwelt.
- **Controller**: Übernimmt Steuerung.

2.1.2 Welche Agententypen gibt es?

Es gibt Reflexive Agenten, Table-Lookup Agenten, Agenten mit internem Weltmodell, Agenten mit expliziten Zielen und nützlichkeitsbasierte Agenten

2.1.3 Woraus besteht der perfekte rationale Agent?

Der perfekte rationale Agent ist eine Abbildung: Percept Sequences × World Knowledge → Actions. Die Realisierung eines solchen Agenten ist aufgrund des großen Definitionsbereichs für interessante Probleme unmöglich.

2.1.4 Was unterscheidet die Agenten mit expliziten Zielen von den nützlichkeitsbasierten Agenten?

Während der Agent mit expliziten Zielen nur jeweils die Aktion wählt, die am nächsten zu Ziel führt, betrachtet der nützlichkeitsbasierte Agent auch die Frage „Wie komme ich am effizientesten ans Ziel?“.

2.1.5 Wie unterscheiden sich ein lernender Agent von einem nicht-lernenden Agenten?

Der lernende Agent verfügt drei zusätzliche Komponenten:

1. **Problemgenerator**: Schlägt Aktionen vor, die zu neuen „Erfahrungen“ führen.
2. **Kritiker**: Wertet die Güte der Aktionen aus.
3. **Lernelement**: Verbessert mit den Informationen des Kritiker das Performance-Element.

2.1.6 Nennen Sie mogliche Eigenschaften der Umgebungen und erklären Sie diese!

- **Accessible** vs. **non-accessible**: Werden alle relevanten Aspekte der Umwelt über die Sensoren erfasst?
- **Deterministic** vs. **non-deterministic**: Führen Aktionen immer zu dem erwarteten Ergebnis?
- **Static** vs. **Dynamic**: Kann sich die Welt verändern, während der Agent nachdenkt?
- **Episodic** vs. **non-episodic**: Ist die nächste Aktion des Agenten ist unabhängig von vorigen Aktionen?
- **Discrete** vs. **Continous**: Ist die Welt mit einer endlichen Zustandsmenge beschreibbar?

2.2 Suche

2.2.1 Welche Kritierien legt man an Suchverfahren an?

- **Vollständigkeit**: Wird eine Lösung gefunden, wenn es eine gibt?
- **Optimalität**: Wird eine „beste“ Lösung gefunden?
- **Zeitkomplexität**: Wie lange dauert die Suche?
- **Platzkomplexität**: Wieviel Speicherplatz benötigt die Suche?

2.2.2 Welche Problemklassen gibt es? Erklären sie diese!

- **Single-State** Problems: Vollständiges Weltwissen, vollständiges Wissen über den Effekt der Aktionen. (Beweis-Assistent)
- **Multiple State** Problems: Unvollständiges Weltwissen, vollständiges Wissen über den Effekt der Aktionen (Reflexiver Staubsauger).
- **Contingency** Problems: Vollständiges Weltwissen, unvollständiges Wissen über den Effekt der Aktionen (Spiele).
- **Exploration** Problems: Unvollständiges Weltwissen, unvollständiges Wissen über den Effekt der Aktionen.

2.2.3 Was ist ein Zustand beim Suchen?

Ein Zustand ist - abstrakt gesehen - ein Knoten im Suchbaum.

2.2.4 Was benötigt man alles zur Suche?

- **Zustände**,
- **Aktionen** mit **Kosten**,
- **Goal-Test** (Test ob der betrachtete Zustand ein Zielzustand ist) und eine
- **Expansionsstrategie**

2.2.5 Was ist ein Ziel?

Ein Zustand aus der Menge der Zielzustände.

2.2.6 Was ist Suche?

Suche ist ein Verfahren, welches ausgehend von einem Intitialzustand eine *Sequenz von Aktionen* sucht, die den *Initialzustand* in einen *Zielzustand* transformiert. Dafür expandiert Suche abhängig von der verwendeten Strategie Knoten (d.h. sie erzeugt alle seine Nachfolger) ausgehend vom Wurzelknoten.

2.2.7 In welchen Fällen interessiert man sich nicht für den Suchpfad?

Wenn das Gesuchte bereits im Zielknoten vorhanden ist, so wie beispielsweise beim POP-Planning oder dem 8-Damen-Problem. Hier ist der Weg egal, da die Lösung - also der Plan bzw. die Aufstellung der Damen - vollständig im Zielknoten steckt.

2.2.8 Welche uninformierten Suchverfahren gibt es?

Breitensuche, Uniform Cost Search, Tiefensuche, Depth-Limited Search, Iterative Deepening und Bidirectional Search.

2.2.9 Wann sind Breitensuche und Tiefensuche vollständig, wann optimal?

Breitensuche ist vollständig, wenn der Verzweigungsfaktor endlich ist und optimal wenn die Lösung *tiefenoptimal* ist, d.h. wenn die flachste Lösung auch die beste ist. Tiefensuche ist vollständig, wenn der Baum keine unendlichen Pfade hat und nicht optimal.

2.2.10 Wie aufwendig sind Breitensuche und Tiefensuche?

Breitensuche und Tiefensuche expandieren im Worst Case den vollständigen Baum, d.h. $O\left(b^d\right)$ Knoten. Während die Breitensuche auch Platz für diese $O\left(b^d\right)$ Knoten benötigt, kommt die Tiefensuche mit Platz für $O\left(b \cdot d\right)$ Knoten aus.

2.2.11 Was ist die „Uniform Cost Search“ und wann ist sie optimal, wann vollständig?

Uniform Cost Search erweitert die Breitensuche um die Berücksichtigung von Kantengewichten und expandiert den Knoten mit den *geringsten Pfadkosten*. Das Verfahren ist vollständig und optimal, wenn die Schrittkosten $> \epsilon$, d.h. die Pfadkosten mit der Suchtiefe monoton ansteigen. Laufzeit und Speicherverbrauch verhalten sich wie bei der Breitensuche.

2.2.12 Wie aufwendig ist „Uniform Cost Search“?

Laufzeit und Speicherverbrauch verhalten sich wie bei der Breitensuche.

2.2.13 Welche uninformierte Suche würden Sie verwenden?

Bei einer unbekannten Suchtiefe würde ich Iterative Deepening verwenden. Das Verfahren *simuliert eine Breitensuche mit wiederholten Tiefensuchen* und ist damit ebenso *vollständig* und *tiefenoptimal*, benötigt jedoch nur Zeit und Platz wie die Tiefensuche, d.h. terminiert in Zeit $O\left(b^d\right)$ und in Platz $O\left(b \cdot d\right)$.

2.2.14 Wann ist Iterative Deepening vollständig, wann optimal?

Iterative Deepening verhält sich *wie Uniform Cost Search*, d.h. es ist für endliche Verzweigungsfaktoren immer vollständig und optimal, wenn die Pfadkosten mit der Suchtiefe ansteigen (beispielweise bei konstanten Schrittkosten).

2.2.15 Was ist eine Heuristik bei der Suche und was macht man damit?

Eine Heuristik bei der Suche ist eine Funktion, welche die Entfernung vom aktuellen Knoten zum Zielknoten *approximiert*. Diese Funktion wird von den Informierten Suchen verwendet um die Suche schrittweise „anzuleiten“ und sie dadurch effizienter zu machen.

2.2.16 Wie findet man eine Heuristik?

Durch *Vereinfachung des Problems*, beispielsweise durch Heranziehen der Luftlinienentfernung zur Pfadbestimmung.

2.2.17 Welche Anforderungen werden an eine Heuristik gestellt?

Eine Heuristik sollte *leicht zu berechnen* sein und die Entfernung zum Ziel möglichst *exakt abschätzen*.

2.2.18 Wann ist eine Heuristik „zulässig“?

Wenn die prognostizierten Kosten $h\left(n\right)$ die tatsächlichen Kosten $h^*\left(n\right)$ immer *unterschätzen*.

2.2.19 Welche informierten Suchverfahren kennen Sie?

Bei den informierten Suchverfahren unterscheidet man zwischen lokalen Suchverfahren und globalen Suchverfahren. Während globale Suchverfahren sich auch für einen Weg zur Lösung interessieren, ist dieser lokalen Suchverfahren irrelevant.

Lokale Suchverfahren

- **Hill Climbing** (manchmal auch „Greedy Local Search“ oder „Gradient Ascent“): Beginnt mit einer beliebigen Lösung und versucht diese nach bestimmten Regeln zu verbessern, d.h. man wählt den Knoten der weiter „bergauf“ liegt aber merkt sich keinen Pfad dahin. Problematisch bei lokalen Optima und Plateaus (vgl. den Versuch, den Mount Everest bei dichtem Nebel zu ersteigen).
- **Gradient Descent**: Versucht mittels *umgekehrtem Hill Climbing* Minima zu finden
- **Simulated Annealing**: Wie Hill Climbing, aber erlaubt auch „schlechtere Aktionen“ mit gewisser Wahrscheinlichkeit, solange die *Temperatur* hoch genug ist.

Globale Suchverfahren

- **Greedy Best-First Search**: Wählt immer Knoten mit kleinstem heuritischem Wert. Unvollständig bei Schleifen, nicht optimal. Ansonsten wie Tiefensuche.
- **A***: Wählt Knoten n mit geringster Summe $f(n)$ aus bekannten Kosten bis dorthin $g(n)$ und *geschätzten* Kosten von dort bis zum Ziel $h(n)$. Vollständig bei *uniformen* Kosten und bei *zulässiger Heuristik* optimal.
- **Iterative Deepening A***: A* mit f-Kosten cutoff.
- **Simplified Memory Bounded A***: Lädt so viele Knoten wie in den Speicher passen und verwirft beim Expandieren eines neuen Knotens den schlechtesten bekannten.

2.2.20 Unter welchen Bedingungen ist A* vollständig, unter welchen optimal?

A* ist für endliche Verzweigungsgrade bei *uniformen* Kosten vollständig. Optimal ist A* nur, wenn die verwendete Heuristik *zulässig* ist, d.h. die Kosten zum Ziel niemals überschätzt (i.Z. $\forall n \in V$ muss gelten: $h(n) \leq h^*(n)$, wobei $h^*(n)$ die tatsächlichen Kosten von n zum Ziel bezeichnet). Mit einer zulässigen Heuristik ist A* sogar *optimal effizient*, d.h. es expandiert minimal viele Knoten.

2.2.21 Welche Komplexität hat der A*-Algoritmus?

A* basiert auf Uniform Cost Search und ist somit im Worst Case sowohl in Zeit als auch im Platz *exponentiell* bzgl. des kürzesten Pfads zu einer Lösung. Wenn man allerdings in einem Baum sucht und der *Fehler der Heuristik logarithmisch klein* ist, i.Z. $|h(x) - h^*(x)| = O(\log h(X))$, dann ist die Lautzeit zumindest polynomiell.

2.2.22 Erklären Sie Iterative Deepening A*!

IDA* ist grundsätzlich Iterative Deepening, jedoch nicht mit einem Tiefenlimit, sondern mit einem $f(n)$-*Kosten-Limit*. In jedem Iterationsschritt werden die Söhne der soeben expandierten Knoten betrachtet und der günstigste verwendet um das neue $f(n)$-Kosten-Limit festzusetzen. IDA* hat den Nachteil, dass es in der Regel mehr Knoten als A* expandiert, dafür jedoch nur die *Platz-Komplexität der Tiefensuche* (bzw. Iterative Deepening) besitzt.

2.2.23 Erklären Sie Simplified Memory Bounded A*!

SMA* ist eine Variante der A* Suche mit einem *Speicherlimit*. Hierbei wird immer der gesamte zur Verfügung stehende Speicher verwendet indem beim Expandieren eines neuen Knoten ein „schlechter“ alter Knoten vergessen wird. Ein gravierender Nachteil dieser Methode ist, das Probleme, die (theoretisch) mit dem A* Algorithmus lösbar sind durch die Speicherbegrenzung zu einer exponentiellen Laufzeit führen können, weil zuvor vergessene „schlechte “ Knoten ggf. neu geladen werden müssen. SMA* ist

- vollständig, falls eine Lösung in den Speicher passt und
- optimal, falls eine optimale Lösung in den Speicher passt.

2.3 Spiele

2.3.1 Wie funktioniert Minimax?

Betrachtet wird der vollständige Spielbaum zwischen zwei Spielern MIN und MAX. Zunächst werden die Terminalknoten durch die Utilityfunktion bewertet. Anschließend werden deren Werte wie folgt durch den Baum hochpropagiert:

- Ist ein Knoten auf MAX-Ebene (also ist MAX am Zug) erhält er den Wert des Maximums seiner Söhne.
- Ist der Knoten auf MIN-Ebene, erhält er das Minimum seiner Söhne.

Am Wurzelknoten steht dann die Punktzahl die MAX höchstens erreichen kann, wenn MIN rational spielt.

2.3.2 Wie komplex ist Minimax?

Da Minimax sich wie eine *Tiefensuche* verhält benötigt es Zeit $O\left(b^d\right)$ und Platz $O(b \cdot d)$.

2.3.3 Was ist wenn Min nicht rational, d.h. nicht optimal spielt, mit dem Wert an der Wurzel?

Spielt MIN nicht rational, ist die Punktzahl am Wurzelknoten nicht der Wert den MAX erhält, sondern das Minimum, das er erreichen kann.

2.3.4 Welches Problem kann auftreten, wenn man die Suche abbricht?

Es könnte passieren, dass wichtige Ereignisse (beispielweise der Verlust einer Dame beim Schach) *über den Such-Horizont fallen* und somit nicht berücksichtigt werden („Horizon Problem"). Dieses Problem kann bei der α-β-Suche nicht auftreten, weil

- MIN so ein Problem hochpropagieren würde und
- MAX so einen Teilbaum nicht auswählen würde.

2.3.5 Wie funktioniert α-β Suche?

Der ziehende Spieler untersucht keine Teilbäume, die „schlecht" aussehen, d.h. den anderen Spiele in eine bessere Lage bringen würden. Realisiert wird dies durch α-β-Fenster - Gültigkeitsintervalle an den Knoten der Bäume, welche die „Worst-Case Szenarien" (Mindestwerte) für Spieler MAX und MIN beschreiben. Mit diesen kann für jeden Knoten getestet werden, ob sein Teilbaum betrachtet werden muss oder nicht. Der Nutzen dieses Verfahrens ist, dass Teilbäume, die auf keinen Fall gewählt werden - egal wie sie ausgewertet würden - verworfen werden.

2.3.6 Wie ist die Komplexität der α-β Suche?

α-β-Suche basiert auf der *Tiefensuche* und ist daher genau wie diese abzuschätzen, d.h. in Zeit $O\left(b^d\right)$ und Platz $O\left(b \cdot d\right)$.

2.3.7 Wie ist die optimale Knotenfolge für α-β Suche?

Die Knoten sind *von links nach rechts aufsteigend* sortiert.

2.3.8 Was ist Expectiminimax?

Dies ist ein Minimax-Algorithmus für Spiele mit Zufallselement. Anstatt nur die Werte der Knoten zu betrachten, betrachtet Expectiminimax die über die *Erwartungswerte* gewichteten Werte der Knoten. Der Spielbaum wird hierzum einen weiteren Knotentyp („DICE") angereichert.

2.4 Planen

2.4.1 Wie unterscheiden sich Planung und Suche?

Suche betrachtet Zustände als Black-Boxen, Planung interessiert sich für deren Komponenten. Während Suche immer alle Knoten expandiert, werden bei Planung nur einige Knoten expandiert. Zustände müssen für die Planung nicht immer vollständig spezifiziert werden. Während Suche nach einer total geordneten Sequenz von Aktionen sucht, die den Startzustand in einen Endzustand transformiert, sucht Planung nach einer (möglicherweise partiell geordneten) Beschreibung eines Plans.

2.4.2 Was ist ein STRIPS-Operator?

Ein STRIPS-Operator ist ein Tripel (Name, Vorbedingungen, Effekte), wobei die Vorbedingungen ausschließlich positive, funktionsfreie Literale enthalten dürfen und die Effekte positive und negative, funktionsfreie Literale enthalten dürfen. Funktionsfrei beinhaltet hierbei *nicht konstantenfrei*.

2.4.3 Was kann man STRIPS-Operatoren vereinfachen?

Man kann die Effekte in ADD- und DELETE-Liste unterteilen und hat es dann nur noch mit positiven Literalen zu tun. Findet man ohne DELETE-Liste einen Plan, dann auch mit dieser, d.h. diese Abstraktion läßt die Vollständigkeit der Planung unbeschadet.

2.4.4 Was ist der Suchraum bei STRIPS-Planung?

Der Suchraum bei STRIPS ist die Menge der STRIPS-Operatoren, da in jedem Schritt ein Operator gesucht wird, der MINDESTENS EINE UNERFÜLLTE VORBEDINGUNG erfüllt.

2.4.5 Was ist ein Threat bei der POP-Planung?

Ein Threat ist ein Planschritt, der potentiell die kausale Verbindung zweier Planschritte ($S_i \xrightarrow{c} S_j$) zerstören kann. Ein solcher Planschritt muss dann entweder nach dieser Verbindung ($S_i \xrightarrow{c} S_j \xrightarrow{c} S_k$, „Promotion“) oder vor dieser Verbindung ($S_k \xrightarrow{c} S_i \xrightarrow{c} S_j as$, „Demotion“) eingefügt werden.

2.4.6 Was ist ein POP-Plan?

Ein Plan der bezüglch der Reihenfolge der Planschritte „least committed“ ist. Formal ist dies ein Tripel aus einer

- Menge von **Planschritten** (STRIPS-Operatoren) zusammen mit einer partiellen Ordnung über diesen. Die partielle Ordnung $S_i \prec S_j$ sagt dabei aus, dass der Planschritt S_i vor dem Planschritt S_j ausgeführt wird,
- Menge von **Variablenzuweisungen** $[x/t]$ und einer

- **Kausalen Relation** ($\xrightarrow{c}$), wobei $S_i \xrightarrow{c} S_j$ bedeutet, dass S_i die Vorbedingung c von S_j erfüllt.

2.4.7 Wann ist ein Plan eine Lösung?

Ein Plan ist eine Lösung wenn er *konsistent* und *vollständig* ist. Ein Plan ist konsistent, wenn die „Unique Name Assumption“ gilt und zusätzlich gilt wenn $S_i \prec S_j$ dann nicht $S_j \succ S_i$, d.h. die STRIPS-Operatoren *wohlgeordnet* sind. Ein Plan ist vollständig, wenn alle Vorbedingungen aller Planschritte erfüllt sind.

2.4.8 Wie sieht der Anfang beim POP-Algorithmus aus?

Man hat zwei STRIPS-Operatoren, START und FINISH, wobei START ausschließlich aus Effekten und FINISH ausschließlich aus Vorbedingungen besteht. Zusätzlich muss eine partielle Ordnung START $\prec$ FINISH existieren. Zu beachten ist, dass der Startzustand ausschließlich aus funktionsfreien Grundliteralen (mit Ausnahme von Konstanten) besteht.

2.4.9 Was ist der Suchraum beim POP-Planning?

POP sucht in der Menge aller möglichen partiell geordneten Pläne indem es Kandidatenpläne transformiert.

2.4.10 Wie komplex sind STRIPS- und POP-Planning?

Jeweils PSPACE-vollständig, wobei es für STRIPS Einschränkungen geben soll unter denen das nur noch NP-vollständig ist.

2.5 Unsicheres Wissen

2.5.1 Wie lautet die Bayes-Formel?

$$P(K|S) = \frac{P(S \wedge K)}{P(S)} = \frac{P(S|K) * P(K)}{P(S)}$$

Hiermit lässt sich beispielsweise vom Symptom „S = Halsschmerzen“ auf die Krankheit „K = Meningitis“ schließen.

2.5.2 Warum ist es vorteilhaft diesen Zusammenhang zu kennen?

Weil sich dieser *Zusammenhang in der Realität selten bis nie ändert.* Die Ergebnisse einer statistischen Erhebung $P(K|S)$, wie häufig eine Krankheit K bei einem Symptom S ist wären nutzlos, sobald sich die a-priori Wahrscheinlichkeit $P(K)$ für die Krankheit ändert.

2.5.3 Wie lautet die Formel für das Bayes-Update und wofür benötigt man sie in der KI?

$$P(A|B,C) = P(A|B) * \frac{P(C|A)}{P(C|B)} \text{ (Folien) bzw.}$$
$$P(A|B,C) = \alpha * P(B|A) * P(C|A) * P(A) \text{ (AIMA)}$$

Sie wird in der KI verwendet um einer bestehenden Wahrscheinlichkeit einen weiteren Beleg hinzufügen, so zum Beispiel in der Diagnose (beim Schließen von Effekten auf deren Ursachen) oder in der Lokalisierung (beim Hinzufügen von Sensordaten). Hierbei erspart dieses Update die Verwendung von größeren gemeinsamen Verteilungen (hier konkret: $P(B,C|A)$ aus Bayes-Regel).

2.5.4 Wann kann man das Bayes-Update einsetzen?

Das Bayes-Update ist nur dann anwendbar, wenn die Belege voneinander *stochastisch unabhängig* sind, d.h. $P(B|A,C) = P(B|A)$, also B von C unabhängig ist, falls A gegeben ist.

2.5.5 Und wie handhabt man das Bayes-Update in der Praxis?

Das Bayes-Update liefert in der Praxis auch bei Verletzung der stochastischen Unabhängigkeit sehr gute Ergebnisse.

2.5.6 Was ist eine gemeinsame Verteilung?

Eine gemeinsame Verteilung ist die Zuweisung von Wahrscheinlichkeiten zu *jeder Kombination der Zufallsvariablen* aus der Verteilung. Die Verteilung ist *normiert*, d.h. alle Wahrscheinlichkeiten addieren sich zu 1 und atomare Ereignisse schließen sich aus. Eine gemeinsame Verteilung kann man als Tabelle notieren. Das Problem hierbei ist, dass die Tabelle exponentiell in der Anzahl der Zufallsvariablen wächst.

2.5.7 Wie kann man die Repräsentation gemeinsamer Verteilungen vereinfachen?

Anstatt alle Wahrscheinlichkeiten zu repräsentieren, beschränkt man sich auf die *kausalen Relationen* und die Wahrscheinlichkeiten der Zufallsvariablen, die Einfluß aufeinander haben. Eine solche Darstellung (als gerichteter azyklischer Graph) heißt Believe Network.

2.5.8 Erklären Sie mal, wie ein Believe Network aufgebaut ist.

Ein Believe Network ist ein gerichteter azyklischer Graph, dessen Knoten Zufallsvariablen und dessen Kanten kausale Verbindungen sind. Zusätzlich enthält jeder Sohnknoten X eine CPT („Conditional Probability Table“), in der alle Wahrscheinlichkeiten der Form $P(X|\text{Parents}(X))$ eingetragen sind.

2.5.9 Wann ist ein Believe Network die korrekte Repräsentation einer gemeinsamen Verteilung?

Ein Believe Network ist dann eine korrekte Darstellung einer gemeinsamen Verteilung, wenn für alle Knoten X_i gilt: $P(X_i|X_{i-1}, \ldots, X_1) = P(X_i|\text{Parents}(X_i))$ und $\text{Parents}(X_i) \subseteq \{X_1, \ldots, X_{i-1}\}$.

2.5.10 Was ist d-Separation?

d-Separation ist ein Verfahren um die stochastische Unabhängigkeit von Mengen von Zufallsvariablen festzustellen. Zwei Mengen von Zufallsvariablen X und Y werden von einer Menge Belege E d-Separiert gdw. jeder *ungerichtete* Pfad zwischen X und Y von E blockiert wird. Ein Pfad wird dann von E blockiert, wenn ein Knoten Z existiert mit:

1. $Z \in E$ und beide Kanten des Pfades gehen aus Z heraus ($X \leftarrow Z \rightarrow Y$).
2. $Z \in E$ und eine Kante des Pfades geht in Z herein, die andere heraus ($X \rightarrow Z \rightarrow Y$).
3. $Z \notin E$, keiner seiner Söhne ist in E und beide Kanten gehen in Z herein ($X \rightarrow Z \leftarrow Y$).

2.5.11 Wie verwendet man d-Separation um stochastische Unabhängigkeit zu zeigen?

Nach dem Theorem von Judea Pearl gilt, dass wenn zwei Mengen X und Y durch E d-separiert werden, dann sind sie unter der Bedingung, dass E gegeben ist auch stochastisch unabhängig. Das Verfahren ist *nicht vollständig*, aber *in Polynomzeit berechenbar*.

2.5.12 Was muss für ein Believe Network gelten, damit der Inferenzalgorithmus effizient arbeiten kann?

Believe Networks in denen der Inferenz-Algorithmus effizient arbeiten kann müssen *singly-connected* (d.h. für jedes Paar Knoten existiert höchstens ein ungerichteter Pfad zwischen diesen) sein. In Graphen die dieser Eigenschaft nicht genügen benötigt der Inferenzalgoritmus statt polynomieller Zeit exponentielle Zeit.

2.5.13 Wie konstruiert man ein Believe Network?

Zunächst wählt man alle relevanten Zufallsvariablen aus und ordnet sie. Sei ZV diese geordnete Menge von Zufallsvariablen $\{X_1, \ldots X_n\}$. Dann nimmt man jeweils den ersten Knoten aus der Menge ZV, berechnet für Ihn die minimale Menge $\text{Parents}(X_i)$ und fügt die Kanten entsprechend ein. Dieses Verfahren iteriert man, bis die Menge ZV leer ist.

2.5.14 Was passiert, wenn die Ordnung schlecht gewählt wurde?

Wenn die Ordnung der Zufallsvariablen schlecht gewählt wurde kann das Netz sehr groß werden.

2.5.15 Welche Arten von Inferenz gibt es in einem Believe Network?

- **Diagnostisch**: Von Effekten zu Ursachen.
- **Kausal**: Von Ursachen zu Effekten.
- **Interkausal**: Zwischen Ursachen eines gemeinsamen Effekts.
- **Mixed**: Kombination von 2 oder mehr der vorher genannten Verfahren.

2.5.16 Wie komplex ist die Inferenz in einem Believe Network?

Bei Polytrees (Singly-Connected) ist eine Anfrage mit polynomiellem Aufwand lösbar (durch d-Separation), bei beliebigen Netzwerken ist der Aufwand exponentiell.

2.5.17 Was versteht man unter „Marginalisierung"?

Unter Marginalisierung versteht man die Berechnung einer Wahrscheinlichkeit $P(A)$ durch Addition aller Wahrscheinlichkeiten der gemeinsamen Verteilung in denen A vorkommt, beispielsweise $P(A) = P(A|B) + P(A|\neg B)$.

2.5.18 Was versteht man unter „Konditionalisierung"?

Konditionalisierung ist ein Sonderfall der „Marginalisierung". Hierbei spaltet man einee Wahrscheinlichkeit auf ihre *bedingten Disjunkte* auf. Zum Beispiel: $P(A) = P(A|B) \cdot P(B) + P(A|\neg B) \cdot P(\neg B)$.

2.5.19 Wann kann man „Variablenelimination" verwenden?

Man kann Variablen Y eliminieren, die bezüglich eine Anfrage $P(X|E)$ *irrelevant* sind, d.h. $Y \notin$ Vorgänger $(\{X\} \cup E)$.

2.6 Lernen

2.6.1 Erklären sie die Informationstheorie!

In der Informationstheorie wird der *Informationsgehalt einer Aussage in Bits* berechnet. So seien also $v_1, \ldots, v_n$ Antworten und $P(v_1), \ldots, P(v_n)$ ihre Auftretenswahrscheinlichkeiten. Dann ist ihr Informationsgehalt $I(v_1, \ldots, v_n) = \sum P(v_i) \cdot \log_2(P(v_i))$.

2.6.2 Wozu kann man die Informationstheorie verwenden?

Man kann die Informationstheorie verwenden um das *beste Teilungsattribut* beim DT-Learning zu bestimmen:

1. Bestimme zuerst wie viel Information benötigt wird, falls man als nächstes das Attribut A_{i+1} auswählt $\rightarrow$ Remainder (A_{i+1}).

2. Berechne hieraus den Nutzen $\rightarrow$ Gain (A_{i+1}) = Insgesamt notwendige Informationen - Remainder (A_{i+1}).

2.6.3 Welche Arten des Feedbacks gibt es beim Lernen?

Man unterscheidet beim Lernen zwischen

- **Supervised Learning**: Agent kennt die optimale Ausgabe bzw. das optimale Verhalten ($\approx$ Lernen mit Lehrer).
- **Reinforcement Learning**: Agent bekommt Feedback nur in Form von *Belohnung* und *Bestrafung* ($\approx$ Haustier trainieren).
- **Unsupervised Learning**: Agent bekommt keine Informationen über die Güte seiner Handlungen und muss seine Erwartungen über diese selbst berechnen ($\approx$ Learning by doing).

2.6.4 Was ist „Induktives Lernen"?

Beim Induktiven Lernen, lernt ein Agent *das Allgemeine aus dem Speziellen* (den Beispielen). Dies geschieht durch Tupel (Eingabe x, Ausgabe $f(x)$), wobei der Agent versucht die optimale Funktion $f(x)$ durch Hypothesen $h(x)$ möglichst exakt anzunähern.

2.6.5 Welche Lernverfahren kennen Sie?

In der Vorlesung wurden folgende Lernverfahren vorgestellt:

- **Decision Trees**: Lernen beliebige boolesche Funktionen.
- **Decision Lists**: Wie DTs, aber benötigen weniger Beispiele wenn eingeschränkt auf k Literale.
- **Neuronale Netzwerke**: Lernen linear separierbare (Perzeptron), stetige oder beliebige Funktionen ($\geq$ 2 Hidden Layer).

2.6.6 Für welche Probleme eignen sich die einzelnen Lernverfahren?

Decision Trees und Decision Lists eignen sich beide für *attributbasierte Probleme* (boolesche Funktionen), bei denen die Attribute eine *möglichst gute Trennung* der Beispiele erlauben. Bei attributbasierten Problemen, bei denen jedes Attribut einen ähnlich schlechten Informationsgehalt hat (z.B. Mehrheitsfunktion) sind Neuronale Netzwerke vorzuziehen.

2.6.7 Welche Vorteile haben Decision Lists, welche Vorteile haben Decision Trees?

- Decision Trees sind sehr *transparent*, *übersichtlich* und *leicht verständlich.*
- Decision Lists erlauben eine *kompaktere Darstellung* und *benötigen weniger Beispiele* (nämlich nur polynomiell viele) als ein unbeschränkter DT um eine bestimmte Funktion zu lernen.
- Dafür stellen k-DL eine echte *Einschränkung* (nur noch Funktionen mit maximal k Literalen darstellbar) dar.

2.6.8 Wie lernt man mit Decision Trees?

Man betrachtet eine Menge von Beispielen die man lernen möchte.

- Gibt es nur noch positive oder negative Antworten, antworte entsprechend.
- Gibt es noch sowohl positive als auch negative Antworten suche gemäß der Informationstheorie das beste Teilungsattribut, teile die Menge und beginne von vorne.

Das beste Teilungsattribut findet man hierbei wie folgt:
Zunächst berechnet man den Informationsgehalt I des Zielattributs, bzw. des letzten Teilungsattributs. Dieses sei mit I bezeichnet. Dann berechnet man für jedes Attribut X, $\text{Gain}(X) = I - \text{Remainder}(X)$. Der Remainder ist dabei der gewichtete Informationsgehalt der einzelnen Werte die das Attribut annehmen kann oder vereinfacht gesagt, der Remainder ist der Informationsgehalt, der nach Teilung der Beispiele durch X erhalten bleibt.

2.6.9 Warum lässt sich die Mehrheitsfunktion mit Decision Trees nur sehr schlecht darstellen?

Die Mehrheitsfunktion gibt an, ob die Mehrheit der Attribute positiv ist. Beim Decision Tree Learning versucht man Attribute zu finden, welche die Beispielmenge möglichst in Mengen aufteilen, die nur Beispiele einer Klassifikation enthalten. So etwas lässt sich hierfür nicht finden, da *alle Attribute gleich schlecht* sind.

2.6.10 Wie groß wird der Decision Tree für eine Mehrheitsfunktion der Größe n?

Er bekommt die Größe 2^n (vollständiger Baum).

2.6.11 Was ist „PAC-Learning“?

Die Idee hinter PAC Learning ist, dass eine falsche Funktion nach hinreichend großer Anzahl Beispiele als solche „auffliegt“. Mittels PAC Learning kann man die *minimale Anzahl an Beispielen* bestimmen, die notwendig ist um die optimale Funktion in einer ϵ-Umgebung („ϵ-Ball“) mit Wahrscheinlichkeit $(1 - \delta)$ anzunähern.

2.6.12 Warum ist die Aussage, dass korrektes Lernen eigentlich unmöglich ist, richtig?

Man kann mittels PAC-Learning zeigen, dass man theoretisch *exponentiell viele* (d.h. fast alle) Beispiele zeigen müsste. In der Realität sind die Probleme meist einfacher, so dass man maschinelles Lernen trotzdem verwenden kann.

2.6.13 Wie viele Beispiele benötigt man bei Decision Trees, wie viele bei einer k-Decision List?

Grundsätzlich benötigt man *logarithmisch in der Größe des Hypothesenraums* viele Beispiele, i.Z.: $N \geq \frac{1}{\epsilon} \cdot \left(\ln \frac{1}{\delta} + |H|\right)$

- Für einen DT ist der Hypothesenraum $|H| = 2^{2^n}$ groß, so dass insgesamt 2^n, d.h. fast alle, Beispiele benötigt werden.
- Für k-DL ist der Hypothesenraum lediglich $|H| = 2^{O\left(n^k \cdot \log_2\left(n^k\right)\right)}$ groß, so dass man insgesamt nur *polynomiell* viele Beispiele benötigt werden, nämlich: $N \geq \frac{1}{\epsilon} * \left(\ln \frac{1}{\delta} + O\left(n^k * \ln\left(n^k\right)\right)\right)$, mit n = Anzahl Attribute und k = Anzahl Literale pro Test (AIMA, S. 671)

2.6.14 Woraus besteht ein Neuronales Netzwerk?

Neuronen mit *Aktivierungsfunktion*, beispielsweise

- Threshold (nicht differenzierbar),
- Sign_n (nicht differenzierbar) ,
- Sigmoid

so wie *Kanten* zwischen den Neuronen und deren und *Kantengewichtungen*.

2.6.15 Welche Arten von Neuronalen Netzwerken gibt es und wie unterscheiden sich diese?

- **Feed-Forward**: Gerichtete azykliche Graphen, d.h. jede Schicht ist nur mit der nachfolgenden Schicht verbunden.
- **Recurrent**: Erlauben Verbindungen zu vorigen Schichten.

2.6.16 Welche Funktionen kann man durch ein Feed-Forward Network mit keinem / einem / zwei Hidden Layern darstellen?

Ohne Hidden Layer (= Perzeptron) lassen sich nur die linear separierbaren (u.a. die Mehrheitsfunktion), mit einem Hidden Layer die stetigen und mit zwei Hidden Layern alle Funktionen darstellen.

2.6.17 Weswegen lassen sich ohne Hidden Layer nur die linear separierbaren Funktionen darstellen?

Die Ausgabe eines Perzeptrons $\sum w_j \cdot x_j$ spannt eine „Hyperebene“ im Eingaberaum auf, welche die positiven Ausgaben von den negativen Ausgaben trennt. Funktionen wie XOR lassen sich auf diese Art nicht trennen.

2.6.18 Was sind die Vor- und Nachteile von Neuronalen Netzwerken?

Vorteile

- Können gut mit *verrauschten Daten* umgehen.
- Eignen sich herrvorragend für linear separierbare Funktionen.
- *Robustheit* („graceful degradation“).
- Massive *Parallelisierbarkeit*.
- *Perzeptrons konvergieren immer* gegen ein globales (Fehler-)Minimum

Nachteile

- Im Allgemeinen ($>$ Perzeptron) können Konvergenz und Effizienz beim Lernen nicht garantiert werden.
- Es gibt keine Heuristiken für die Wahl des richtigen Netzes (Großes Netz = keine Generalisierung, kleines Netz = Darstellbarkeit).
- Es ist nicht möglich zusätzliches Wissen zu integrieren.

2.6.19 Was kann passieren, wenn man das Netz zu groß, bzw. zu klein wählt?

Wenn man ein Neuronales Netzwerk zu groß wählt, wird das Rauschen mit gelernt, d.h. das Netz kann nur sehr *schlecht generalisieren*. Ist das Netz zu klein, kann es sein, dass die *Funktion nicht mehr darstellbar* ist.

2.6.20 Was bedeutet Lernen bei Neuronalen Netzwerken?

Wir suchen nach geeigneten Parametern, also den Kantengewichten, mit denen sich die gesuchte Funktion gut annähern lässt.

2.6.21 Warum konvergiert das im Allgemeinen nicht?

Wir wollen ja den Fehler minimieren. Wenn wir as als „Gradient Descent Search“ auffassen, dann kann es auch *lokale Minima* geben, in denen wir festhängen.

2.6.22 Wie könnte man das Lernen in Neuronalen Netzwerken verbessern?

Dem Festhängen an lokalen Minima könnte man durch „Random-Restart“ und „Simulated Annealing“ entgegenwirken.

2.6.23 Wie aufwendig ist das Lernen in Neuronalen Netzwerken?

Das Lernen in Perceptrons konvergiert immer, soweit die Funktion darstellbar ist. Ferner ist der Aufwand polynomiell in der Anzahl der Beispiele. Im Allgemeinen muss jedoch der *Back-Propagation* Algorithmus verwendet werden, der weder Konvergenz noch Effizienz garantiert, sondern eine exponentielle Anzahl *Epochen* benötigt. Eine Epoche besteht aus der Eingabe aller Beispiele ins Netzwerk und entsprechender Korrektur der Kantengewichte.

2.6.24 Wie lautet die Update-Regel für Perceptrons?

Die Update-Regel („Hebbsche Lernregel“) lautet:

$$W_{i,j} = W_{i,j} + \alpha \cdot I_j \cdot (t - o),$$

wobei α die Learning Rate, I_j der Input, t der erwartete Output und o der tatsächliche Output sind.

2.6.25 Wie funktioniert der Back-Propagation Algorithmus?

Der Back-Propagation Algorithmus funktioniert wie folgt:

1. Ein Eingabemuster wird angelegt und vorwärts *durch das Netz propagiert.*
2. Die Ausgabe des Netzes wird mit der gewünschten Ausgabe verglichen. Die Differenz der beiden Werte wird als *Fehler berechnet.*
3. Aus dem Fehler berechnet man unter Berücksichtigung von Lernrate und Eingabe (siehe oben) die *Gewichtsveränderungen der Kanten.*
4. Der Fehler wird nun wieder über die Ausgabe- zur Eingabeschicht *zurück propagiert*, dabei werden die Gewichtungen der Neuronenverbindungen abhängig von ihrem Einfluss auf den Fehler geändert. Dies garantiert bei einem erneuten Anlegen der Eingabe eine Annäherung an die gewünschte Ausgabe.
   Hierbei werden für jedes Beispiel die Gewichte wie folgt aktualisiert:
   - Auf dem Output Layer:
     $W_{i,j} = W_{i,j} + \alpha \cdot a_j \cdot \text{Error}_i \cdot g'(\text{In}_i)$
   - Für jedes folgende Layer:
     $W_{k,j} = W_{k,j} + \alpha \cdot I_k \cdot \Delta_j$ mit
     $\Delta_j = g'(\text{In}_j) \cdot \sum_i W_{i,j} \cdot \Delta_i$

   d.h. jeder Knoten j wird mit Δ_j der gesamte durch die Schicht j verursachte Fehler, gewichtet um die Eingabe In_j dieses Knotens, aktualisiert.

2.6.26 Kennen Sie eine Heuristik um die Anzahl der Kanten in einem Neuronalen Netzwerk anzugeben?

Da gibt es die Heuristik des „Optimal Brain Damage“. Hier beginnt man mit einer maximalen Anzahl von Kanten und streicht dann Kanten die gemäß der Informationstheorie zu vernachlässigen sind.

2.7 Knowledge Bases

2.7.1 Nennen sie die drei Ebenen der KR und erläutern sie diese!

- **Knowledge Level**: Abstraktes Wissen.
- **Symbolic Level**: Kodiertes Wissen in einer Logiksprache.
- **Implementation Level**: Implementierung (beispielsweise Bitmuster).

2.7.2 Warum ist Inferenz bei Knowledge Bases schwerer als bei Datenbanken?

Das liegt daran, dass Datenbanken annehmen *vollständiges Wissen* über die Welt und deren Individuen zu besitzen, also Inferenz mittels CWA betreiben. Weiterhin enthalten Datenbanken regelmäßig keine Variablen.

2.7.3 Was ist die „Closed World Assumption“ und was bewirkt sie?

Unter der CWA nimmt man an, dass ein *variablenfreier Grundterm* als falsch gilt, sofern er nicht explizit als wahr angegeben ist. D.h. man nimmt an die Wahrheit über alle in der KB erwähnten *Grundterme* zu kennen, so dass das Beweisen von komplexen Sätzen auf „nachschlagen“ von Fakten reduziert werden kann. Um All- oder Existenzaussagen treffen zu können, benötigt man zusätzlich die „Domain Closure“, welche annimmt, dass nur die *Grundterme* der KB existieren. Zuletzt benötigt man die „Unique Names Assumption“ um über mehr als eine Entität schließen zu können. Beispielsweise führt die Anfrage „Wie viele Studenten haben kein Hobby?“

- Ohne CWA zur Antwort: Zwischen einem und ∞.
- Mit CWA und UNA zur Antwort: Mindestens 10.
- Mit CWA, UNA und DC: Genau 10.

2.7.4 Definieren Sie die Closed World Assumption $\models_c$!

Ein Satz α ist unter der CWA aus einer KB ableitbar $\mathrm{KB} \models_c \alpha \Leftrightarrow \mathrm{KB} \cup \mathrm{Negs} \models \alpha$, mit $\mathrm{Negs} = \{\neg p \mid p \text{ ist atomar und } \mathrm{KB} \models p\}$

2.7.5 Wie wird einen Anfrage unter der CWA ausgewertet?

Unter der CWA werden komplexe Anfragen auf atomare reduziert, zum Beispiel:

$$\text{KB} \models_c (\alpha \vee \beta) \Leftrightarrow \text{KB} \models_c \alpha \text{ oder KB} \models_c \beta.$$

2.7.6 Was können Sie zur Konsistenz unter der CWA sagen?

Die Konsistenz einer CWA-KB ist wichtig, da aus dieser sonst alles gefolgert werden könnte. Probleme entstehen, wenn eine CWA-KB Disjunktionen ($\alpha \vee \beta$) enthält, aber keine Fakten die diese Diskjunktionen stützen (hier α oder β).

2.7.7 Was kann bei CWA zu Inkonsistenzen führen?

$\text{KB} \models \alpha \vee \beta$, aber $\text{KB} \not\models \alpha$ und $\text{KB} \not\models \beta$

2.7.8 Was ist mit der Monotonie unter der CWA?

Normalerweise ist das Schließen über konsistente Wissensbasen monoton, d.h. *neue Fakten invalidieren keine alten Folgerungen.* Das Schließen unter CWA ist hingegen nichtmonoton. Beispielsweise gilt $P \models_c \neg Q$, aber $\{P, Q\} \not\models_c \neg Q$.

2.7.9 Was macht man mit Anfragen, die Quantoren enthalten?

Domain Closure annehmen. Damit werden alle Entitäten der Welt als bekannt angenommen und man kann Existenz- und Allaussagen über diese treffen.

2.8 Roboter

2.8.1 Wie konstruiert ein Roboter seine „Occupancy Grid Map“?

Jedem Feld des zugrunde liegenden Rasters wird eine Wahrscheinlichkeit, basierend auf der relativen Anzahl der von dort reflektierten Signale zugewiesen.

2.8.2 Wie funktioniert Selbstlokalisierung?

Man initialisiert die Wahrscheinlichkeiten an Position l basierend auf der „Occupancy Grid Map“ und integriert dann Bewegungsdaten und Sensordaten in dieses Modell.

2.8.3 Wie werden die Bewegungen integriert?

Die Wahrscheinlichkeit $P(l)$ gerade an *Position* l zu sein ist die Summe der Wahrscheinlichkeiten $P(l')$ zuvor an *Position* l' gewesen zu sein und sich in *Zeit* t in *Richtung* τ zu Position $P(l)$ bewegt zu haben, i.Z. $P(l) := \alpha \sum_{l'} P(l') \cdot P(l|l', \tau, t)$

2.8.4 Wie werden die Sensordaten integriert?

Es geht darum die Wahrscheinlichkeit zu bestimmen, unter einer *Historie von Sensordaten* $s_t, s_{t-1}, \ldots, s_1$ und einer *Karte* m, an *Position* l zu sein. Diese ergibt sich aus den Annahmen

1. bereits zum Zeitpunkt s_{t-1} an Position l gewesen zu sein und
2. zum Zeitpunkt t die Sensordaten s_t unter Position l und Karte m wahrzunehmen.

Formal sieht das dann so aus: $P(l|s_t, s_{t-1}, \ldots, s_1, m) := P(l|s_{t-1}, \ldots, s_1, m) \cdot P(s_t|l, m)$ mit Wahrnehmung s_i zum Zeitpunkt i und Karte m.

2.8.5 Erklären Sie den Algorithmus zur Pfadplanung!

Der Value-Iteration-Algorithmus berechnet iterativ den Nutzen zu allen Felder $V(x, y)$ gegeben den Nutzen der Nachbarn $C(x, y,)$; dies verwendet man beispielsweise zur Einplanung stochastischer Bewegungen.

2.8.6 Was sind die Vorteile des Value-Iteration Algorithmus?

- Der Agent kann sich bereits sinnvoll bewegen, bevor der Algorithmus konvergiert.
- Der Algorithmus berechnet den optimalen Pfad von jedem Punkt der Karte zum Ziel. Sollte der Agent die gewählten Pfad verlassen müssen, besteht kein Grund neu zu planen.

2.8.7 Erklären Sie die Markov-Annahme!

Die aktuelle Situation resultiert nur aus der letzten Situation und der darin ausgeführten Aktion, d.h. sie legt eine *episodische* Welt zu Grunde. Russel und Norvig behaupten hierzu (AIMA, S.539) „. . . current state depends only on a *finite* history of previous states“, also eine weniger starke Annahme.

2.8.8 Was passiert wenn man in einer dynamischen Welt von der Markov-Assumption ausgeht?

Bei leichten Abweichungen funktioniert die Markov Annahme noch, bei großen Abweichungen müssen die Fehler herausgerechnet werden.

3 Wissensrepräsentation

3.1 Schließen

3.1.1 Welche Arten logischen Schließens gibt es?

- **Deduktion**: $A, A \rightarrow B \Rightarrow B$.
- **Abduktion**: $B, A \rightarrow B \Rightarrow A$.
- **Induktion**: $A_1 \rightarrow B_1, \ldots, A_n \rightarrow B_n \Rightarrow A$.

3.1.2 Wie funktioniert Deduktion?

Gegeben eine Knowledge Base KB und einen Satz α: Gilt KB $\models \alpha$?

- **Korrekt**, falls für jede Herleitung KB $\models \alpha$ gilt KB $\rightarrow \alpha$.
- **Vollständig**, falls jedes α mit KB $\rightarrow \alpha$ auch KB $\models \alpha$ herleitbar ist.

3.1.3 Was bedeutet die logische Implikation „KB $\models \alpha$“ ?

Die Wahrheit von α ist bereits implizit in KB vorhanden, d.h. dass jedes Modell von KB auch ein Modell von α ist - unabhängig von der Interpretation. Den Umkehrschluß, dass dann auch KB $\cup \{\neg\alpha\}$ unerfüllbar ist macht man sich bei der Resolution zu nutze.

3.1.4 Wie sieht die Interpretationsfunktion aus?

- $\Phi[P] \subseteq D \times D \times \ldots \times D$.
- $\Phi[f] \subseteq D \times D \times \ldots \times D \rightarrow D$.

3.1.5 Was ist der Unterschied zwischen Wissensbasen und Datenbanken?

Datenbanken nehmen an über *vollständiges Wissen über alle Grundterme* zu verfügen. Deshalb sind Anfragen in Datenbank auch auf atomare Anfragen reduzierbar und so effizient lösbar.

3.1.6 Wofür benötigt man die „Domain Closure“ formell?

Die DC wird benötigt um Anfragen mit Quantoren zu beantworten. Ohne die DC könnte man beispielsweise aus der Satzmenge $\forall\, c_i.\ \neg\text{Flug}(c_i, d)$ nicht folgern, dass $\neg\exists x.\text{Flug}(x, d)$.

3.1.7 Wie stellt man Anfragen, wenn man FO-Logik ohne CWA und DC hat?

Mittels Resolution.

3.1.8 Wie kann man „Negation as Failure“ realisieren?

Mittels „Negation as Failure“ möchte man herleiten können, ob eine Satzmenge S eine Anfrage G implizit nicht erfüllt, d.h. $S \models \text{not}\,(G) \Leftrightarrow S \not\models G$:

- NOT (G) :- G, !, FAIL und
- NOT (G).

3.1.9 Was ist daran nicht monoton?

Wenn wir keine Fakten zu $P\,(a)$ in der KB haben, dann könnten wir $\neg\exists x P\,(x)$ folgern. Das geht nicht mehr, wenn wir $P\,(a)$ als Fakt hinzunehmen.

3.1.10 Wie funktioniert „Forward-Chaining“?

Man „entfaltet“ durch *wiederholte Anwendung des Modus Ponens* („Bottom-Up“) das implizite Wissen der KB und prüft dann ob die Ziele darin liegen („Data Driven“). Das naive Verfahren für atomare Horn-Klauseln ist linear.

3.1.11 Wie funktioniert „Backward-Chaining“?

Man sucht *rückwärts vom Ziel* („Top-Down“) durch die KB nach Fakten und Regeln welche die Ziele stützen („Goal Driven“). Der Aufwand hierfür kann deutlich geringer als linear sein, es kann aber auch sein, dass das Verfahren - wie die *Tiefensuche*, aufgrund unendlicher Pfade (beispielsweise $P \Rightarrow P$) - nicht terminiert.

3.2 Resolution

3.2.1 Wie funktioniert der einzelne Resolutionsschritt?

$$\frac{[C_1 \cup \{p\}], [C_2 \cup \{\neg p\}]}{[C_1 \cup C_2]}$$

3.2.2 Weswegen darf man Resolution verwenden?

Resolutionsschritte sind *durch Interpretationen in der Semantik verankert*, d.h. wenn es eine Interpretation I gibt, für die $I \models (p \vee \alpha)$ und $I \models (\neg p \vee \beta)$, dann gilt entweder

1. $I \models p$, dann gilt auch $I \models \beta$ und damit $I \models \alpha \vee \beta$, oder
2. $I \models \neg p$, dann gilt auch $I \models \alpha$ und damit $I \models \alpha \vee \beta$.

3.2.3 Wie hängen Ableitung $\rightarrow$ und Folgerbarkeit $\models$ zusammen?

- Wenn $S \models c$, dann $S \rightarrow c$, d.h. Resolution ist korrekt.
- Resolution ist unvollständig aber *widerlegungsvollständig*: Es gilt $[P] \models [P, Q]$, aber dies ist nicht durch Resolution herleitbar.

3.2.4 Wie führt man eine Anfrage „KB $\models \alpha$“ durch?

1. Umformen zur **Negation** KB $\wedge \neg\alpha$.
2. **Skolemisierung**: Quantoren herausziehen ($\Rightarrow$ Pränex-NF) und und $\exists$'s durch neue Funktionen ersetzen ($\Rightarrow$ Skolem-NF).
3. *Matrix* von α in **Konjunktive Normalform** (CNF) umwandeln.
4. Mittels **Resolutionschritten** leere Klausel oder Antwortprädikate herleiten.

3.2.5 Welches Problem besteht bei Skolemisierung eines Terms?

Term und Resultat sind *nicht mehr äquivalent*, da durch die Skolemisierung möglicherweise neue Funktionsterme eingeführt wurden. Glücklicherweise sind Term und Resultat aber zumindest *erfüllbarkeitsäquivalent*.

3.2.6 Wie schwer ist Unifikation?

Wegen des „Occur Check“ quadratisch in Größe der zu unifizierenden Ausdrücke.

3.2.7 Welche Fehler können bei Unifikation auftreten?

- **Occur Failure**: Versuch x durch den Term $f(x)$ zu ersetzen.
- **Symbol Clash**: Versuch die Funktionen f und g zu unifizieren.

3.2.8 Was ist „Answer Extraction“?

Man fügt der *bereits negativ umformulierten Anfrage* $\neg\alpha$ ein spezielles Antwort-Prädikat hinzu, welches die gleiche Variable beinhaltet wie die gesuchte Lösung. Durch Unifikation wird die Variable in diesem Prädikat mit der Antwort belegt.

3.2.9 Wie schwer ist AL-Resolution?

AL-Resolution ist *exponentiell entscheidbar*, d.h. *NP schwer*.

3.2.10 Wie schwer ist FO-Resolution?

FO-Resolution ist *NP-vollständig* und *semi-entscheidbar*, d.h. ist die leere Klausel $\square$ ableitbar, wird diese in *exponentieller Zeit* gefunden; existiert diese nicht, muss das Verfahren nicht terminieren (heißt auch „widerlegungsvollständig“). Die Komplexität entsteht durch die *Fallunterscheidungen* die bezüglich der Variablen im Zuge des Beweisens getroffen werden. Resolution genau so schwer wie das *Halteproblem*.

3.2.11 Was kann man bei der Resolution vereinfachen?

- Halte die *Suche möglichst allgemein*, d.h. verwende den „Most General Unifier“(MGU).
- Entferne *unnötige Klauseln*, z.b. L, wenn $\neg L$ nirgendwo vorhanden ist.
- Entferne Klauseln die von anderen *subsummiert* werden.

3.2.12 Wie aufwendig ist es den MGU zu bestimmen?

Der in der Vorlesung gezeigte Algorithmus ist *polynomiell* wegen dem „Occur Check“. Es gibt aber lineare Verfahren.

3.2.13 Welche Strategien gibt es Resolution effizienter zu machen?

Dem *Nichtdeterminismus 1. Art*, d.h. der Auswahl der zu resolvierenden Klausel, begegnet man mit folgenden Einschränkungen der Resolution:

- **Lineare Resolution**: Resolvent aus vorigem Resolventen und beliebiger Klausel.
- **Inputresolution**: Resolvent aus vorigem Resolventen und KB.
- **SLD-Resolution**: Resolvent aus vorigem Resolventen und *definiter* Klausel der KB.

Dem *Nichtdeterminismus 2. Art* d.h. der Auswahl des zu resolvierenden Literals, begegnet man mit *kanonischer* Resolution der Literale von links nach rechts.

3.2.14 Welche Möglichkeiten prozeduraler Kontrolle gibt es?

- Steuerung der **Suchreihenfolge**: Es ist effizienter erst alle Cousins zu suchen und dann unter diesen die Amerikaner, als umgekehrt.
- **Cut**-Operator: Legt die Suche auf Literale vor dem „!“ fest und schränkt damit den Suchraum ein.
- **Golog**: Entwurf von Algorithmen die die Suche lenken.

3.3 Horn Logik

3.3.1 Was ist eine Horn-Formel?

Eine Disjunktion von Literalen mit *höchstens einem* positiven Literal. Damit ensprechen Hornklauseln

- mit einem positiven Literal und beliebig vielen negativen Literalen: *Implikationen* mit positiver Konklusion.
- mit einem positiven Literal ohne negative Literale: *Fakten*.
- ohne positives Literal: *Anfragen*.

3.3.2 Welchen Nutzen haben „Horn-Klauseln"?

Hornklauseln sind meistens hinreichend um das Wissen eines Weltmodells darzustellen und mit Hornklauseln kann man *aussagenlogische* Anfragen durch Forward Chaining in linearem Aufwand beantworten lassen.

3.3.3 Was ist SLD-Resolution?

Korrekte und *widerlegungsvollständige* Resolution auf *Hornklauseln*, bei der immer der letzte Resolvent mit einer *definiten* Klausel der Eingabemenge resolviert wird. Hierbei entstehen immer wieder Horn-Klauseln.

3.3.4 Warum verwendet man SLD-Resolution?

SLD-Resolution ist für Horn-Klauseln korrekt und *widerlegungsvollständig*. Zwar ist SLD-Resolution weiterhin *exponentiell* und *semi-entscheidbar*, allerdings im Average Case deutlich besser.

3.4 Produktionssysteme

3.4.1 Was ist ein „Produktionssystem"?

Ein Produktionssystem ist ein Forward Chaining Reasoner, der mittels „If ... Then ..." Regeln auf „Working Memory Elementen" der Art (type attr_1 val_1 ... attr_n val_n) arbeitet und lediglich die Aktionen Add, Remove und Modify kennt.

3.4.2 Erklären Sie die verschieden Phasen eines Zyklus!

1. **Recognize**: Finde Regeln die feuern können.
2. **Resolve**: Suche eine der Regeln zum feuern aus.
3. **Act**: Führe diese Regel aus.

3.4.3 Wie bestimmt man die Regeln die feuern („Conflict Resolution“)?

Man könnte die Regeln in Reihenfolge der Vorkommens in der Datei, nach Aktualität oder Spezifität feuern lassen.

3.4.4 Welche Vorteile verspricht man sich von solchen Systemen?

- **Modularität**: Regeln unabhängig voneineinander.
- **Transparenz**: Regeln einfach in natürliche Sprache zu fassen.
- **Fine grained control**: Keine aufwendigen Ziel- oder Kontroll-Stacks

3.5 Beschreibungslogiken

3.5.1 Was sind die Vorteile von Konzeptsprachen?

Üblicherweise werden Informationen sequentiell abelegt, z.B. STUDENT(X), ... , MATNR(X,43). Dies macht es schwierig alle Informationen über ein Objekt (hier Student „x“) zu sammeln. Description Logics sind hingegen *objektorientiert*, d.h. sie vereinen alle relevanten Daten über ein Objekt in der Beschreibung. Hiermit lassen sich effizient *Subsumption* und *Konzeptzugehörigkeit* berechnen.

3.5.2 Was ist eine „Role“?

Eine Rolle ist eine zweistellige Relation die eine Beziehung zwischen zwei Entitäten abbildet, beispielsweise CHILDOF, HASCOLOR, OWNS, ...

3.5.3 Was für Operatoren gibt es?

- $\langle \text{ALL } r\ C \rangle$
- $\langle \text{AND } C_1\ C_2 \rangle$
- $\langle \text{AT-LEAST } n\ r \rangle = \langle \text{EXISTS } n\ r \rangle$
- $\langle \text{AT-MOST } n\ r \rangle$
- $\langle \text{FILLS } r\ i \rangle$
- $\langle \text{RESTR } r\ C \rangle$
- $\langle \text{SOME } r \rangle = \langle \text{EXISTS } 1\ r \rangle$ (KRR, S. 329)

3.5.4 Woraus besteht eine DL-KB?

- **Definitionen**: „MENSCH $\doteq$ (AND Lebewesen Reflektionsfähig)“
- **Partial Assertions**: „PHILOSOPH $\sqsubseteq$ MENSCH“
- **Assertions**: „sokrates $\rightarrow$ PHILOSOPH“

3.5.5 Geben sie die Interpretation $I = (D, \Phi)$ formal an!

Eine Interpretation $I = (D, \Phi)$ besteht aus einer Domäne D und einer Interpretationsabbildung Φ mit

- $\Phi\,[\text{Konstante}] \in D$
- $\Phi\,[\text{Konzept}] \subseteq D$
- $\Phi\,[\text{Rolle}] \subseteq D \times D$

3.5.6 Schreiben Sie ALL, AT-LEAST, AND und FILLS formal auf!

- $\Phi\,[\langle \text{ALL } r\ C\rangle] = \{x \in D \mid \forall y.\ \text{if } (x,y) \in \Phi\,[r] \text{ then } y \in \Phi\,[C]\}$
- $\Phi\,[\langle \text{AT-LEAST } n\ r\rangle] = \{x \in D \mid \exists_n x\,(x,y) \in \Phi\,[r]\}$
- $\Phi\,[\langle \text{AND } C_1\ C_2\rangle] = \Phi\,[C_1] \cap \Phi\,[C_2]$
- $\Phi\,[\langle \text{FILLS r i}\rangle] = \{x \in D \mid\ (x, \Phi\,[i]) \in \Phi\,[r]\}$

3.5.7 Wie kann man $\langle$ALL $r\ C\rangle$ nach FOL übersetzen?

$\langle \text{ALL } r\ C\rangle$ enspricht $\forall y\ R\,(x,y) \supset C\,(y)$.

3.5.8 Erklären sie den Operator $\langle$SOME $r\ C\rangle$ von FL$^-$!

Das sind alle Objekte, die über die Rolle r mit mindestens einem Objekt des Konzepts C verbunden sind. Beispielsweise ist $\langle$SOME child STUDENT$\rangle$ die Menge der Objekte, die mindestens ein Kind haben welches studiert.

3.5.9 Schreiben Sie $\langle$SOME $r\ C\rangle$ formal auf!

- In (KRR) gibt es lediglich $\langle$SOME $r\rangle$ als Abkürzung für $\langle$EXISTS 1 $r\rangle$.
- Ein $\Phi\,[\langle \text{SOME } r\ C\rangle]$ entspräche $\{x \in D \mid \exists y.\ (x,y) \in \Phi\,[r] \wedge y \in \Phi\,[C]\}$.
- Letzteres sollte äquivalent zu $\langle$AT-LEAST 1 $\langle$RESTR $r\ C\rangle\rangle$.

3.5.10 Wie berechnet man die Subsumierung?

Mit dem Strukturmatching-Algorithmus berechnet man Subsumption $C \subseteq D \Leftrightarrow \Phi(C) \subseteq \Phi(D)$ in etwa wie folgt:

1. Ausdrücke normalisieren.
2. Für jeden Term des Subsumees $\langle \text{AND } D_1\ D_2 \rangle$ muss ein subsummierender Term im Subsumer $\langle \text{AND } C_1\ C_2 \rangle$ vorhanden sein.

3.5.11 Wo arbeitet der Strukturmatching-Algorithmus rekursiv?

Beim $\langle \text{ALL } r\ C \rangle \sqsubseteq \langle \text{ALL } r\ D \rangle$ ist rekursiv zu prüfen ob C von D subsummiert wird.

3.5.12 Wozu benötigt man dieses Verfahren?

Beispielsweise beim Einordnen eines Konzepts in eine bestehende Hierarchie. Hierbei sucht man sich von der Wurzel herab den „speziellsten Subsummierer und das allgemeinste Subsummierte“.

3.5.13 Wird $\langle$SOME $r\ C\rangle$ von $\langle$ALL $r\ C\rangle$ subsumiert?

- Nein, da $\langle \text{SOME } r\ C \rangle$ auch Entitäten beinhaltet, bei denen einige r nicht C sind.
- Auch umgekehrt gilt dies nicht, da $\langle \text{ALL } r\ C \rangle$ auch zutrifft, wenn es keine Erfüller von r gibt.

3.5.14 Wie sieht der RESTR-Operator von FL formal aus?

Der RESTR-Operator schränkt die Rolle r auf ein Konzept C ein, d.h.

$$\Phi\left[\langle \text{RESTR } r\ C \rangle\right] = \{(x,y) \mid (x,y) \in \Phi\left[\text{r}\right] \text{und } y \in \Phi\left[C\right]\}$$

3.5.15 Wie komplex ist das Subsumption Problem?

Polynomiell (quadratisch) durch „Structure Matching“, sofern man auf RESTR, NOT und AT-MOST verzichtet.

3.5.16 Was hat die Verwendung von RESTR fur Auswirkungen?

Ohne $\langle \text{RESTR } r\ D \rangle$ ist die Berechnung der Subsumption in *quadratischer Zeit* möglich, mit RESTR nur in *exponentieller* Zeit. Dies rührt daher, dass man sich unter Verwendung von RESTR nicht mehr auf „part-by-part-matching“ zurückziehen kann. Die Hinzunahme von RESTR ist also ein Tausch von Ausdrucksstärke gegen Komplexität.

3.5.17 Gibt es weitere störende Operatoren?

- Verwenden wir NOT klappt das strukturelle Matching auch nicht mehr.
- Wenn wir AT-MOST verwenden, ist es nicht mehr vollständig.

3.6 Defaults

3.6.1 Wie ist das „Minimal Entailment“ $\models_m$ definiert?

Beim „Minimal Entailment“ versucht man die Ausnahmefälle (beispielsweise Vögel die nicht fliegen können) in Abnormalitäts-Prädikaten Ab zu fassen („Circumscription“) und schränkt sich dann auf Folgerungen ein, bei denen die Menge der abnormalen Fälle kleinstmöglich ist, d.h.

$$\mathrm{KB} \models_m \alpha \Leftrightarrow \forall\, I.\, \langle I \models \mathrm{KB} \wedge \neg\exists I^*\, (I^* < I \wedge I^* \models \mathrm{KB})\rangle \Rightarrow I \models \alpha,$$

wobei $I^* < I \Leftrightarrow I^*\,[\mathrm{Ab}] < I\,[\mathrm{Ab}]$.

3.6.2 Was hat es mit Reiters „Default Logic“ auf sich?

Default Logic erweitert FOL um „Default Rules“: Tripel aus <VORAUSSETZUNG, BEGRÜNDUNG, SCHLUSSFOLGERUNG> mit denen Sonderfälle berücksichtigt werden können, beispielweise <VOGEL(X), PINGUIN(X), ¬FLIEGT(X)>. Eine DL-KB besteht dann aus $\langle F, D\rangle$ mit F Fakten und Regeln der eigentlichen KB und D Default-Regeln.

3.6.3 Wann ist eine Menge E eine Erweiterung einer DL-KB?

Wenn sie alle möglichen Schlußfolgerungen aus KB und Default-Regeln beinhaltet, i.Z.

$$\Phi \in E \Leftrightarrow KB \cup \Delta \models \Phi \text{ mit}$$
$$\Delta = \{\gamma \mid \langle \alpha, \beta, \gamma\rangle \in D, \alpha \in E, \neg\beta \notin E\}$$

3.6.4 Weswegen schlug Reiter eine andere Definition vor?

Es gibt Default Logic Wissensbasen, mit denen die obige Definition *kontra-intuitive Erweiterungen* produziert: Sei $F = \emptyset$ und $D = \{\langle p, \mathrm{True}, p\rangle\}$, dann ist $\{p\}$ eine Erweiterung weil $p \in E$ und $\neg\mathrm{True} \notin E$. Es gibt allerdings keinen guten Grund p zu glauben.

3.6.5 Wie berechnet man die Erweiterungen einer DL-KB $\langle F, D\rangle$?

1. Bestimme zuerst die Menge M der konfligierenden Schlüsse (beispielsweise FLIES(TWEETY) und ¬FLIES(TWEETY)).
2. Prüfe für jede Teilmenge $\Delta \subseteq M$ ob $\{\varphi \mid F \cup \Delta \models \varphi\}\}$ eine gültige Erweiterung bezüglich D ist.

3.6.6 Was hat es mit „Autoepistemic Logic“ auf sich?

Autoepistmische Logik führt ein Believe-Prädikat **B** ein mit welchem über den *Glauben* des Agenten geschlossen werden kann. Beispielsweise kann ein Agent *wissen*, dass KB $\models \alpha \vee \beta$ ohne zu *glauben*, dass α oder β.

3.6.7 Wie ist die „Stable Expansion“ einer Satzmenge ε definiert?

Eine stabile Expansion einer Satzmenge E beinhaltet alle die Sätze α die konsistent als wahr angenommen werden können, d.h.

1. **Abgeschlossen unter Entailment**: $\forall\alpha.E \models \alpha \Rightarrow \alpha \in E$.
2. **Positive Introspektion**: $\alpha \in E \Rightarrow \mathbf{B}\alpha \in E$.
3. **Negative Introspektion**: $\neg\alpha \in E \Rightarrow \neg\mathbf{B}\alpha \in E$.

3.6.8 Wie berechnet die stabilen Extensionen einer AE-KB?

1. Ersetze jedes $\mathbf{B}\alpha$ durch True oder $\neg$True .
2. Vereinfache die resultierende KB zu KB^0.
3. Wurde $\mathbf{B}\alpha$ durch True ersetzt, bestätige, dass $\text{KB}^0 \models \alpha$ (bzw. umgekehrt: $\neg$True $\Rightarrow$ $\text{KB}^0 \not\models \alpha$).
4. Falls obiges für alle α passt, ist KB^0 stabile Erweiterung von .

3.6.9 Wie schwer sind solche Default Logiken?

Default Logiken sind noch schwieriger als FO und damit unentscheidbar.

3.7 Hierarchie und Vererbung

3.7.1 Welche Preemtion-Strategien haben wir kennengelernt?

Bei „Preemption“ handelt es sich um die Vorwegnahme, bzw. das Ausschließen von Folgerungen durch „abkürzen“ im Vererbungsnetz.

- **Shortest Path**.
- **Inferential Distance**: Ein Knoten a liegt näher an b als an c, wenn es einen Pfad von a nach c durch b gibt („topologische Distanz“).

3.7.2 Was ist eine „Credulous Extension“ eines Vererbungsnetzes?

Eine „Credulous Extension“ bezüglich einem Knoten a, ist eine *maximale*, *eindeutige* Teilhierarchie des Netzes, die *a-connected* ist, d.h. deren Knoten alle von a aus zu erreichen sind.

3.7.3 Was kennzeichnet eine „Preferred Extension“ eines Vererbungsnetzes?

Eine Erweiterung X wird vor einer Erweiterung Y bevorzugt, wenn Y *unzulässige* Kanten beinhaltet.

3.7.4 Auf welche Arten kann man in Vererbungsnetzen schließen?

- Credulous: Beliebige bevorzugte Erweiterung.
- Skeptical: Schlussfolgerungen die durch einen Pfad überstützt werden, der in jeder bevorzugten Erweiterung ist.
- Ideally Skeptical: Nur gemeinsame Schlußfolgerungen aller *bevorzugten* Erweiterungen (nicht pfadbasiert).

Eine Erweiterung heißt

- **Bevorzugt** („Preferred“):
- **Leichtgläubig** („credulous“): Wenn sie beliebig aus den möglichen Erweiterungen gewählt wurde.
- **Skeptisch**: Wenn sie nur die Schlußfolgerungen beinhaltet die allen Erweiterungen gemein sind.

3.7.5 Wie übersetzt man ein „Inheritance Network“ in „Autoepistemic Logic“?

- Für positive $(\alpha \rightarrow \beta)$ Kanten füge einen Satz $\forall x.\alpha\,(x) \wedge \mathbf{B}\beta\,(x) \supset \beta\,(x)$
- Für negative Kanten $(\alpha \not\rightarrow \beta)$ füge einen Satz $\forall x\ \alpha\,(x) \wedge \neg\mathbf{B}\beta\,(x) \supset \neg\beta\,(x)$

3.8 Situationskalkül

3.8.1 Was ist das Situationskalkül?

Das Situationskalkül ist ein Dialekt der FOL, der zur Beschreibung von dynamischen Welten verwendet wird. Es wird *bewiesen*, dass es einen Plan gibt, der zum Ziel führt und als *Seiteneffekt* wird dieser generiert. Hierzu kennt dieser Dialekt:

- **Situationen**: Folgen von Aktionen beginnend in einer Initialsituation.
- **Fluenten**: Veränderliche Eigenschaften des Weltmodells.
- **Aktionen**: Aestehend aus Name, Bedingungen, Effekten und Frames.
- Ein spezielles Prädikat $\mathbf{do}(\alpha,\mathbf{s})$, das Situationen durch Aktionen ineinander überführt.

3.8.2 Welche Fragen möchte man mit dem Situationskalkül beantworten?

1. Projection: Gegeben eine Folge von Aktionen: Gilt Φ nachdem diese ausgeführt wurden? $\mathrm{KB} \models \Phi\left(\mathrm{do}\left(\langle a_1, \ldots, a_n\rangle, S_0\right)\right)$.
2. Legality: Kann eine Folge von Aktionen ausgeführt werden? $\forall i.\ \mathrm{KB} \models \mathrm{poss}\left(a_i, \mathrm{do}\left(\langle a_1, \ldots, a_{i-1}\rangle, S_0\right)\right)$.

3.8.3 Was ist das Frame-Problem?

Das Frame Problem besteht darin, angeben zu müssen welche Aktionen welche Fluenten *nicht beeinflussen*. Die Anzahl der Frame-Axiome beträgt $2 \cdot F \cdot A$, wobei F die Anzahl der Fluenten und A die Anzahl der Aktionen ist. Es gilt zur Lösung des Problems eine kompaktere Darstellung zu finden damit man nicht alle Frame Axiome aufzählen muss.

3.8.4 Geben Sie eine einfache Lösung des Frame-Problems an!

Reiter definierte pro Fluent einen Satz, aus dem sich durch „Completeness Assumption“ und „Unique Names Assumption“ *alle Effekte und Frames bezüglich dieses Fluenten ableiten* lassen. Dieses „Successor State Axiom“ hat die Form $\mathrm{Poss}(a, s) \supset [\mathrm{Fluent}(\ldots) \equiv \Phi]$, wobei Φ alle Möglichkeiten beinhaltet das Fluent zu ändern.

3.8.5 Erklären Sie den Regressions-Operator!

Der Operator ersetzt *Fluenten* durch alle Möglichkeiten das Fluent zu ändern aus dessen „Successor State Axiom“, i.Z. für einen Fluenten F mit $\mathrm{Poss}(a, s) \supset [\mathrm{F}(\ldots) \equiv \Phi]$

$$R\left[F\left(t, s\right)\right] = \Phi_t^x$$

3.8.6 Erklären Sie Linear Regression Planning!

Das Verfahren wählt zuerst *nichtdeterministisch* eine Aktion α. Hiermit wird die *Regression des Ziels durch* α berechnet, wobei das Fluent durch Möglichkeiten dies zu ändern ersetzt wird, die Variablen unifiziert werden und die *preconditions* von α an den Satz verkettet werden. Schließlich erhält man einen FO-Satz über die Initialsituation S_0 und eine Folge von Aktionen die zum Ziel $G(s)$ führen. Existiert keine solche Sequenz muss das Verfahren nicht terminieren. Weiterhin ist der Suchraum zu groß, als dass das Verfahren praxistauglich wäre.

3.9 Abductive Reasoning

3.9.1 Was versteht man unter Abductive Reasoning?

Das *gehaltserweiternde* Schließen von Beobachtungen auf Ursachen, d.h. man schließt aus $\beta, \alpha \supset \beta$, dass α vorliegen muss, bzw. fragt, welche Erklärung Δ man benötigt, damit $\mathrm{KB} \cup \Delta \models \alpha$. Eine Erklärung die *hinreichend*, *konsistent* (bzgl. der KB), *möglichst einfach* und *nicht trivial* ist, heißt dann „abductive explanation“ von α bzgl. KB.

3.9.2 Wie sieht eine Abduktive KB aus?

Besteht aus Regeln der Form „Diseases $\wedge$ Conditions $\Rightarrow$ Symptoms“.

3.9.3 Was ist ein Primimplikat?

Eine kleinste Klausel die von der KB erfüllt wird, d.h. KB $\models c$ und $\forall c^* \subset c$ KB $\not\models c^*$.

3.9.4 Wie berechnet man die Menge möglicher Erklärungen für ein Literal p?

1. Primimplikate berechnen: KB solange *resolvieren bis konvergiert* und alle Klauseln δ mit p behalten.
2. p aus den Klauseln δ entfernen.
3. Negation $\neg\delta$ zurückgeben

3.9.5 Was ist an der Berechnung der Primimplikate problematisch?

Es kann exponentiell viele Primimplikate geben.

3.9.6 Wir hatten eine Anwendung bei der Fehlererkennung in Schaltkreisen. Wie funktioniert die Berechnung des minimalen Fehlerszenarios?

Man benötigt ein *Modell des Schaltkreises* bestehend aus Gattern, Ein- und Ausgabeverbindungen, Wahrheitstabellen und *Normalverhalten* so wie ein *Fehlermodell*. Gegeben Eingaben und Beobachtungen (Ausgaben) sucht man nun eine *minimale Konjunktion abnormaler Gatter* Δ, so dass diese die Beobachtungen erfüllt, d.h.

$$\text{KB} \cup \text{Eingabe} \cup \Delta \models \text{Ausgabe}$$

Problem dabei ist, dass zum einen die Diagnose minimal ist, obwohl *nicht nur minimal viele Bauteile defekt* sein müssen, und dass im Fehlermodell *nicht alle möglichen Fehler erfasst* sein könnten. Formell berechnet man alle Resolventen der Eingabeklauseln für eine Ausgabe λ, d.h. die Menge der Erklärungen $\{P \setminus \lambda \mid P \text{ Primimplikat und } \lambda \in P\}$.

3.9.7 Wie sieht es dabei mit der Komplexität aus?

Die Berechnung von Primimplikaten geschieht durch Resolution (Resolution ist vollständig für nicht tautologische Primimplikanten). Da Resolution verwendet wird, kann das Verfahren exponentiell werden.

4 Datenbanken

4.1 Grundlegendes

4.1.1 Woraus besteht ein DBMS?

Daten und Programme die darauf zugreifen.

4.1.2 Wofür benötigt man überhaupt Datenbanken im Gegensatz zu Dateien des Betriebssystems?

- Anfrageoptimierung
- Benutzersynchronisation
- Recovery-Maßnamen
- Integritätsbedingungen
- Sicherheitsaspekte

4.1.3 Welche Ziele verfolgt man beim Datenbankdesign?

- **Phyische Datenunabhängigkeit**: Digitale Repräsentation beeinflusst Datenmodell nicht.
- **Logische Datenunabhängigkeit**: Datenmodell beeinflusst digitale Repräsentation nicht.

4.1.4 Welche Schichten kennt ein Datenbankmodell?

1. **Phsysisch**: Datenrepräsentation auf dem Computer.
2. **Logisch**: Datenmodell (Objekte und Operatoren), Domänen, Schema/Struktur und Ausprägung/Inhalte.
3. **Sichten**: Eingeschränkter „Unterraum“ der logischen Schicht.

4.1.5 Welche Ebenen des Datenbankentwurfs gibt es?

1. **Physische Ebene**: Indexstrukturen, ...
2. **Implementationsebene**: Relatione Algebra, Relationenkalkül.
3. **Konzeptuelle Ebene**: ER-Modelle, Use-Cases.

4.2 Entity-Relationship-Modell

4.2.1 Was kann man mit dem Entity-Relationship-Modell machen?

ERM dient der graphischen *Darstellung der statischen Aspekte des Weltmodells*, das wir abbilden wollen. Hierbei werden *Entitäten* über *Relationen* miteinander verbunden und durch *Attribute* beschrieben. Den Relationen können *Kardinalitäten* zugeordnet werden. Neben den normalen Relationen gibt es *Vererbung* („is-a“), *Aggregation* („part-of“) und *Komposition*.

4.2.2 Wie überträgt man eine (n:m)-Beziehung in das relationale Modell?

Man definiert je eine Tabelle für die Entitäten und eine weitere für deren Beziehungen, so dass diese die Schlüsselpaare der zueinander in Relation stehenden Entitäten enthält.

4.2.3 Warum wollen wir Vererbung haben?

Damit *Redundanzen vermieden* werden, d.h. die Attribute einer Oberklasse nicht mehrfach gespeichert werden müssen.

4.2.4 Wie überträgt man Vererbung in das relationale Modell?

Lässt sich nicht direkt übertragen, sondern muss abhängig von der Art der Vererbung modelliert werden:

- **Totale Zerlegung**: Nur Relation für Subtypen.
- **Partielle Zerlegung**: Relation für Eltern, Relationen für Typen und Zeiger von Kind auf Elter.
- **Disjunkte Vererbung**: Einzelne Relation für alle Entitäten und Enum-ähnliches Attribut für Typen.
- **Überlappende Vererbung**: Einzelne Relation wie ober aber mit booleschem Vektor für Typzugehörigkeit.

Alternativ kann man Vererbung durch Sichten modellieren. Hierbei wird eine Sicht von den Subtypen auf die Supertypen erstellt.

4.3 Relationale Algebra

4.3.1 Welche Operatoren kennt die Relationale Algebra?

- **Selektion**: Wählt bestimmte Tupel aus ($\approx$ Zeilen).
- **Projektion**: Schränkt eine komplette Relation auf bestimmte Attribute ein ($\approx$ Spalten).

- **Natürlicher Join**: $(\approx \langle R - S \mid R \cap S \mid S - R \rangle)$.
- **Umbenennung**.
- und die übliche Mengenoperationen: $\cup, \cap, -, \times$.

4.3.2 Was ist eine „Funktionale Abhängigkeit“?

Eine funktionale Abhängigkeit ist eine (semantische) Kausalität zwischen zwei Atrtibutsmengen A und B. Wir bezeichnen A und B als funktional abhängig $A \to B \Leftrightarrow A$ bestimmt die Werte von B. Beispielsweise gilt für einen Superschlüssel α einer Relation R: $\alpha \to R$.

4.3.3 Was ist ein „Schlüssel“?

Ein *Superschlüssel* einer Relation ist eine Attributsmenge, die Tupel dieser Relation eindeutig determiniert. *Schlüsselkandidaten* sind kleinstmögliche solcher Attributsmengen und der *Primärschlüssel* einer Relationen wird aus diesen gewählt.

4.3.4 Wieso möchte man Datenbanken in Normalformen haben?

Willkürliche Relationen führen führen zu *Redundanzen* und *Anomalien*:

- **Einfügeanomalie**: Einfügen nur einer Entität macht Auffüllen mit NULL notwendig.
- **Löschanomalie**: Löschen eines Tupels führt zum Verlust der anderen Entität.
- **Updateanomalie**: Es werden nicht alle redundanten (speicherbedarf!) Vorkommen einer Entität zugleich geändert, was zu inkonsistenten Daten führt.

4.3.5 Wann ist eine Attributsmenge voll funktional von einer anderen abhängig?

Eine Attributsmenge B ist voll funktional von einer Attributsmenge A abhängig, wenn man kein Attribut aus A entfernen kann, ohne dass die Abhängigkeit verloren geht; also A ein Kandidatenschlüssel für B ist.

4.3.6 Was besagen die ersten vier Normalformen?

1. Jedes Attribut ist atomar, keine mengenwertigen Attribute.
2. Jedes Nichtschlüsselattribut A ist voll funktional (d.h. vom ganzen) von jedem Kandidatenschlüssel κ abhängig, i.Z. $\forall \kappa.\ \kappa \to A \in F^+$ und diese FD ist *linksreduziert*.
3. Kein Nichtschlüsselattribut darf *transitiv* von einer Menge anderer Nichtschlüsselattribute abhängig sein, i.Z. $\forall$ FD.$\alpha \to \beta$ über R gilt
   a) $B \in \alpha$, d.h. diese FD ist trivial,

b) B ist prim, d.h. in keinem Kandidatenschlüssel enthalten oder

c) α ist Superschlüssel von R.

d.h. die Nicht-Schlüssel-Attribute sind funktional unabhängig voneinander.

4. Es darf nicht mehrere - voneinander unabhängige - 1:N Beziehungen (auch: *mehrwertige Abhängigkeiten*) zu einem Schlüsselwert in einer Relation geben (vgl. Beispielrelation [Assistent, Sprache, Programmiersprache] mit Assistent $\overset{1:N}{\rightarrow}$ Sprache und Assistent $\overset{1:N}{\rightarrow}$ Programmiersprache).

4.3.7 Was besagt die BCNF?

$\forall \alpha \rightarrow \beta$: α Superschlüssel oder $\beta \subseteq \alpha$, d.h. *kein NKA bestimmt ein Schlüsselattribut* (beispielweise ist [Name, Sportart, Verein], mit Verein $\rightarrow$ Sportart, in 3NF aber nicht in BCNF.

4.3.8 Welche Ansprüche stellt man an Normalisierungsverfahren?

- **Verlustlostigkeit**: $R = R_1 \times R_2$.
- **Abhängigkeitserhaltung**: Jede funktionale Abhängigkeit $A \rightarrow B$ bleibt in *einer* Relation erhalten.

4.3.9 Wie verhalten sich die Normalformen bzgl Verlustlostigkeit und Abhängigkeitserhaltung?

Alle fünf Normalformen erlauben verlustlose Zerlegungen, aber nur die ersten drei Normalformen sind abhängigkeitserhaltend.

4.3.10 Was besagen die „Armstrong-Axiome“ ?

Die Armstrong-Axiome dienen dazu aus bestehenden funktionalen Abhängigkeiten einer Relation weitere Abhängigkeiten herzuleiten:

- **Reflexivität**: $B \subseteq A \Rightarrow A \rightarrow B$.
- **Verstärkung**: $A \rightarrow B \Rightarrow AC \rightarrow BC$.
- **Transitivität**: $A \rightarrow B, B \rightarrow C \Rightarrow A \rightarrow C$.

4.3.11 Wie verwendet man diese um weitere funktionale Abhängigkeiten zu bestimmen?

Man wendet die Axiome auf bestehende funktionale Abhängigkeiten F an um weitere funktionale Abhängigkeiten herzuleiten. Das AttrHülle-Verfahren sammelt so lange funktionale Abhängigkeiten $\beta \rightarrow \gamma$ bezüglich einer Eingabe

4.3.12 Was ist die Hülle funktionaler Abhängigkeiten?

Die Menge Attribute F^+ die von $\langle F, \alpha \rangle$ bezüglich α bestimmt wird.

4.3.13 Wie kann man prüfen ob eine Attributsmenge A ein Schlüssel der Relation R ist?

Man berechnet die Hülle F^+ funktionaler Abhängigkeiten von F bezüglich α, AttrHülle(F,α), und prüft danach ob $R = \alpha^+$.

4.3.14 Wann sind funktionale Abhängigkeiten äquivalent?

Genau dann, wenn deren Hüllen gleich sind.

4.3.15 Was ist die kanonische Überdeckung einer Menge funktionaler Abhängigkeiten?

Die kleinste Menge funktionaler Abhängigkeiten mit gleicher Hülle.

4.3.16 Wie sieht es mit der Mächtigkeit von SQL im Vergleich zum Relationenkalkül aus?

SQL ist eine *deklarative Anfragesprache* mit der spezifiziert werden kann, wie interessante Daten aussehen (nicht, wie diese zu zu beschaffen sind = prozedural). SQL ist damit, wie die relationale Algebra und das Relationenkalkül, grundsätzlich nicht *Turing-vollständig*. SQL kennt außerdem Aggregatfunktionen und neuere SQL-Versionen bieten die Möglichkeit Spezialfälle von Rekursionen darzustellen, welches beides mit dem Relationenkalkül nicht geht. D.h. SQL ist mindestens *relational vollständig*.

4.3.17 Sind die unterschiedlichen relationalen Anfragesprachen unterschiedlich mächtig?

Relationale Algebra, Tupel-Kalkül und Domänen-Kalkül sind - auf sichere Ausdrücke eingeschränkt - gleichmächtig und werden als *relational vollständig* bezeichnet.

4.3.18 Was sind sichere Ausdrücke?

- Ergebnisse von Anfragen im Tupel-Kalkül müssen *Teilmenge der Domäne* sein.
- Im Domänen-Kalkül müssen außerdem noch gelten, dass $\exists x\ P(x) \Rightarrow x \subseteq \mathrm{Dom}(P)$ und $\forall x\ P(x) \Leftrightarrow \forall d \in \mathrm{Dom}(P) : P(d)$.

4.4 Sichten

4.4.1 Was sind Sichten?

Sichten entsprechen „virtuellen“, dynamisch erzeugten, Relationen.

4.4.2 Wie erzeugt man eine Sicht?

Mittels CREATE VIEW, beispielsweise CREATE VIEW studenten AS (SELECT name FROM personen WHERE beruf = student)

4.4.3 Wozu dienen Sichten?

Mit Sichten lassen sich Daten für Benutzer (un-)zugänglich machen, die Schreibweise von Anfragen vereinfachen und Vererbung modellieren. Hierzu kann man entweder eine Sicht von den Subtypen auf die Supertypen erzeugen oder eine Sicht von den Supertypen auf die Subtypen.

4.4.4 Wie steht es um die Updatefähigkeit von Sichten?

Sichten sind Updatefähig, wenn

- **keine Aggregatfunktionen** (COUNT, SUM, ...) benutzt werden,
- der **Schlüssel der Basisrelation** enthalten ist und
- **nur eine Tabelle** verwendet wird.

4.4.5 Warum stellt man diese Kritierien auf und lässt nicht bei Bedarf überprüfen, ob die Sicht updatefähig ist?

Weil dieses Problem **unentscheidbar** ist.

4.5 Datenintegrität

4.5.1 Welche Arten semantischer Integritätsbedingungen kennen Sie?

1. **Referentielle**: Fremdschlüssel = NULL oder Fremdschlüssel = Primärschlüssel eines Datensatzes der referentierten Tabelle.
2. **Statische**: Durch zum Beispiel „CHECK note BETWEEN 0.7 AND 5.0“.
3. **Constraints**: In der Tabellendefinition durch „CONSTRAINT Name (Bedingung)“. Wird vor jeder Manipulation der Tabelle überprüft.
4. **Trigger**: „Methoden“ bestehend aus Name, Auslöser (ON UPDATE), einschränkender Bedingung (WHEN :OLD.VALUE = X) und prozeduralem Code.

4.5.2 Woraus besteht ein Trigger?

1. **Kopf**: Name und Parameterliste.
2. **Auslöser** mit **Ziel**: Beispielweise „BEFORE UPDATE ON autos“.
3. **Rumpf**: Prozedur.

4.6 Physische Datenorganisation

4.6.1 Welche Arten von Speichermedien unterscheidet man?

- Primäre ($\approx$ RAM).
- Sekundäre ($\approx$ Festplatte, RAID).
- Archive ($\approx$ Bandspeicher).

4.6.2 Wie werden Tupel auf der Festplatte gespeichert?

Tupel werden in Seiten gespeichert und durch Tupel-Identifikatoren (TID) aus $\langle$Seiten-Nr, Index$\rangle$ aufgefunden.

4.6.3 Welche Datenstrukturen gibt es bei Datenbanken?

- **ISAM**: Suche wie *Daumenindex* in $\log(n)$. Einfügen und Löschen können Index komplett nach rechts bzw. links verschieben.
- **B-Bäume**: Balanciert, jeder Knoten hat zwischen k und 2k Einträge, Suchen und Einfügen jeweils in $\log(n)$.
- **B^+-Bäume**: „Hohle“ Bäume mit Referenzschlüsseln in Knoten, Daten nur in Blättern.
- **Hashing**: Index durch Hashfunktion, Überlaufende Seiten, Range-Anfragen, offenes Hashing.

4.6.4 Was ist ISAM?

ISAM ist eine Indexstruktur aus den 60er Jahren und steht für „Indexed Sequential Access Method“ und bedeutet, dass eine *Menge von Hashes* die Daten indiziert. Da diese Menge deutlich kleiner sein kann als die eigentlichen Daten und deren Ordnung eine *binäre Suche* erlaubt kann sehr schnell auf die Daten zugegriffen werden. Das gleiche Konzept findet man beispielweise beim Telefonbuch: Hier bestehen die Indizes aus den Anfangsbuchstaben der Nachnahmen.

4.6.5 Was gibt es bei ISAM für Probleme?

Aufwendige (lineare) Indexverschiebung, wenn ein komplett neuer Knoten in den Datenseiten benötigt wird bzw. ein Knoten gelöscht wird.

4.6.6 Welche Vorteile bietet ISAM?

Schnelles (logarithmisches) Lesen, d.h. besonders für statische Daten geeignet.

4.6.7 Hashing ist ja sehr schnell: Wieso braucht man überhaupt noch etwas anderes?

Hashing ist nur für punktuelle Anfragen schnell. Range-Anfragen können von Hashes nicht gut bearbeitet werden: Man muss für jeden Wert die Hashfunktion erneut berechnen.

4.6.8 Wie geht das besser?

B^+-Bäume: Absteigen bis Blatt (logarithmisch) und dann einfach Blätter durchlaufen, da diese in sortierter Reihenfolge vorliegen und über Zeiger verknüpft sind.

4.6.9 Wie groß ist ein typischer Block?

512 Byte

4.6.10 Wie groß ist ein Knoten im Baum?

Ein Knoten entspricht einer Seite (mehrere nebeneinander liegende Blöcke), d.h. 4KB, 8KB, ...

4.7 Anfragebearbeitung

4.7.1 Welche Schritte durchläuft eine Anfrage?

1. Deklarative Anfrage → Scanner, Parser, Sichtenauflöser ⇒
2. Algebraischer Ausdruck → Anfrageoptimierer ⇒
3. Auswertungsplan → Codeerzeugung, Ausführung

4.7.2 Welche Möglichkeiten hat man, die Auswertung einer Anfrage zu beschleunigen?

Man möchte die Ausgabe der einzelnen Operatoren möglichst klein halten.

- **Logische Optimierung**: Äquivalenzumformungen der RA Ausdrücke = Anordnung der Operatoren verändern (beispielsweise besser „zuerst Cousinen heraussuchen, dann darunter die Amerikanerinnen“ als umgekehrt)
- **Physische Optimierung**: Optimierung der Implementierung (beispielweise Verwendung von ISAM für Archive)

4.7.3 In welche Reihenfolge sollte man Joins bringen?

Joins ordnet man am besten so an, dass *innerste Joins möglichst kleine Ergebnisse* liefern, die man zwischenspeichern muss.

4.8 Transaktionen

4.8.1 Was sind Transaktionen?

„Arbeitseinheiten“ einer Anwendung, d.h. *gebündelte Folgen* von Datenbankoperationen die entweder *ganz oder gar nicht* ausgeführt werden.

4.8.2 Welche grundlegenden Anforderungen stellt man an Transaktionen?

- **Recovery**
- **Synchronisation**
- **Konsistenz**: Jede Transaktion führt die DB wieder in eine konsistenten Zustand über.

4.8.3 Was hat es mit dem ACID-Paradigma auf sich?

- **Atomicity**: Transaktion als kleinste, nicht teilbare Einheit, „Ganz oder gar nicht“. (Recovery).
- **Consistency**: Nach Beendigung der Transaktion ist Datenbasis konsistent (Synchronisation).
- **Isolation**: Nebenläufige Transaktionen beeinflussen sich nicht (Synchronisation).
- **Durability**: Wirkungen abgeschlossener Transaktionen bleiben dauerhaft bestehen. (Recovery).

4.8.4 Welche Fehlerarten können bei Transaktionen entstehen?

- **Lokale Fehler**: Transkation bricht ab → Lokales Undo.
- **Fehler mit Hauptspeicherverlust**
  1. Abgebrochene Transaktionen rückgängig machen → Globales Undo.
  2. Abgeschlossene Transaktionen nachvollziehen → Globales Redo.
- **Fehler mit Hintergrundspeicherverlust**: Hoffentlich Backups gemacht...

4.9 Mehrbenutzersynchronisation

4.9.1 Welche Probleme können im Mehrbenutzerbetrieb entstehen?

Beim *Interleaving* von Transaktionen können folgende Probleme auftauchen:

- **Lost Update**: T_1 überschreibt Änderungen von T_2.
- **Dirty Read**: T_1 liest Daten von T_2 bevor T_2 zurückgesetzt wird.

- **Phantom Read**: T_1 erstellt (zu spät) Daten, die von T_2 nicht mehr berücksichtigt werden.

4.9.2 Wie ist die formale Definition der Serialisierbarkeit?

Zwei *Transaktionen* sind serialisierbar, gdw. es eine Historie gibt, die äquivalent zur Hintereinanderausführung der Transaktionen ist. Wobei zwei Historien äquivalent sind, wenn deren *Konfliktoperationen* in gleicher Reihenfolge ausgeführt werden.

4.9.3 Was ist das 2-Phasen-Sperrmodell?

2PL ist ein Modell zur Mehrbenutzersynchronisation, bei dem Transaktionen zwei Phasen - *Wachstumsphase* (Locks nehmen) und *Schrumpfungsphase* (Locks abgeben) - durchlaufen. Bedigungen an die Transaktionen:

- Jedes Objekt muss vor Benutzung gesperrt werden.
- Keine Transaktion fordert Sperren zweimal an.
- Lock nicht gekriegt → Warten.
- Kein Lock wird über die Schrumpfungsphase hinaus behalten.

Strenges 2PL gibt am Ende alle Locks auf einmal frei.

4.9.4 Welchen Nutzen bietet das 2-Phasen-Sperrmodell?

2PL *garantiert Serialisierbarkeit*, strenges 2PL schützt vor *kaskadierenden Rollbacks*.

4.9.5 Welche Möglichkeiten zur Deadlock-Vermeidung gibt es?

- **Time-Out**.
- **Wartegraph**: Zyklus = Deadlock.
- **Preclaim**: Alle Locks am Anfang nehmen.
- **Zeitstempel**: Wound-Wait oder Wait-Die.

4.9.6 Was bewirken „Wound/Wait“ und „Wait/Die“?

Sei T_1 älter als T_2:

Wound/Wait

- T_1 fordert Lock an, dass T_2 hat: Starte die jüngere Transaktion T_2 neu.
- T_2 fordert Lock an, dass T_1 hat: Warte.

Wait/Die

- T_1 fordert Lock an, dass T_2 hat: Warte.
- T_2 fordert Lock an, dass T_1 hat: T_2 startet sich selbst (mit altem Zeitstempel!) neu.

4.10 Objektorientierte Datenbanksysteme

4.10.1 Welche Nachteile haben RDBMS gegeüber OODBMS?

- Ein Anwendungsobjekt wird auf unterschiedliche Relationen *segmentiert* (beispielweise ein Polygon auf mehrere Relationen „Flächen“, „Geraden“, „Punkte“.
- Künstliche Schlüsselattribute.
- Keine Möglichkeit *Verhalten* zu speichern.
- **Impedance Mismatch**: OODBMS sind *mengenorientiert*, RDBMS sind *satzorientiert* (Zeiger auf Nachfolgerdatensatz, vgl. Linkedlist).
- *Kein Information-Hiding* modellierbar.

4.10.2 Was muss ein OODMBS zusätzlich haben?

- **Vererbung**: Einfach und mehrfach.
- **Objektidentität**: Durch eindeutige, systemweite ObjectID.
- **Einbettbarkeit** in die Zielsprache: Beispielweise über einen objekt-relationalen-Mapper.

4.10.3 Wie bildet man objektorientierte Beziehungen ab?

Klassen die eine Beziehung mit anderen Klassen eingehen, werden über eine RELATIONSHIP-Eigenschaft miteinander verbunden.

Beispielweise würde die 1:N-Beziehung 1 Assistent betreut N Studierende in der Klasse Assistent wie folgt abgebildet RELATIONSHIP SET<STUDIERENDE> BETREUT INVERSE STUDIERENDE::BETREUTVON und in der Klasse Studierende wie folgt RELATIONSHIP ASSISTENTEN BETREUTVON INVERSE ASSISTENTEN::BETREUT.

4.10.4 Wie wird auf Objekte zugegriffen?

Man benötigt einen *Iterator* über der Extension einer Objektmenge und kann mit *Pfadausdrücken* auf Objektmengen zugreifen, beispielsweise „SELECT p.Name FROM p IN RWTH.Informatik.Professoren“, wobei p Iterator über der Extension der Professoren ist.

4.10.5 Was gibt es neben RDBMS und OODBMS noch?

Sogenannte „objektrelationale Datenbanken“ mit

- BLOBs
- Mengenwertigen Attributen
- Geschachtelten Reklationen (Relationen als Attribute)
- Benutzerdefinierten Typen
- Objektidentität
- Pfadausdrücken
- Verberung

4.11 Deduktive Datenbanken

4.11.1 Woraus besteht ein deduktives Datenbanksystem?

- **Extensionale Datenbasis** (EDB): Entspricht Fakten einer KB bzw. Tabellen im relationalen Modell.
- **Herleitungsregeln**: Verfasst in Datalog und analog zu Regeln einer Prolog .KB
- **Intensionale Datenbasis** (IDB): $\{\varphi|\text{KB} \models \varphi\}$, entspricht Views im relationalen Modell.

4.11.2 Woraus bestehen Datalog-Programme?

Datalog-Programme bestehen aus Regeln völlig analog zu Prolog-Programmen, d.h. der Form p :- $q_1, \ldots, q_n$ die ebenfalls für Implikationen der Art $q_1, \ldots, q_n \supset q$ stehen.

4.11.3 Wie unterscheiden sich Datalog- und Prolog-Programme?

- Datalog kennt *keine Funktionen*
- In Datalog ist die Reihenfolge der Regeln dank *Breitensuche* unwichtig
- Die Regeln müssen *stratifiziert* sein, d.h. alle Variablen aus dem Rumpf müssen auch im Kopf vorkommen

4.11.4 Wie leitet man die Implikationen einer deduktiven Datenbank her?

Die einzelnen Relationen werden in Reihenfolge der *topologischen Sortierung*, d.h. nach den Abhängigkeiten der sie bestimmenden Regeln, *materialisiert* bis der kleinste Fixpunkt der Datenbank erreicht ist. Hierzu berechnet man die *Kreuzprodukte* aus den Rümpfen der Regeln (p :- $q_1, \ldots, q_n \Rightarrow p = q_1 \times \ldots \times q_n$) und die *Vereinigungen* über den einzelnen Regeln p.

4.11.5 Was ist das Problem an der „naiven Auswertung“?

In jedem Iterationsschritt werden die Relationen aus vorigen Schritten neu berechnet. Die „semi-naive“ Auswertung berechnet hingegen nur die neu hinzukommenden Tupel.

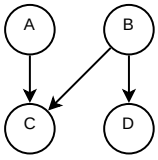
4.11.6 Wie lassen sich Selektion, Projektion, Join und Kreuzprodukt in Datalog realisieren?

- Selektion $\sigma_{X>n}(R)$: Q(X) :- R(X), X > n.
- Projektion $\pi_{A,B}(R)$: Q(A,B) :- R(A,B,C,...).
- Join $\pi_{A,B}(R \bowtie_{R.A_1=S.B_2} S)$: Q($A_2$,X,$B_1$,$B_3$) :- R(X,$A_2$), S($B_1$,X,$B_3$).
- Kreuzprodukt $R \times S$: Q(X,Y,Z) :- R(X), S(Y,Z).

5 Beispiele

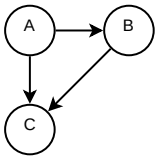
5.1 Künstliche Intelligenz

5.1.1 Berechnen Sie die Wahrscheinlichkeit $P(A, B, \neg D)$!



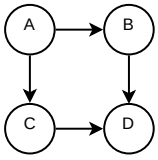
In diesem Bayesnetz gilt es zwei Fälle, nämlich C und $\neg C$ zu unterscheiden, so dass gilt: $P(A, B, \neg D) = P(\neg D|B) \cdot P(C|A, B) \cdot P(A) \cdot P(B) + P(\neg D|B) \cdot P(\neg C|A, B) \cdot P(A) \cdot P(B)$. Diese Fallunterscheidung heißt „Marginalisierung“ (vlg. 2.5.16).

5.1.2 Berechnen Sie die Wahrscheinlichkeit $P(A, \neg C)$!



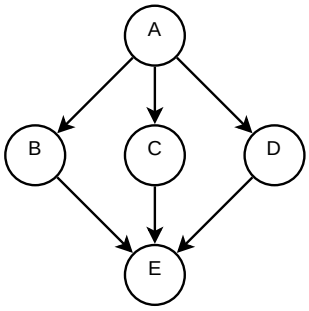
In diesem Bayesnetz gilt es zwei Fälle, nämlich B und $\neg B$ zu unterscheiden, so dass gilt: $P(A, \neg C) = P(\neg C|A, B) \cdot P(B|A) \cdot P(A) + P(\neg C|A, \neg B) \cdot P(\neg B|A) \cdot P(A)$. Wieder „Marginalisierung“(vlg. 2.5.16).

5.1.3 Berechnen Sie die Wahrscheinlichkeit $P(\neg A, B, \neg C, D)$!



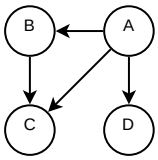
In diesem Bayesnetz gibt es hierfür genau einen Fall, nämlich $P(\neg A, B, \neg C, D) = P(\neg A) \cdot P(B|\neg A) \cdot P(\neg C|\neg A) \cdot P(D|B, \neg C)$

5.1.4 Berechnen Sie die Wahrscheinlichkeit $P(A, B, C, D)$!



In diesem Bayesnetz gibt es hierfür genau zwei Fälle, so daß $P(A, B, C, D) = P(A) \cdot P(B|A) \cdot P(C|A) \cdot P(D|A) \cdot P(E|B, C, D) + P(A) \cdot P(B|A) \cdot P(C|A) \cdot P(D|A) \cdot P(\neg E|B, C, D)$ gilt.

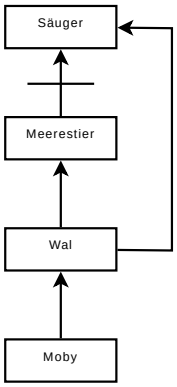
5.1.5 Berechnen Sie die Wahrscheinlichkeit $P(A, \neg B, \neg D)$!



In diesem Bayesnetz gibt es hierfür genau zwei Fälle, nämlich C und $\neg C$, so daß $P(A, \neg B, \neg D) = P(A) \cdot P(\neg B|A) \cdot P(C|A, \neg B) \cdot P(D|A) + P(A) \cdot P(\neg B|A) \cdot P(\neg C|A, \neg B) \cdot P(D|A)$ gilt.
Dies lässt sich durch *Variablenelimination* zu $P(A) \cdot P(\neg B|A) \cdot P(D|A)$ vereinfachen.

5.2 Wissensrepräsentation

5.2.1 Moby-Vererbungsnetz nach AL

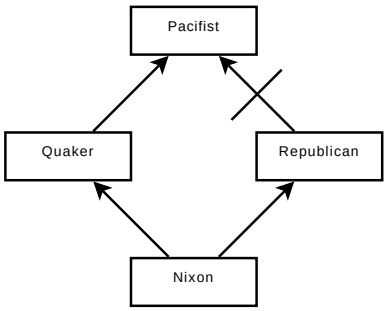


Das Mody-Vererbungsnetz entspricht der autoepistemischen Wissensbasis

- Wal(Moby),
- $\forall x\,(\text{Wal}\,(x) \supset \text{Meerestier}\,(x))$,

- $\forall x\,(\text{Wal}\,(x) \supset \text{Säuger}\,(x))$ und
- $\forall x\,(\text{Wal}\,(x) \wedge \neg\mathbf{B}\text{Säuger}\,(x) \supset \text{Meerestier}\,(x))$.

5.2.2 Nixon-Vererbungsnetz nach AL



Autoepistemische Wissensbasis

- Quaker(Nixon), Republican(Nixon),
- $\forall x\,(\text{Quaker}\,(x) \wedge \mathbf{B}\text{Pacifist}\,(x) \supset \text{Pacifist}\,(x))$,
- $\forall x\,(\text{Republican}\,(x) \wedge \neg\mathbf{B}\text{Pacifist}\,(x) \supset \neg\text{Pacifist}\,(x))$.

mit Erweiterungen

1. $E_0 = \{\text{Quaker}\,(\text{Nixon})\,, \text{Republican}\,(\text{Nixon})\,, \neg\text{Pacifist}\,(\text{Nixon})\}$
2. $E_1 = \{\text{Quaker}\,(\text{Nixon})\,, \text{Republican}\,(\text{Nixon})\,, \text{Pacifist}\,(\text{Nixon})\}$.

Möchte man hier eine Erweiterung ausschließen, muss man eine der Regeln anpassen:
$\forall x\,(\text{Quaker}\,(x) \wedge \mathbf{B}\text{Pacifist}\,(x) \supset \text{Pacifist}\,(x))$

Default-Logic Wissensbasis

- Quaker(Nixon), Republican(Nixon),
- $\langle \text{Quaker}\,(X)\,, \text{Pacifist}\,(X)\,, \text{Pacifist}\,(X) \rangle$,
- $\langle \text{Republican}\,(X)\,, \neg\text{Pacifist}\,(X)\,, \neg\text{Pacifist}\,(X) \rangle$,

mit Erweiterungen

1. $\text{Quaker}\,(\text{Nixon})\,, \text{Republican}\,(\text{Nixon})\,, \neg\text{Pacifist}\,(\text{Nixon})$
2. $\text{Quaker}\,(\text{Nixon})\,, \text{Republican}\,(\text{Nixon})\,, \text{Pacifist}\,(\text{Nixon})$.

Hier gegen hilft die nicht-normale Default-Regel:
$\langle \text{Republican}\,(X)\,, \neg\text{Republican}\,(X) \wedge \neg\text{Pacifist}\,(X)\,, \neg\text{Pacifist}\,(X) \rangle$.

5.3 Datenbanken

5.3.1 Relation die in 3. aber nicht in 4. Normalform ist.

Beispielsweise ist die Relation [Assistent, Fremdsprache, Programmiersprache] zwar in dritter Normalform, da

1. alle Attribute atomar sind (Erste Normalform),
2. jedes Nicht-Schlüsselattribut voll funktional vom ganzen Schlüssel abhängt (Zweite Normalform) und
3. kein Nichtschlüsselattribut darf *transitiv* von einer Menge anderer Nichtschlüsselattribute abhängig ist (Weder bestimmt Fremdsprache die Programmiersprache, noch umgekehrt) (Dritte Normalform).

Allerdings liegen hier zwei mehrwertige Abhängigkeiten, nämlich Assistent $\overset{1:N}{\rightarrow}$ Sprache und Assistent $\overset{1:N}{\rightarrow}$ Programmiersprache vor. Hierdurch ist die vierte Normalform (siehe 4.3.5) verletzt.

5.3.2 Bewertung von Relationsschemata

Sportvereine

Das Schema [Verein, Datum, Spiel, Trainer] mit den funktionalen Abhängigkeiten Verein → Trainer, Trainer → Verein ist zwar in erster Normalform, aber nicht in zweiter, weil Trainer nicht voll-funktional vom ganzen Schlüssel abhängig ist. Eine bessere Darstellung ist [Verein, Trainer], [Datum, Verein, Spiel]. Dies garantiert dann auch, das an jedem Datum nur ein Spiel stattfinden kann.

Auto

Gegeben das Schema [Auto, Ersatzteillieferant, Farbe, Hersteller] mit funktionalen Abhängigkeiten Auto → Farbe, Hersteller.
Das Schema erfüllt trivialerweise die erste Normalform, verletzt die zweite aber, da Farbe und Hersteller nicht voll funktional vom ganzen Schlüssel abhängen. Besser wäre [Auto, Farbe, Hersteller] und [Auto, Ersatzteillieferant].