Wissensrepräsentation und Problemlösung

Vorlesung an der Technischen Universität Chemnitz

Wintersemester 2004/2005

Prof. Dr. Werner Dilger

Inhalt

Literatur

J. Ferber: Multi-Agent Systems. An Introduction to Distributed Artificial Intelligence. Addison-Wesley, Harlow, 1999.

D. Pyle: Business Modeling and Data Mining. Morgan Kaufmann Publ., San Francisco, 2003.

M.M. Richter: Prinzipien der Künstlichen Intelligenz. B.G. Teubner, Stuttgart, 1992.

S.J. Russell, P. Norvig: Artificial Intelligence. A Modern Approach. Prentice Hall, Upper Saddle River, NJ, 1995.

1. Problemformulierung und Problemtypen

1.1. Problemlösende Agenten

Beispiel Rumänien

Herr *A* befindet sich in Arad, Rumänien, gegen Ende einer Ferienrundreise. Er hat ein Rückflugticket für den nächsten Tag ab Bukarest. Das Ticket ist nicht rückerstattbar, *A*'s Visum läuft in Kürze ab und nach dem morgigen Tag gibt es innerhalb der nächsten sechs Wochen keine freien Plätze für einen Rückflug. Das Performanzmaß von *A* wird noch durch andere Faktoren bestimmt, z.B. die Vertiefung seiner Sonnenbräune, die Verbesserung seiner rumänischen Sprachkenntnisse oder irgendwelche Besichtigungen. Dementsprechend viele Aktionen können gewählt werden. Angesichts des Ernstes der Lage sollte *A* jedoch das Ziel verfolgen, nach Bukarest zu fahren.

Die **Zielformulierung**, ausgehend von der aktuellen Situation, ist der erste Schritt beim Problemlösen. Ein Ziel wird als eine Menge von Weltzuständen betrachtet, in denen das Ziel erfüllt ist. Aktionen werden als Übergänge zwischen Weltzuständen betrachtet.

Bei der **Problemformulierung** wird entschieden welche Aktionen und welche Zustände betrachtet werden sollen. Sie folgt auf die Zielformulierung.

Hat ein Problemlöser mehrere mögliche Aktionen mit unbekanntem Ausgang, dann kann er eine Entscheidung dadurch herbeiführen, dass er verschiedene mögliche Aktions*folgen*, die zu Zuständen mit bekanntem Wert führen, untersucht und dann diejenige Aktionsfolge wählt, die zu dem Zustand mit dem besten Wert führt. Dieser Vorgang heißt **Suche**. Ein Suchalgorithmus nimmt eine Problembeschreibung als Eingabe und liefert eine **Lösung** in Form einer Aktionsfolge. Die Lösung kann dann in der **Ausführungsphase** ausgeführt werden. Damit ergibt sich als erster Ansatz zum Problemlösen der folgende Algorithmus:

```
function EINFACHES-PROBLEMLÖSUNGS-VERFAHREN(p) returns action
   inputs: p            ; eine Wahrnehmung
   static: s            ; eine Aktionsfolge, zu Beginn leer
           state        ; eine Beschreibung das aktuellen Weltzustands
           g            ; ein Ziel, zu Beginn leer
           problem      ; eine Problemformulierung

   state ← UPDATE-STATE(state, p)
   if s ist leer then
      g ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, g)
      s ← SEARCH(problem)
   action ← RECOMMENDATION(s, state)
   s ← REMAINDER(s, state)
   return action
```

1.2. Problemformulierung

Beispiel Staubsauge-Roboter

Die Staubsaugerwelt bestehe aus zwei Plätzen. An jedem Platz kann sich Schmutz befinden oder nicht und der Agent *Staubsauge-Roboter* (*SR*) befindet sich am einen oder am anderen Platz. In dieser Welt gibt es acht verschiedene Zustände, die graphisch in Abbildung 1.1 dargestellt sind, und *SR* kann drei verschiedene Aktionen ausführen: *Links*, *Rechts* und *Saugen*. Die Saugeaktion arbeite zuverlässig. Das Ziel ist, allen Schmutz aufzusaugen, d.h. das Ziel ist in einem der Zustände 7 und 8 von Abbildung 1.1 erfüllt.

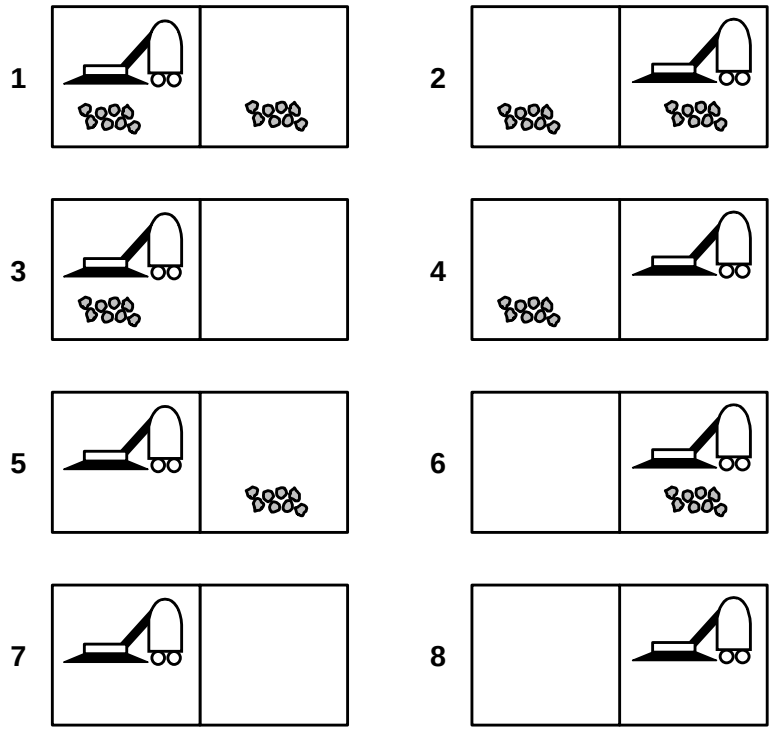

Abbildung 1.1

1.1.1. Problemtypen

Ein-Zustands-Problem

Der Problemlöser (d.h. hier: der Roboter) weiß, in welchem Zustand er sich befindet, d.h. die Umgebung ist zugänglich, und er weiß, was jede Aktion bewirkt. Dann kann er, ausgehend von seinem Zustand, für eine Aktionsfolge vorausberechnen, in welchem Zustand er nach Ausführung der Aktionen sein wird.

Mehr-Zustands-Problem

Der Problemlöser weiß nicht, in welchem Zustand er sich befindet, d.h. die Umgebung ist unzugänglich oder nur beschränkt zugänglich, aber er weiß, was jede Aktion bewirkt. Dann kann er trotzdem das Erreichen eines Zielzustands vorausberechnen. Dazu muss er aber über eine Menge von Zuständen schlussfolgern, nicht nur über einzelne Zustände.

Kontingenz-Problem

Der Problemlöser kann nicht im Voraus eine bestimmte Aktionsfolge berechnen, die zu einem Zielzustand führt, vielmehr benötigt er während der Ausführung der Aktionen Sensorinformationen um zu entscheiden, welche von mehreren möglichen Aktionen zum Ziel führt. Statt einer Folge von Aktionen braucht er also einen Baum von Aktionen, in dem jeder Zweig von der Wurzel zu einem Blatt eine Aktionsfolge darstellt, und jede Aktionsfolge behandelt ein kontingentes Ereignis.

Explorations-Problem

Der Problemlöser kennt die Auswirkungen seiner Aktionen nicht. In diesem Fall muss er experimentieren um herauszufinden, was seine Aktionen bewirken und welche Art von Zuständen es gibt. Dies kann er nur in der realen Welt tun. Er kann aber dabei „lernen“, d.h. sich eine interne Repräsentation der Welt aufbauen und diese für weitere Problemlösungen nutzen.

1.1.2. Wohldefinierte Probleme und Lösungen

Zur Definition eines Ein-Zustands-Problems sind folgende Informationen erforderlich:

- Ein Anfangszustand;
- eine Menge möglicher Aktionen;
- ein Zielprädikat;
- eine Pfadkostenfunktion.

Der Anfangszustand ist der Zustand, von dem der Problemlöser weiß, dass er sich in ihm befindet. Eine Aktion wird als **Operator** (alternativ: **Nachfolgefunktion**) bezeichnet in Verbindung mit dem Zustand, zu dem sie von einem gegebenen Zustand aus führt.

Der Anfangszustand und die Menge der Aktionen definieren den **Zustandsraum** des Problems. Er besteht aus der Menge aller Zustände, die vom Anfangszustand aus durch irgendwelche Aktionsfolgen erreichbar sind. Ein **Pfad** im Zustandsraum ist eine Aktionsfolge, die von einem Zustand zu einem anderen führt.

Das **Zielprädikat** kann der Problemlöser auf einen Zustand anwenden um zu testen, ob er ein Zielzustand ist.

Die **Pfadkostenfunktion** ordnet jedem Pfad im Zustandsraum einen Kostenwert zu. Dieser ist die Summe der Kosten jeder einzelnen Aktion entlang des Pfades. Die Pfadkostenfunktion wird häufig mit *g* bezeichnet.

Der **Datentyp** *Problem* ist definiert durch:

datatype PROBLEM
components: ANFANGSZUSTAND, OPERATOREN, ZIELPRÄDIKAT, PFADKOSTENFUNKTION

Instanzen dieses Datentyps bilden die Eingaben für Suchalgorithmen. Die Ausgabe eines Suchalgorithmus ist eine Lösung, d.h. ein Pfad vom Anfangszustand zu einem Zustand, der das Zielprädikat erfüllt.

Bei Mehr-Zustands-Problemen werden die Zustände durch Zustandsmengen ersetzt, eine Aktion wird durch eine Menge von Operatoren repräsentiert, die die Folgezustandsmenge spezifizieren. Ein Operator wird auf eine Zustandsmenge angewandt, indem er auf jeden einzelnen Zustand in der Menge angewandt wird und die Ergebnisse vereinigt werden. Anstelle des Zustandsraums erhält man einen **Zustandsmengenraum** und ein Pfad in diesem Raum führt über Zustandsmengen. Eine Lösung ist ein Pfad zu einer Zustandsmenge, die nur aus Zielzuständen besteht.

Die **Performanz** einer Problemlösung wird durch drei Größen gemessen:

- Wird eine Lösung gefunden?
- Wie hoch sind die Pfadkosten?
- Wie hoch sind die Suchkosten?

Die **Gesamtkosten** der Problemlösung sind die Summe aus Pfadkosten und Suchkosten.

1.1.3. Festlegung von Zuständen und Aktionen

Um Zustände und Aktionen für ein Problem zu definieren muss man immer von den realen Gegebenheiten abstrahieren. Im Folgenden wird anhand des Rumänien-Beispiels illustriert, wie Abstraktionen vorgenommen werden sollten. Abbildung 1.2 zeigt eine vereinfachte Landkarte mit den Straßenverbindungen zwischen verschiedenen Orten.

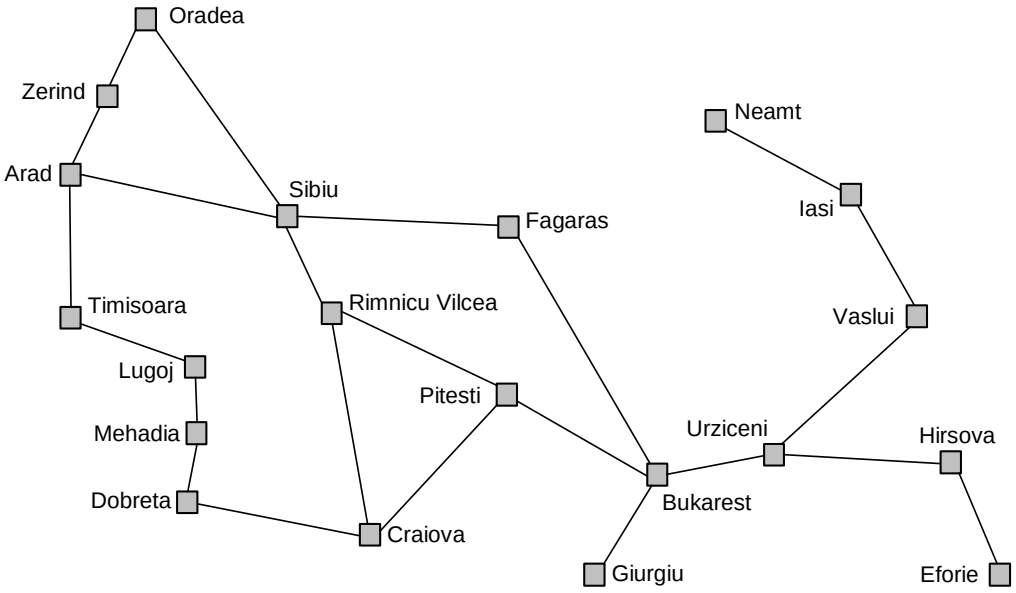

Abbildung 1.2

Das Problem heißt: „Fahre von Arad nach Bukarest unter Benutzung der Landkarte von Abbildung 1.2.“ Der Zustandsraum hat 20 Zustände, die durch die Befindlichkeit an den Orten gegeben sind. Der Anfangszustand ist „in Arad“, das Zielprädikat heißt: „Ist dies Bukarest?“ Die Operatoren entsprechen den Fahrten zwischen den Orten entlang den eingezeichneten Straßen. Als Pfadkosten wird die Anzahl der für eine Lösung erforderlichen Schritte genommen.

Das Weglassen von nicht erforderlichen Details aus einer Problembeschreibung heißt **Abstraktion**. Eine Abstraktion ist gut, wenn so viele Details wie möglich weggelassen werden, aber die Gültigkeit der Problembeschreibung erhalten bleibt und die abstrakten Aktionen einfach auszuführen sind.

1.3. Beispielprobleme

1.3.1. Spielzeugprobleme

8-Puzzle

Das 8-Puzzle ist das in Abbildung 1.3 dargestellte bekannte Spielzeug. Durch Verschieben der Plättchen soll, ausgehend von einer beliebigen Anordnung, eine bestimmte Ordnung der Zahlen hergestellt werden, z.B. die im Zielzustand von Abbildung 1.3 dargestellte.

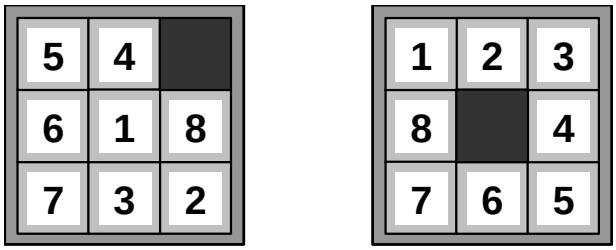

Abbildung 1.3

- **Zustände**: Beschreibung der Platzierung aller acht Plättchen sowie der Leerstelle in den neun Quadraten.
- **Operatoren**: Bewegung der Leerstelle nach links, rechts, oben oder unten.
- **Zielprädikat**: Der Zustand, der identisch ist mit dem in Abbildung 1.3 dargestellten Zielzustand.
- **Pfadkosten**: Jeder Schritt hat die Kosten 1, die Kosten eines Pfades sind damit gleich der Länge des Pfads.

8-Damen-Problem

8 Damen müssen so auf einem Schachbrett angeordnet werden, dass keine Dame eine andere angreift, d.h. dass keine horizontal, vertikal oder diagonal zu ihr steht. Abbildung 1.4 zeigt eine Anordnung der Damen, die die Bedingung verletzt.

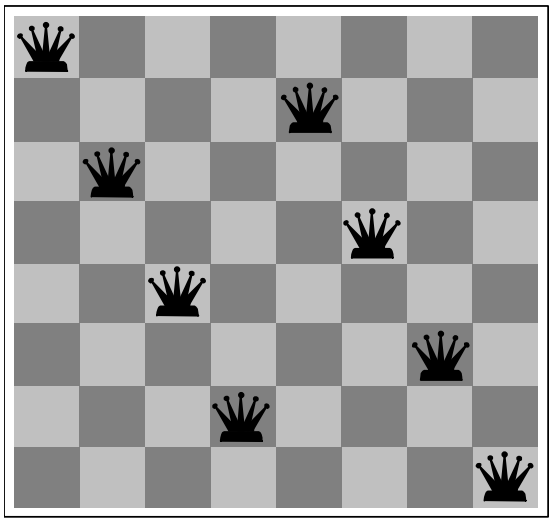

Abbildung 1.4

- **Zielprädikat**: 8 Damen auf dem Brett, keine angegriffen.
- **Pfadkosten**: Null.

Erste Formulierung für Zustände und Operatoren:

- **Zustände**: Beliebige Anordnung von 0 bis 8 Damen auf dem Brett.
- **Operatoren**: Setze eine Dame auf irgendein Feld.

Zweite Formulierung für Zustände und Operatoren:

- **Zustände**: Anordnungen von 0 bis 8 Damen auf dem Brett, bei denen keine Dame angegriffen ist.
- **Operatoren**: Setze eine Dame so in die am weitesten links liegende Spalte, dass sie von keiner anderen Dame angegriffen wird.

Zustände und Operatoren für eine Vollzustandsformulierung:

- **Zustände**: Anordnungen von 8 Damen, je eine in einer Spalte.
- **Operatoren**: Bewege eine angegriffene Dame auf ein anderes Feld in derselben Spalte.

Kryptoarithmetik

Ein kryptoarithmetisches Problem ist die Formulierung einer arithmetischen Aufgabe samt Lösung, in der die Ziffern durch Buchstaben ersetzt sind. Es ist eine Ersetzung der Buchstaben durch Ziffern gesucht, so dass die arithmetische Aufgabe korrekt gelöst ist. Ein Beispiel ist die folgende Aufgabe:

```
 FORTY      Lösung: 29785       F=2, O=9, R=7 usw.
+  TEN                850
+  TEN                850
 ----              -----
 SIXTY             31485
```

Eine einfache Problemformulierung:

- **Zustände**: Eine kryptoarithmetische Aufgabe mit teilweise durch Ziffern ersetzten Buchstaben.
- **Operatoren**: Ersetze alle Vorkommen eines Buchstabens mit einer Ziffer, die noch nicht in der Aufgabe vorkommt.
- **Zielprädikat**: Die Aufgabe enthält nur Ziffern und repräsentiert eine korrekte arithmetische Aufgabe.
- **Pfadkosten**: Null.

Staubsaugen

Zunächst wird das Ein-Zustands-Problem mit vollständiger Information und korrekt arbeitendem Staubsauger betrachtet. Die Problemformulierung ist:

- **Zustände**: Einer der acht Zustände von Abbildung 1.1.
- **Operatoren**: Nach links bewegen (L), nach rechts bewegen (R), Saugen (S).
- **Zielprädikat**: Kein Schmutz an beiden Plätzen.
- **Pfadkosten**: Jede Aktion hat die Kosten 1.

Abbildung 1.5 zeigt den vollständigen Zustandsraum mit allen möglichen Pfaden.

Nun wird das Mehr-Zustands-Problem betrachtet, bei dem der Roboter keinen Sensor hat. Die Problemformulierung ist:

- **Zustandsmengen**: Teilmengen der acht Zustände von Abbildung 1.1.
- **Operatoren**: Nach links bewegen (L), nach rechts bewegen (R), Saugen (S).
- **Zielprädikat**: Alle Zustände in der Zustandsmenge haben keinen Schmutz an beiden Plätzen.
- **Pfadkosten**: Jede Aktion hat die Kosten 1.

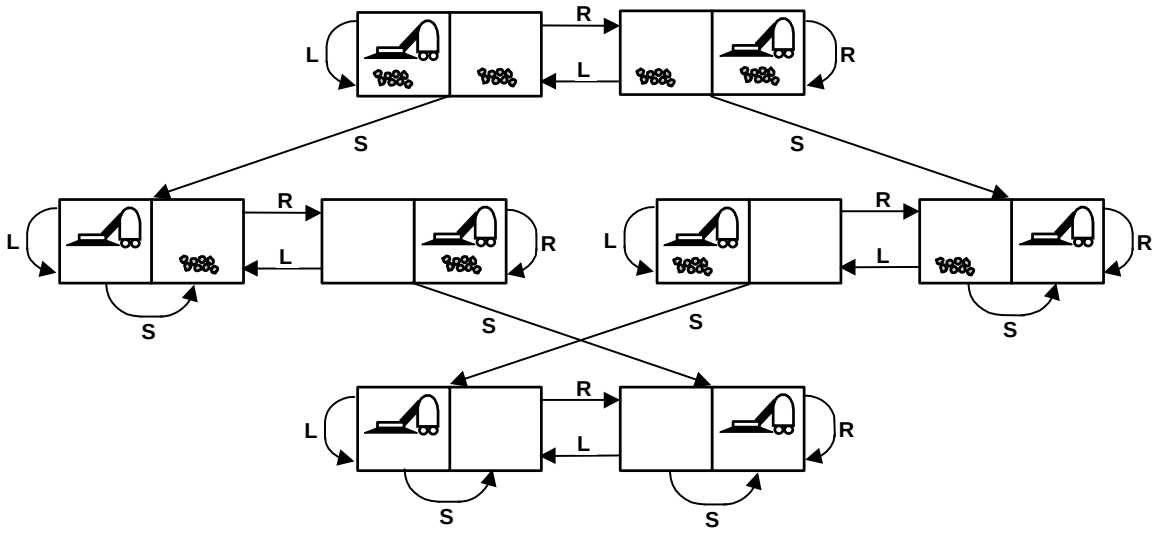

Abbildung 1.5

Der Startzustand ist die Menge aller Zustände, da der Roboter keinen Sensor hat um seinen Platz festzustellen. Abbildung 1.6 zeigt den Zustandsmengenraum für das Problem.

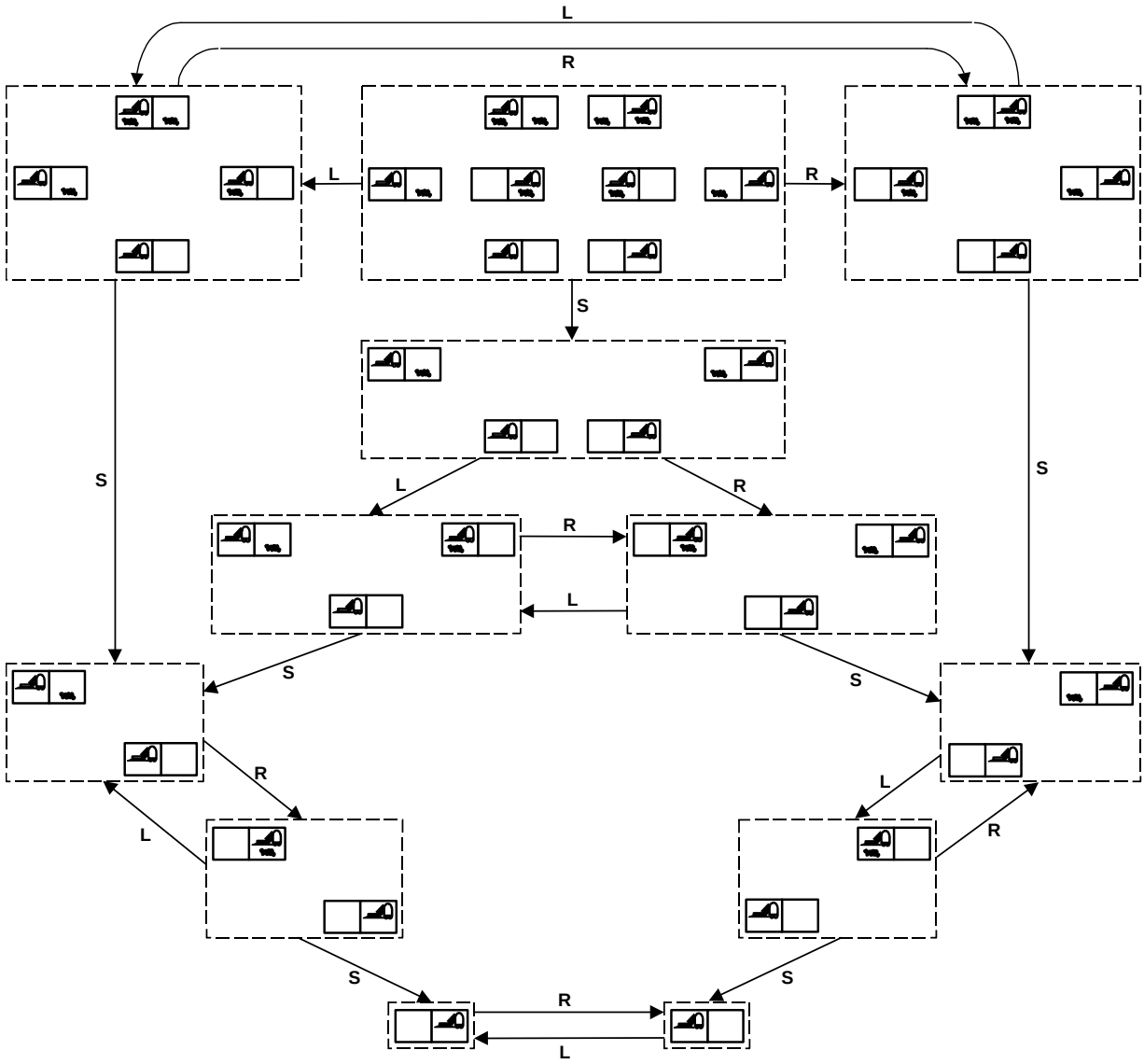

Abbildung 1.6

Missionars-Kannibalen-Problem

Drei Missionare und drei Kannibalen befinden sich auf einer Seite eines Flusses und wollen übersetzen. Es steht ein Boot zur Verfügung, das höchstens zwei Personen fasst. Gesucht ist eine Möglichkeit die Personen so über den Fluss zu bringen, dass zu keinem Zeitpunkt an einem der beiden Ufer mehr Kannibalen als Missionare sind (weil sonst die Missionare Gefahr laufen verspeist zu werden).

Problemformulierung:

- **Zustände**: Ein Zustand besteht aus drei Zahlen. Die erste repräsentiert die Anzahl der Missionare, die zweite die der Kannibalen und die dritte die der Boote am Startufer. Der Startzustand ist (3, 3, 1).
- **Operatoren**: Das Boot fährt mit einem oder zwei Missionaren oder mit einem oder zwei Kannibalen oder mit einem Missionar und einem Kannibalen zwischen den Ufern. Es gibt also fünf Operatoren. Bei einer Fahrt vom Startufer zum anderen Ufer verringern sich die Zahlen des aktuellen Zustands, in der umgekehrten Richtung vergrößern sie sich.
- **Zielprädikat**: Ist der aktuelle Zustand (0, 0, 0)?
- **Pfadkosten**: Anzahl der Flussüberquerungen.

1.3.2. Realweltprobleme

Routen finden

Die Aufgabe ist das Finden von Routen über vorgegebene Plätze und Übergänge zwischen den Plätzen, repräsentiert durch Kanten. Anwendungsbereiche: Routing in Computernetzen, Reiseberatungssysteme, Flugreisen-Planungssysteme.

Tourenplanung und das Travelling Salesman-Problem

Die Aufgabe ist das Finden einer Tour von einem Ort zu einem anderen (oder demselben), wobei bestimmte andere Orte besucht werden müssen. Zusätzlich zum Routen finden müssen hier die Zustände auch noch Informationen über die bereits besuchten Orte enthalten. Das bekannteste Tourenplanungsproblem ist das Travelling Salesman-Problem, bei dem jeder Ort genau einmal besucht werden muss. Gesucht ist die kürzeste Tour mit dieser Eigenschaft. Anwendungsbereiche: Logistik, Steuerung von Maschinen bei der Bearbeitung großer Flächen, z.B. beim Bohren.

VLSI-Layout

Die Aufgabe ist Lösungen für das Zellen-Layout und das Channel Routing zu finden. In den Zellen sind die elementaren Komponenten des Schaltkreises zu Gruppen zusammengefasst, von denen jede eine bestimmte Funktion realisiert. Jede Zelle hat eine bestimmte Größe und Form und benötigt eine bestimmte Anzahl von Verbindungen zu den anderen Zellen. Die Zellen müssen so auf dem Chip platziert werden, dass sie sich nicht überlappen und dass noch genug Raum zwischen ihnen für die Kanäle bleibt. Das Channel Routing bestimmt eine spezielle Route für jede Verbindung zwischen zwei Zellen.

Roboternavigation

Die Aufgabe ist einen Weg für einen Roboter in einem zweidimensionalen Raum zu bestimmen. Das Problem ist eine Verallgemeinerung des Routen-finden-Problems insofern, als hier ein kontinuierlicher Raum mit im Prinzip unendlich vielen möglichen Aktionen und Zuständen vorliegt.

Montageplanung

Die Aufgabe ist eine Reihenfolge zu finden, in der eine Menge von Teilen zu einem größeren Ganzen zusammengebaut werden können. Eine falsche Reihenfolge führt in eine Sackgasse und macht eine teilweise Demontage erforderlich. Es handelt sich bei diesem Problem um ein komplexes geometrisches Suchproblem.

1.4. Zusammenfassung

In diesem Kapitel ging es darum, was ein Problemlöser tun kann, wenn unklar ist, welche Aktion von mehreren möglichen im nächsten Schritt die beste ist. Der Problemlöser kann dann Aktionsfolgen betrachten, ihre Bestimmung nennt man **Suche**.

- Zunächst muss der Problemlöser ein Ziel formulieren und dieses dann zur Formulierung eines Problems benutzen.
- Eine Problembeschreibung besteht aus vier Teilen: dem **Anfangszustand**, einer Menge von **Operatoren**, einem **Zielprädikat** und einer **Pfadkostenfunktion**. Die Umgebung des Problems ist durch einen Zustandsraum spezifiziert. Ein Pfad durch den Zustandsraum vom Anfangszustand zu einem Zielzustand heißt Lösung.
- Man kann vier Problemtypen unterscheiden: Ein-Zustands-Probleme, Mehr-Zustandsprobleme, Kontingenz-Probleme und Explorations-Probleme.
- Die meisten realen Probleme sind schlecht definiert; durch eine gründliche Analyse können aber viele in das Zustandsraum-Modell abgebildet werden.

2. Problemlösen durch Suchen

2.1. Suche nach Lösungen

2.1.1. Erzeugung von Aktionsfolgen

Durch Anwendung eines Operators wird von einem gegebenen Zustand aus eine Menge anderer Zustände erzeugt. Dieser Prozess heißt **Expandieren** des Zustands. Das Wesen der Suche ist einen Zustand aus einer Menge auszuwählen und die anderen für einen eventuellen späteren Gebrauch zurückzustellen, nämlich dann, wenn die getroffene Auswahl nicht zum Erfolg führt. Die Wahl des als nächster zu expandierender Zustands wird durch eine **Suchstrategie** bestimmt. Der Suchprozess kann als Aufbau eines **Suchbaums**, der über den Zustandsraum gelegt wird, gedacht werden. Die Wurzel des Suchbaums ist der **Suchknoten**, der dem Anfangszustand entspricht. Die Blätter des Baums entsprechen Zuständen, die keine Nachfolger im Baum haben, entweder weil sie noch nicht expandiert wurden oder weil sei keine Nachfolger haben (bei der Expansion entstand die leere Menge). In jedem Schritt wählt der Suchalgorithmus einen Blattknoten zum Expandieren aus.

2.1.2. Datenstrukturen für Suchbäume und allgemeiner Suchalgorithmus

Ein Knoten *v* eines Suchbaums ist eine Datenstruktur mit fünf Komponenten:

- Der Zustand des Zustandsraums, dem *v* entspricht;
- derjenige Knoten im Suchbaum, der *v* erzeugt hat (der **Vaterknoten** von *v*);
- der zur Erzeugung von *v* verwendete Operator;
- die Zahl der Knoten auf dem Pfad von der Wurzel zu *v* (die **Tiefe** von *v*);
- die Pfadkosten des Pfades vom Anfangszustand bis zu dem *v* entsprechenden Zustand.

Der **Datentyp** *Knoten* ist definiert durch

datatype KNOTEN
components: ZUSTAND, VATERKNOTEN, OPERATOR, TIEFE, PFADKOSTEN

Es wird ferner eine Datenstruktur benötigt zur Repräsentation der Knoten, die zwar schon erzeugt, aber noch nicht expandiert worden sind. Diese Knotenmenge heißt **Rand**.

Die Knotenmenge *Rand* wird als **Liste** implementiert. Auf einer Liste sind folgende Operationen definiert:

- MAKE-LIST(*Elemente*) erzeugt eine Liste aus den eingegebenen Elementen
- EMPTY?(*Liste*) gibt *true* zurück, wenn keine Elemente mehr in der Liste sind
- REMOVE-FRONT(*Liste*) entfernt das erste Element der Liste und gibt es aus
- LISTING-FN(*Elemente*, *Liste*) fügt eine Menge von Elementen in die Liste ein

Bei der Funktion LISTING-FN kommt es darauf an, wie die Elemente in die Liste eingefügt werden. Daraus ergeben sich verschiedene Varianten des Suchalgorithmus. Der allgemeine **Suchalgorithmus** ist unter Verwendung dieser Funktionen in folgender Weise definiert:

function ALLGEMEINE-SUCHE(*Problem*, LISTING-FN) **returns** eine Lösung oder Fehler

nodes ← MAKE-LIST(MAKE-NODE(INITIAL-STATE[Problem]))
loop do
if EMPTY?(*nodes*) **then return** Fehler

```
    node ← REMOVE-FRONT(nodes)
    if STATE(node) erfüllt ZIELPRÄDIKAT[Problem] then return node
    nodes ← LISTING-FN(nodes, EXPAND(node, OPERATORS[Problem]))
end
```

2.2. Suchstrategien

Die Bestimmung der richtigen Suchstrategie erfolgt nach vier Kriterien:

- **Vollständigkeit**: Findet die Suchstrategie garantiert eine Lösung wenn es eine gibt?
- **Zeitbedarf**: Wieviel Zeit benötigt die Suchstrategie um eine Lösung zu finden?
- **Speicherplatzbedarf**: Wieviel Speicherplatz benötigt die Suchstrategie für die Suche?
- **Optimalität**: Findet die Suchstrategie die beste Lösung, wenn es mehrere Lösungen gibt?

2.3. Blinde Suchverfahren

2.3.1. Breitensuche

Bei der **Breitensuche** (Breadth-first search) wird zuerst der Wurzelknoten expandiert, dann alle seine Nachfolger, dann deren Nachfolger usw. Allgemein werden immer erst alle Knoten auf Tiefe d des Suchbaums expandiert bevor ein Knoten auf Tiefe d + 1 expandiert wird. Die Breitensuche wird durch Aufruf des Algorithmus ALLGEMEINE-SUCHE mit einer Listenfunktion, die die neuen Knoten an das Ende der Liste einfügt, implementiert.

```
function BREITENSUCHE(Problem) returns eine Lösung oder Fehler
    return ALLGEMEINE-SUCHE(Problem,ENLIST-AT-END)
```

Die Breitensuche erfüllt das Kriterium der Vollständigkeit, denn sie findet auf jeden Fall eine Lösung, falls eine existiert, und sie findet diejenige mit dem kürzesten Pfad. Sind die Pfadkosten eine monoton wachsende Funktion der Tiefe des Suchbaums, dann ist deshalb die Breitensuche sogar optimal.

Zur Abschätzung des Zeit- und Speicherplatzbedarfs wird ein hypothetischer Zustandsraum angenommen, in dem jeder Zustand zu genau b Folgezuständen expandiert werden kann. b heißt der **Verzweigungsfaktor** (branching factor) der Zustände. Die Anzahl der insgesamt erzeugten Knoten des Suchbaums ist ein Maß für den Zeit- und Speicherplatzbedarf. Auf Tiefe 0 gibt es einen Knoten (die Wurzel), auf Tiefe 1 b Knoten, auf Tiefe 2 b^2 Knoten usw., allgemein auf Tiefe d b^d Knoten. Liegt eine (die erste) Lösung auf Tiefe d, dann ist die Zahl der maximal zu erzeugenden Knoten

$$1 + b + b^2 + b^3 + \ldots + b^d = \frac{b^{d+1} - 1}{b - 1} = O(b^d)$$

Zeit- und Speicherplatzbedarf sind komplexitätsmäßig gleich groß ($O(b^d)$), denn im ungünstigsten Fall müssen alle Knoten der Tiefe d gespeichert werden, im günstigeren Fall immerhin die Knoten der Tiefe d-1.

2.3.2. Kostengesteuerte Breitensuche

Bei der kostengesteuerten Breitensuche wird immer der Knoten aus dem Rand mit den niedrigsten Pfadkosten als nächster expandiert. Unter bestimmten Voraussetzungen wird damit die kostengünstigste Lösung als erste gefunden.

Die kostengesteuerte Breitensuche findet die kostengünstigste Lösung, wenn sich die Kosten entlang eines Pfades an keiner Stelle verringern, d.h. wenn für alle Knoten *n* gilt

$$g(\text{NACHFOLGER}(n)) \geq g(n)$$

2.3.3. Tiefensuche

Die Tiefensuche expandiert immer einen Knoten auf der tiefsten Ebene des Baumes. Nur wenn keiner der Knoten auf der tiefsten Ebene expandierbar ist, geht sie zurück und expandiert einen Knoten auf einer niedrigeren Ebene. Diese Suchstrategie kann durch Aufruf des Algorithmus ALLGEMEINE-SUCHE mit einer Listenfunktion, die die neuen Knoten an den Anfang der Liste einfügt, implementiert werden.

function TIEFENSUCHE(*Problem*) **returns** eine Lösung oder Fehler
  return ALLGEMEINE-SUCHE(*Problem*, ENLIST-AT-FRONT)

Der Speicherplatzbedarf der Tiefensuche ist linear, denn beim Absteigen in die Tiefe muss immer nur ein Knoten zusammen mit seinen Geschwisterknoten im Speicher gehalten werden. Ist *m* die größte im Suchbaum vorkommende Tiefe (*m* kann größer sein als die Lösungstiefe *d*), dann ist der Speicherplatzbedarf $b \cdot m$.

Der Zeitbedarf der Tiefensuche ist $O(b^m)$, da im ungünstigsten Fall alle Knoten bis zur maximalen Tiefe *m* expandiert und besucht werden müssen. In der Praxis kann der Zeitbedarf niedriger sein, weil nur ein Teil des Zustandsraums durchsucht werden muss bis eine Lösung gefunden ist.

Der Nachteil der Tiefensuche gegenüber der Breitensuche ist, dass sie bei sehr tiefen oder gar unendlich tiefen Suchbäumen in einem falschen Zweig „hängenbleibt“ oder zumindest sehr viel Zeit verbraucht und deshalb keine Lösung findet, auch wenn eine existiert. Aus ähnlichem Grund ist bei der Tiefensuche auch nicht garantiert, dass sie eine optimale Lösung findet. Die Tiefensuche ist also weder vollständig noch optimal.

2.3.4. Tiefenbegrenzte Suche

Die tiefenbegrenzte Suche ist Tiefensuche mit einem Tiefenschnitt. Jeder Pfad im Suchbaum wird höchstens bis zu einer vorgegebenen Tiefe entwickelt. Die tiefenbegrenzte Suche kann mit einem speziellen Algorithmus oder durch Aufruf von ALLGEMEINE-SUCHE zusammen mit Operatoren, die die Tiefe kontrollieren, implementiert werden. Der Zeit- und Speicherplatzbedarf ist ähnlich wie bei der Tiefensuche; ist *l* der Tiefenschnitt, dann ist der Zeitbedarf $O(b^l)$ und der Speicherbedarf $O(b \cdot l)$.

2.3.5. Suche mit iterativem Vertiefen

Die Suche mit iterativem Vertiefen ist tiefenbeschränkte Suche mit variablem Tiefenschnitt. Dabei werden sukzessive wachsende Tiefenschnitte 0, 1, 2, ... festgelegt und für jeden Wert die tiefenbeschränkte Suche durchgeführt. Die Implementierung kann durch folgende Funktion geschehen:

function SUCHE-MIT-ITERATIVEM-VERTIEFEN(*Problem*) **returns** eine Lösungsfolge oder Fehler
  inputs: *Problem* ; eine Problembeschreibung

  for *depth* ← 0 **to** ∞ **do**
    if Tiefenbegrenzte-Suche(*Problem*, *depth*) erfolgreich **then return** ihr Ergebnis
  end
  return Fehler

Die Suche mit iterativem Vertiefen vereinigt die Vorteile der Breitensuche und der Tiefensuche, sie ist vollständig und optimal. Sie benötigt nur so viel Speicherplatz wie die Tiefensuche.

Der Zeitbedarf der Suche mit iterativem Vertiefen ist $O(b^d)$ und der Speicherplatzbedarf $O(b \cdot d)$. Die Suche ist vollständig und optimal. Damit ist die Suche mit iterativem Vertiefen das beste blinde Suchverfahren. Es ist geeignet für große Suchräume mit unbekannter Tiefe der Lösung.

2.3.6. Bidirektionale Suche

Bei der bidirektionalen Suche wird simultan vorwärts vom Anfangszustand aus und rückwärts vom Zielzustand aus gesucht bis sich die beiden Suchen in der Mitte treffen. Ist der Verzweigungsfaktor in beiden Richtungen gleich *b* und liegt die Lösung auf Tiefe *d*, dann kann sie in der Zeit $O(2b^{d/2}) = O(b^{d/2})$ bestimmt werden.

Die folgenden Voraussetzungen sollten gegeben sein, damit bidirektionale Suche sinnvoll angewandt werden kann:

- Es muss möglich sein, die Vorgänger eines Knotens zu erzeugen. Diese sind für die Rückwärtssuche die zu erzeugenden Nachfolger. Die Vorgänger eines Knotens *n* sind alle Knoten, die *n* als Nachfolger haben.
- Alle Operatoren müssen umkehrbar sein. Dann sind die Menge der Vorgänger und die Menge der Nachfolger identisch. Das Berechnen der Vorgänger kann aber schwierig sein.
- Wenn es mehrere Zielknoten gibt, dann muss die Rückwärtssuche im Sinne eines Mehr-Zustands-Problems durchführbar sein. Dies ist immer möglich, wenn die Zielzustände explizit gegeben sind. Gibt es nur eine Beschreibung des Zielknotens durch ein Zielprädikat, dann wird die Definition von Zuständen schwierig.
- Die Prüfung, ob ein neu generierter Knoten bereits im anderen Teil des Suchverfahrens vorkommt, muss effizient möglich sein.
- Es muss möglich sein zu entscheiden, welche Art der Suche in beiden Richtungen ablaufen soll.

2.3.7. Vergleich der Suchverfahren

In der folgenden Tabelle sind die sechs Suchverfahren nach ihrer Komplexität und den Kriterien *Vollständigkeit* und *Optimalität* aufgelistet.

Kriterium	Breitensuche	kostengesteuerte Breite	Tiefensuche	Tiefenbegrenzte Suche	Iteratives Vertiefen	Bidirektional
Zeit	b^d	b^d	b^m	b^l	b^d	$b^{d/2}$
Speicherplatz	b^d	b^d	bm	bl	bd	$b^{d/2}$
Optimal?	ja	ja	nein	nein	ja	ja
Vollständig?	ja	ja	nein	ja, falls $l \geq d$	ja	ja

2.4. Constraintpropagierung

Ein **Constraint Satisfaction Problem** (CSP) ist eine spezielle Art von Problem mit besonderen Struktureigenschaften über die allgemeinen Probleme hinaus. In einem CSP sind die Zustände durch die Werte einer Menge von **Variablen** definiert und das Zielprädikat ist durch eine Reihe von **Constraints** formuliert, die die Variablenwerte erfüllen müssen. Die Lösung eines Constraintproblems spezifiziert Werte der Variablen, die die Constraints erfüllen.

Es gibt verschiedene Typen von Constraints. Da sie praktisch Relationen zwischen Variablen darstellen, können sie nach ihrer *Stelligkeit* in ein-, zwei-, drei- usw. stellige Constraints unterteilt werden. Sie können *hart* oder *weich* sein.

Jede Variable V_i in einem CSP hat einen **Wertebereich** D_i, eine Menge möglicher Werte für die Variable. Der Wertebereich kann kontinuierlich oder diskret sein. Ein einstelliges Constraint spezifiziert die zulässige Teilmenge des Wertebereichs, ein zweistelliges Constraint spezifiziert die zulässige Teilmenge des kartesischen Produkts zweier Wertebereiche. In diskreten Wertebereichen können Constraints einfach durch Aufzählung der zulässigen Werte oder Wertetupel definiert werden.

Es gibt verschiedene Lösungsverfahren für CSPs. Generell muss man damit rechnen dass der Zeitbedarf exponentiell sein kann, da es bekannte NP-vollständige CSPs gibt. Am interessantesten ist die **Kantenkonsistenzprüfung**. Ein Zustand ist kantenkonsistent, wenn jede Variable einen Wert in ihrem Wertebereich hat, der alle auf diese Variable bezogenen Constraints erfüllt. Die Kantenkonsistenzprüfung erfolgt dadurch, dass man bei jeder Variablen diejenigen Werte aus dem Wertebereich entfernt, die eines der Constraints verletzen. Dabei kann es passieren, dass auch Werte anderer Variablen, die an demselben Constraint beteiligt sind, das Constraint nicht mehr erfüllen, auch sie werden dann weggelassen. Dieser Prozess pflanzt sich über alle Constraints fort, deshalb heißt er auch **Constraintpropagierung**.

2.5. Heuristische Suchverfahren

Will man Wissen über ein zu lösendes Problem zur Steuerung des Suchprozesses nutzen, dann kann man dies mit Hilfe der allgemeinen Suchprozedur tun. Mit ihr können verschiedene Suchstrategien dadurch verwirklicht werden, dass man neue Knoten in unterschiedlicher Weise in die Schlange einfügt, sofern man die Knoten immer an derselben Stelle aus der Schlange entfernt. Für das Einfügen benötigt man eine **Evaluierungsfunktion**, die die richtige Reihenfolge bestimmt. Werden die Knoten so geordnet, dass der mit der besten Bewertung zuerst expandiert wird, dann nennt man das Verfahren **Best-first-Suche**. Der folgende Algorithmus realisiert diese Suche.

```
function BEST-FIRST-SUCHE(Problem, EVAL-FN) returns eine Lösungsfolge oder Fehler
   inputs: Problem          ; eine Problembeschreibung
           EVAL-FN          ; eine Evaluierungsfunktion

   LISTING-FN ← eine Funktion, die Knoten mittels EVAL-FN ordnet
   return ALLGEMEINE-SUCHE(Problem, LISTING-FN)
```

Es gibt eine ganze Familie von Best-first-Suchverfahren. Sie benutzen alle ein geschätztes Kostenmaß für die Lösung und versuchen, dieses zu minimieren. Dieses Kostenmaß muss die Kosten eines Pfads von einem Zustand zu dem nächstgelegenen Zielzustand erfassen.

2.5.1. Greedy-Suche

Eine einfache Best-first-Suche ist die Minimierung der geschätzten Kosten für das Erreichen des Ziels. Dazu wird derjenige Knoten, dessen Zustand als dem Zielzustand am nächsten liegend eingeschätzt wird, als erster expandiert. Eine Funktion, die die Kosten schätzt, nennt man **heuristische Funktion**, bezeichnet mit *h*. $h(n)$ sind also die geschätzten Kosten des billigsten Pfads vom Knoten n zu einem Zielzustand. Best-first-Suche, die eine heuristische Funktion h verwendet, **heißt Greedy-Suche**. Der folgende Algorithmus ist eine Implementierung der Greedy-Suche.

function GREEDY-SUCHE(*Problem*) **returns** eine Lösung oder Fehler
return BEST-FIRST-SUCHE(*Problem, h*)

Jede beliebige Funktion kann als heuristische Funktion verwendet werden. Es muss nur $h(n) = 0$ gelten, falls n ein Zielknoten ist.

Greedy-Suche geht ähnlich vor wie Tiefensuche, insofern als sie einen direkten Weg zur Lösung bevorzugt und rücksetzen muss, wenn sie in eine Sackgasse gerät. Sie hat die gleichen Nachteile wie die Tiefensuche, sie ist nicht vollständig und nicht optimal. Der worst case-Zeitbedarf ist $O(b^m)$, wobei m die maximale Tiefe des Suchraums ist. Es werden alle Knoten im Speicher gehalten, deshalb ist auch der worst case-Speicherplatzbedarf $O(b^m)$.

2.5.2. A*-Suche

Durch Addition der heuristischen Funktion h und der Pfadkostenfunktion g erhält man die Funktion

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

die die geschätzten Kosten der billigsten Lösung durch den Knoten n wiedergibt.

Hat die Funktion h die Eigenschaft, dass sie die Kosten eines Pfades bis zu einem Zielknoten nicht überschätzt, dann heißt sie eine **zulässige Heuristik**. Formal definiert: Sind h' die tatsächlichen (aber nicht bekannten) Kosten, dann muss für h gelten: Für alle Knoten n: $h(n) \leq h'(n)$. Wenn h zulässig ist, dann überschätzt auch die Funktion f niemals die Kosten der besten Lösung durch n. Best-first-Suche unter Verwendung von f als Evaluierungsfunktion mit zulässigem h heißt **A*-Suche**. Der folgende Algorithmus implementiert die A*-Suche.

function A*-SUCHE(*Problem*) **returns** eine Lösung oder Fehler
return BEST-FIRST-SUCHE(*Problem, g + h*)

Eine heuristische Funktion, deren Werte entlang der Pfade im Suchbaum von der Wurzel zu einem Blatt nie kleiner werden, nennt man **monoton**. Ist eine heuristische Funktion nicht monoton, dann kann sie in eine monotone Funktion umgeformt werden. Dies geschieht auf folgende Weise. Ist n ein Knoten und n' einer seiner Nachfolger. Dann setze

$$f(n') = \max(f(n), g(n') + h(n'))$$

Diese Gleichung heißt **Pfadmaximierungsgleichung**. Durch sie wird jede heuristische Funktion monoton.

2.6. Heuristische Funktionen

2.6.1. Heuristische Funktionen für das 8-Puzzle

- h_1 = Anzahl der falsch platzierten Plättchen. In Abbildung 1.3 sind 7 Plättchen an der falschen Position, also ist hier $h_1 = 7$. h_1 ist zulässig, denn jedes falsch platzierte Plättchen muss mindestens um eine Position verschoben werden.

- h_2 = Summe der Entfernungen der Plättchen von ihren Zielpositionen. Die Entfernungen werden in horizontaler und vertikaler Richtung gemessen, wie die Plättchen bewegt werden können. h_2 gibt also an, wieviele Züge mindestens notwendig sind, um ein falsch platziertes Plättchen in die

richtige Position zu bringen. Damit ist h_2 ebenfalls zulässig. Die Heuristik wird auch *Stadtblock-Distanz* oder *Manhattan-Distanz* genannt. Im Beispiel von Abbildung 1.3 ist

$$h_2 = 2 + 3 + 3 + 2 + 4 + 2 + 0 + 2 = 18$$

2.6.2. Konstruktion heuristischer Funktionen

Sei eine Problembeschreibung gegeben. Durch Abschwächung der Restriktionen für die Operatoren der Problembeschreibung ist es oft möglich, ein **relaxiertes Problem** zu bekommen, dessen exakte Lösung eine gute Heuristik für das Ausgangsproblem darstellt. Liegt die Problembeschreibung in formaler Sprache vor, dann können relaxierte Probleme automatisch konstruiert werden. Für das 8-Puzzle-Problem ist folgende Konstruktion möglich (in quasi-formaler Notation): Die Operatoren haben die Form

Daraus können drei Formen von relaxierten Problemen durch Weglassen einer oder mehrerer Bedingungen erzeugt werden:

(a) Ein Plättchen kann vom Quadrat A zum Quadrat B bewegt werden, wenn A und B benachbart sind.
(b) Ein Plättchen kann vom Quadrat A zum Quadrat B bewegt werden, wenn B leer ist.
(c) Ein Plättchen kann vom Quadrat A zum Quadrat B bewegt werden.

Liegt eine Menge zulässiger Heuristiken $h_1, ..., h_m$ für ein Problem vor, die sich nicht gegenseitig dominieren, dann kann man daraus die bestmögliche Heuristik h auf folgende einfache Weise konstruieren:

$$h(n) = \max(h_1(n), ..., h_m(n))$$

$h(n)$ wählt für den aktuellen Knoten n die momentan günstigste Heuristik aus. Es gilt: $h(n) \geq h_i(n)$ für alle $i = 1, ..., m$, d.h. h dominiert alle anderen Heuristiken.

Eine weitere Möglichkeit zur Konstruktion guter Heuristiken ist die Nutzung **statistischer Information**. Dazu lässt man die Suche über eine Menge von Trainingsproblemen laufen und wertet die Ergebnisse statistisch aus.

Eine dritte Möglichkeit zur Konstruktion guter Heuristiken ist die Bestimmung von **Merkmalen** eines Zustands, die bei der Berechnung der heuristischen Funktion eine Rolle spielen können, selbst wenn man nicht genau sagen kann, welche Rolle sie spielen. Mit Hilfe eines Lernalgorithmus ist es dann möglich, diese Merkmale bezüglich ihres Einflusses auf die heuristische Funktion zu gewichten.

Generell muss bei der Verwendung heuristischer Funktionen beachtet werden, dass ihre Verwendung nicht zu viel Rechenzeit benötigen darf, weil sonst der Einsparungseffekt wieder verloren geht.

2.6.3. Heuristische Funktionen für die Constraintpropagierung

Die Heuristik, bei der immer zuerst die am stärksten eingeschränkte Variable ausgewählt wird, heißt **Heuristik der am stärksten eingeschränkten Variablen**. An jedem Punkt der Problemlösung wird die Variable mit der geringsten möglichen Zahl von Werten ausgewählt und ihr wird ein neuer Wert (eine neue Wertemenge) zugewiesen. Auf diese Weise wird der Verzweigungsfaktor bei der Suche tendenziell verkleinert.

Die **Heuristik der am stärksten einschränkenden Variablen** wählt diejenige Variable als nächste aus, die an der größten Anzahl von Constraints mit noch nicht behandelten anderen Variablen, d.h. solchen, denen noch kein Wert zugewiesen wurde, beteiligt ist und ordnet ihr einen Wert zu. Auch auf diese Weise kann der Verzweigungsfaktor eingeschränkt werden.

Bei der Wahl eines Wertes für eine Variable ist die **Heuristik der geringsten Einschränkung** günstig. Nach ihr wird derjenige Wert ausgewählt, der möglichst wenige Werte bei den anderen Variablen, mit denen die aktuelle Variable über irgendwelche Constraints verknüpft ist, ausschließt.

2.7. Zusammenfassung

In diesem Kapitel ging es darum, was ein Problemlöser tun kann, wenn unklar ist, welche Aktion von mehreren möglichen im nächsten Schritt die beste ist. Der Problemlöser kann dann Aktionsfolgen betrachten, ihre Bestimmung nennt man **Suche**.

- Ein einziger **allgemeiner Suchalgorithmus** genügt zur Lösung aller Probleme; spezielle Varianten dieses Algorithmus verwenden verschiedene Suchstrategien.
- Suchalgorithmen werden nach den Kriterien **Vollständigkeit**, **Optimalität**, **Zeitbedarf** und **Speicherplatzbedarf** beurteilt. Die Komplexität hängt vom Verzweigungsfaktor b und der Tiefe d der Lösung auf niedrigster Tiefe ab.
- Die **Breitensuche** expandiert den am weitesten oben im Suchbaum stehenden Knoten zuerst. Sie ist vollständig, für Operatoren mit einfachen Kosten optimal und hat den Zeit- und Speicherplatzbedarf $O(b^d)$. Wegen des hohen Speicherplatzbedarfs ist sie in der Praxis kaum verwendbar.
- Die **kostengesteuerte Breitensuche** expandiert den Blattknoten mit den niedrigsten Kosten zuerst. Sie ist vollständig und optimal und hat denselben Zeit- und Speicherplatzbedarf wie die Breitensuche.
- Die **Tiefensuche** expandiert den am tiefsten gelegenen Knoten im Suchbaum zuerst. Sie ist weder vollständig noch optimal und hat den Zeitbedarf $O(b^m)$ und den Speicherplatzbedarf $O(bm)$, wobei m die maximale Tiefe ist. Bei Suchbäumen großer oder unendlicher Tiefe ist sie wegen des Zeitbedarfs nicht praktisch einsetzbar.
- Die **tiefenbegrenzte Suche** setzt einen Tiefenschnitt, bis zu dem die Suche höchstens gehen darf. Wenn der Tiefenschnitt auf der Tiefe des am weitesten oben liegenden Zielknoten liegt, dann sind der Zeit- und Speicherplatzbedarf minimal.
- Die **Suche mit iterativem Vertiefen** führt wiederholte tiefenbegrenzte Suche mit stetig wachsendem Tiefenschnitt durch bis eine Lösung gefunden ist. Sie ist vollständig und optimal und hat den Zeitbedarf $O(b^d)$ und den Speicherplatzbedarf $O(bd)$.
- Die **bidirektionale Suche** kann u.U. den Zeitbedarf stark reduzieren, ist aber nur unter bestimmten Voraussetzungen anwendbar. Der Speicherplatzbedarf kann sehr groß werden.
- **Best-first-Suche** ist eine allgemeine Suchmethode, bei der immer die Knoten mit den geringsten Kosten zuerst expandiert werden.
- Bei der **Greedy-Suche** werden die geschätzten Kosten $h(n)$ bis zu einem Zielknoten minimiert. Im Allgemeinen vermindert sich die Zeit gegenüber der blinden Suche, aber das Verfahren ist weder vollständig noch optimal.
- Verfeinert man die Abschätzung, indem man statt $h(n)$ den Wert $f(n) = g(n) + h(n)$ minimiert, dann erhält man aus der Greedy-Suche **A*-Suche**.

- A*-Suche ist vollständig, optimal und optimal effizient. Der hohe Speicherplatzaufwand beschränkt allerdings ihre Einsatzmöglichkeiten.
- Der Zeitaufwand bei heuristischer Suche hängt von der Qualität der heuristischen Funktionen ab. Gute Heuristiken können manchmal durch genaue Betrachtung der Problembeschreibung oder durch Verallgemeinerungen von Erfahrungen mit Problemklassen gewonnen werden.

3. Problemlösen durch Optimieren

3.1. Bergsteigen

Der folgende Algorithmus realisiert die Suche durch Bergsteigen.

```
function BERGSTEIGEN(Problem) returns einen Lösungszustand
  inputs: Problem              ; eine Problembeschreibung
  local variables: node        ; der aktuelle Knoten
                   next        ; ein Knoten

  node ← MAKE-NODE(INITIAL-STATE[Problem])
  loop do
    next ← ein Nachfolger von node mit höchstem Wert
    if VALUE[next] < VALUE[node] then return node
    node ← next
  end
```

Gibt es bei der Suche mehrere Nachfolgerknoten mit dem besten Wert, dann muss einer zufällig ausgewählt werden. Dabei kann eines der drei folgenden Probleme auftreten:

- **Lokales Maximum**: Ein lokales Maximum ist eine Spitze in der Landschaft, d.h. ein Punkt, dessen Nachbarn alle einen schlechteren Wert haben, aber er ist nicht das globale Maximum. Stößt die Suche auf ein lokales Maximum, dann bleibt sie dort hängen und findet kein globales Maximum.
- **Plateau**: Ein Plateau ist ein flacher Bereich der Landschaft, d.h. die Nachbarn eines Punktes in einem Plateau haben alle denselben Wert wie der Punkt selbst. Die Suche kann nur in eine zufällig gewählte Richtung fortgesetzt werden.
- **Grat**: Ein Grat ist ein Bereich mit steilen Flanken, aber entlang des Grats kann das Gefälle gering sein. Man benötigt Operatoren, die dafür sorgen, dass die Suche entlang des Grates verläuft und nicht nutzlos zwischen den Flanken hin und her pendelt.

Um diesen Problemen aus dem Weg zu gehen, kann man die Suche durch Bergsteigen mehrfach von verschiedenen zufällig gewählten Startpunkten aus durchführen und den besten erzielten Wert nehmen oder die Suche so oft durchführen, bis nach einer gewissen Anzahl von Wiederholungen kein besserer Wert mehr gefunden wird.

3.2. Simuliertes Ausglühen

Die Idee des simulierten Ausglühens ist, während der Suche in zufälligen Abständen Knoten mit schlechteren Werten auszuwählen um auf diese Weise den lokalen Maxima zu entgehen. Der folgende Algorithmus realisiert das simulierte Ausglühen.

```
function SIMULIERTES-AUSGLÜHEN(Problem, Zeitplan) returns einen Lösungszustand
  inputs: Problem              ; eine Problembeschreibung
          Zeitplan             ; eine Abbildung der Zeit auf „Temperatur“
  local variables: node        ; der aktuelle Knoten
                   next        ; ein Knoten
                   T           ; eine „Temperatur“, die die Wahrscheinlichkeit von
```

Abwärtsschritten steuert

node ← MAKE-NODE(INITIAL-STATE[*Problem*])
for $t \leftarrow 1$ **to** ∞ **do**
  $T \leftarrow$ *Zeitplan*[t]
  if $T = 0$ **then return** *node*
  next ← ein zufällig gewählter Nachfolger von *node*
  $\Delta E \leftarrow$ VALUE[*next*] − VALUE[*node*]
  if $\Delta E > 0$ **then** *node* ← *next*
    else *node* ← *next* nur mit Wahrscheinlichkeit $e^{\Delta E/T}$
end

3.3. Genetische Algorithmen

Genetische Algorithmen laufen auf Populationen von *Genomen* ab. Ein Genom ist eine Folge von *Genen*. Die Gene sind die elementaren Informationseinheiten genetischer Algorithmen, im einfachsten Fall sind sie (kurze) Bitstrings. Die genetischen Algorithmen erzeugen in einer großen Schleife immer neue Generationen von Genomen durch Anwendung der Operationen *Selektion*, *Kreuzung* und *Mutation*. Dabei werden folgende Schritte durchgeführt:

1. Definiere die Genome und eine Fitnessfunktion und erzeuge eine initiale Population von Genomen.
2. Modifiziere die aktuelle Population durch Anwendung der Operationen Selektion, Kreuzung und Mutation.
3. Wiederhole Schritt 2 so lange, bis sich die Fitness der Population nicht mehr erhöht.

Das Ziel des Algorithmus ist die Fitness der Genome zu maximieren. Die Fitnessfunktion, die beliebig definiert sein kann, bewertet jedes neu entstandene Genom. Dazu muss das Genom in seinen zugehörigen Phänotyp umgewandelt werden, auf ihm operiert die Fitnessfunktion. Durch die Operation *Selektion* werden bei jedem Durchlauf durch die Schleife des Algorithmus eine bestimmte Anzahl von Genomen ausgesondert, sie sorgt also dafür, dass die Größe der Population konstant bleibt. Gleichzeitig werden (in der Regel) die fittesten Genome für die nächsten Operationen ausgewählt. Bei der Operation *Kreuzung* werden zwei Genome an einer bestimmten Stelle aufgetrennt und die beiden Bruchstücke über Kreuz zu neuen Genomen kombiniert. Die Operation *Mutation* verändert ein oder mehrere zufällig ausgewählte Gene in einem Genom, wodurch ebenfalls ein neues Genom entsteht. Abbildung 3.1 illustriert die drei Operationen.

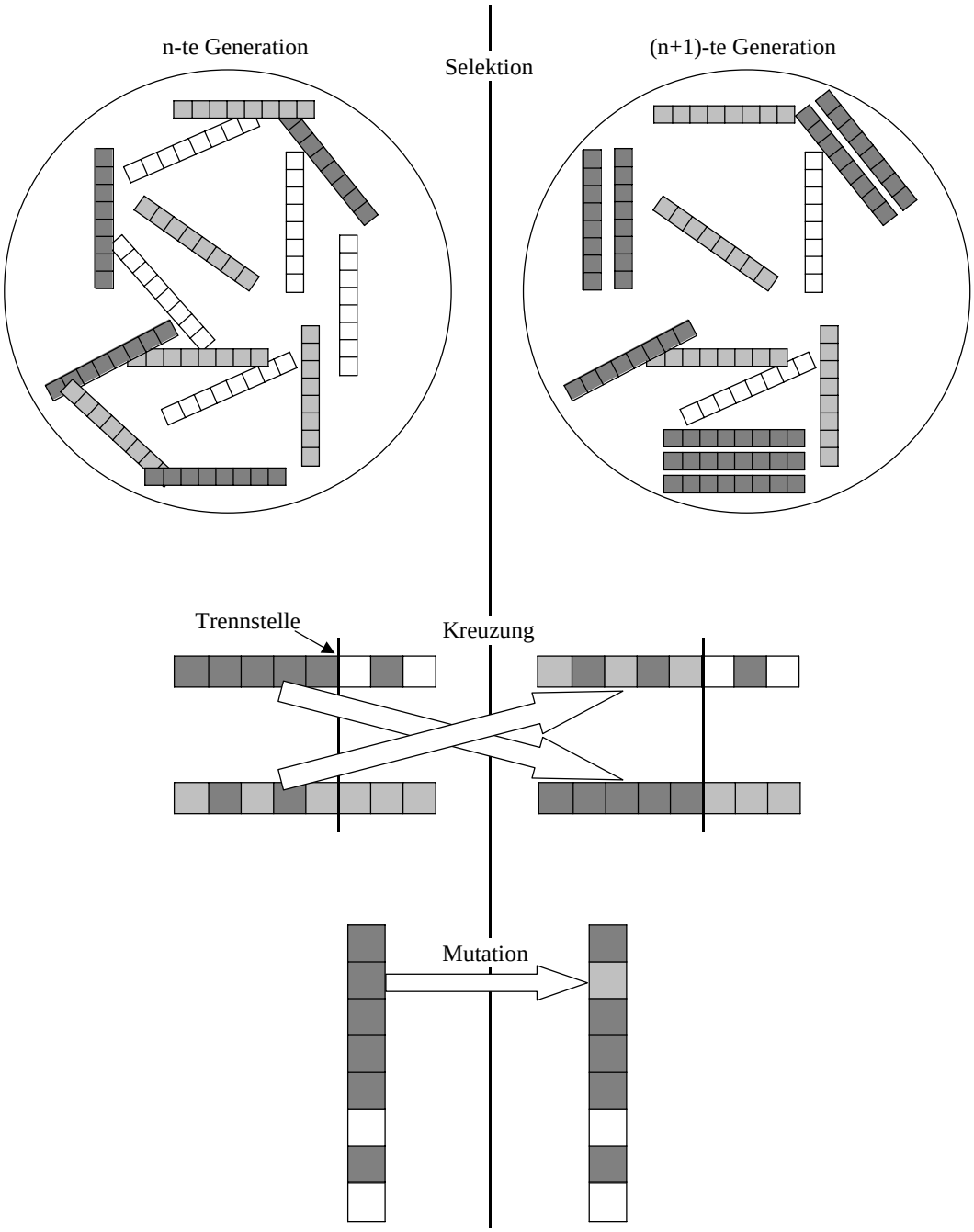

Abbildung 3.1

3.4. Zusammenfassung

In diesem Kapitel wurden verschiedene heuristische Suchmethoden betrachtet. Insgesamt stellte sich heraus, dass auch mit guten Heuristiken die Suche recht zeitaufwendig oder auch speicherplatzaufwendig sein kann.

- Algorithmen zum Problemlösen durch Optimierung speichern immer nur einen Zustand, sie können aber bei lokalen Maxima hängen bleiben. Das betrifft vor allem die Verfahren zum Bergsteigen.
- Das simulierte Ausglühen entgeht den lokalen Maxima, es ist vollständig und optimal wenn ein ausreichend langer Zeitraum für die Abkühlung gegeben ist.

- Genetische Algorithmen simulieren die Evolution. Die Basis ist eine Population, die durch Selektion, Kreuzung und Mutation von einer Generation zur nächsten weiter entwickelt wird. Die Größe der Population wird durch Entfernen der am wenigsten fitten Individuen konstant gehalten.

4. Logik erster Ordnung

In der Logik erster Ordnung besteht die Welt aus **Objekten**, d.h. Dingen, die individuelle Identitäten und **Eigenschaften** haben, nach denen sie unterscheidbar sind. Zwischen den Objekten können verschiedene **Relationen** bestehen. Einige der Relationen können **Funktionen** sein.

4.1. Syntax und Semantik

In der Logik erster Ordnung gibt es **Sätze**, die Fakten repräsentieren. Diese Sätze bestehen aus **Termen** und Prädikatsymbolen. Die Terme können funktionale Ausdrücke, Variable oder Konstanten sein. Aus einfachen Sätzen werden komplexe durch Junktoren und Quantoren aufgebaut. Die Syntax der Logik erster Ordnung lässt sich durch die folgende Grammatik beschreiben.

Satz	→	*atomarer_Satz* | *Satz Junktor Satz* | *Quantor Variable, ... Satz* | ¬*Satz* | (*Satz*)
atomarer_Satz	→	*Prädikat*(*Term, ...*) | *Term* = *Term*
Term	→	*Funktion*(*Term, ...*) | *Konstante* | *Variable*
Junktor	→	∧ | ∨ | ⇔ | ⇒
Quantor	→	∀ | ∃
Konstante	→	*A* | X_1 | *Hans* | ...
Variable	→	*a* | *x* | *s* | ...
Prädikat	→	*Vor* | *HatFarbe* | *VerwandtMit* | ...
Funktion	→	*Mutter* | *LinkesBeinVon* | ...

4.1.1. Symbole

- **Konstantensymbole**

Durch eine Interpretation wird festgelegt, auf welches Objekt in der Welt sich ein Konstantensymbol bezieht. Jedes Konstantensymbol benennt genau ein Objekt, aber nicht alle Objekte der Welt müssen Namen haben und manche Objekte können mehrere Namen haben.

- **Prädikatsymbole**

Durch eine Interpretation wird festgelegt, auf welche Relation in der Welt sich ein Prädikatsymbol bezieht. Die Relation ist als Menge von Tupeln von Objekten, die die Relation erfüllen, definiert.

- **Funktionssymbole**

Funktionssymbole werden durch Interpretationen auf spezielle Relationen abgebildet, die in einer Stelle (üblicherweise der letzten) eindeutig sind. Sie heißen Funktionen. Eine *n*-stellige Funktion

kann durch eine Menge von (n+1)-Tupeln definiert werden. Die letzte Stelle eines jeden solchen Tupels stellt den Wert der Funktion unter den ersten n Stellen dar. In der Menge gibt es keine zwei Tupel mit identischen ersten n Stellen und verschiedenen (n+1)-ten Stellen.

4.1.2. Terme

Ein **Term** ist eine logischer Ausdruck, der sich auf ein Objekt bezieht. Deshalb sind Konstanten- und Variablensymbole Terme. Außer Konstantensymbolen sind Ausdrücke, gebildet aus einem Funktionssymbol und einer in Klammern eingeschlossenen Folge von Termen wiederum Terme. Die Semantik von Termen ist folgendermaßen definiert: Ist der Term t eine Konstante, dann ist seine Bedeutung das Objekt, dem er unter einer Interpretation zugeordnet wird. Ist t ein funktionaler Ausdruck der Form $f(t_1, t_2, ..., t_m)$ dann wird f unter einer Interpretation auf eine (m+1)-stellige funktionale Relation abgebildet, die Terme t_i auf irgendwelche Objekte und die Bedeutung von t unter der Interpretation ist dann die (m+1)-te Komponente des Tupels, dessen erste m Komponenten die Bedeutungen der Terme t_i darstellen.

4.1.3. Atomare Sätze

Ein **atomarer Satz** besteht aus einem Prädikatsymbol gefolgt von einer in Klammern eingeschlossenen Liste von Termen. Ein atomarer Satz beschreibt ein Fakt. Die Bedeutung eines atomaren Satzes ist einer der Wahrheitswerte *true* oder *false*. Ein atomarer Satz ist wahr oder hat den Wert *true*, wenn die Relation, auf die sich das Prädikatsymbol bezieht, zwischen den Objekten gilt, auf die sich die Terme beziehen, d.h. wenn das entsprechende Tupel von Objekten in der Relation enthalten ist.

4.1.4. Zusammengesetzte Sätze

Aus Sätzen kann man mittels **Junktoren** komplexere Sätze bauen. Die Teilsätze können ihrerseits zusammengesetzt oder atomar sein. Die Semantik zusammengesetzter Sätze sind Wahrheitswerte und sie ist genauso definiert wie die der aussagenlogischen zusammengesetzten Sätze.

4.1.5. Quantoren

Universelle Quantifizierung

Zur Repräsentation von natürlichsprachlichen Sätzen, die allgemeine regelartige Aussagen machen, kann der Allquantor benutzt werden. Ein typischer Satz dieser Art ist etwa *Katzen sind Säugetiere*. Damit ist genauer gemeint: *Alle Katzen sind Säugetiere*. Die Begriffe *Katze* und *Säugetier* beschreiben bestimmte Klassen von Tieren, also Mengen von Objekten mit bestimmten Eigenschaften, in diesem Fall *Katze-sein* oder *Säugetier-sein*. Deshalb werden sie am besten durch einstellige Prädikate wiedergegeben. Die Formulierung *Alle Katzen sind Säugetiere* lässt sich damit so umschreiben: Wenn Objekt x eine Katze ist (zur Klasse der Katzen gehört), dann ist x auch ein Säugetier (gehört zur Klasse der Säugetiere). Diese Formulierung lässt sich direkt in einen allquantifizierten Satz in der Logik erster Ordnung umschreiben:

$$\forall x \; Katze(x) \Rightarrow Säugetier(x)$$

Schreibkonvention: Variable beginnen mit kleinen Buchstaben, Konstanten-, Prädikat- und Funktionssymbole mit Großbuchstaben.

Eine Variable ist selbst ein Term und kann deshalb als Argument einer Funktion oder eines Prädikats benutzt werden. Ein Term, der keine Variablen enthält, heißt **Grundterm**.

Existenzielle Quantifizierung

Existenziell quantifizierte Sätze können zur Repräsentation natürlichsprachlicher Sätze, die Aussagen über *einige* Objekte aus einer Menge machen, benutzt werden. Der logische Satz $\exists x\ P$ ist wahr, wenn P wahr ist für irgendein Objekt aus einem Universum von Objekten. Eine natürlichsprachliche Formulierung wie *Micky hat eine Schwester, die eine Katze ist* lässt sich als logischer Satz wie folgt wiedergeben:

$$\exists x\ Schwester(x, Micky) \wedge Katze(x)$$

Geschachtelte Quantoren

Komplexere Sätze werden oft mit mehreren Quantoren formuliert. Im einfachsten Fall hat man eine Folge gleichartiger Quantoren, also Sätze der Form $\forall x_1\ \forall x_2\ ...\ \forall x_n\ P$ oder $\exists x_1\ \exists x_2\ ...\ \exists x_n\ P$. Solche Sätze werden abkürzend geschrieben $\forall x_1, x_2, ...\ , x_n\ P$ bzw. $\exists x_1, x_2, ...\ , x_n\ P$.

Kommen Allquantoren und Existenzquantoren gemischt vor, dann ist die Reihenfolge wichtig. Eine Vertauschung der Reihenfolge verändert die Bedeutung eines Satzes. Der natürlichsprachliche Satz *Jeder liebt jemand* (*Everybody loves somebody*) heißt in logischer Notation

$$\forall x\, \exists y\ Liebt(x, y)$$

Der Satz *Es gibt jemand, der von allen geliebt wird* würde heißen

$$\exists y\ \forall x\, Liebt(x, y)$$

Eine Variable kann innerhalb eines logischen Satzes verschieden quantifiziert sein. In diesem Fall gilt immer der Quantor, in dessen Skopus die Variable steht. Eine Variable ist immer im Skopus des im Satz am tiefsten eingebetteten Quantors.

Jede Variable muss in einem syntaktisch korrekten Satz im Skopus eines Quantors stehen. Deshalb ist z.B. ein Satz der Form $\forall x\ P(y)$ nicht korrekt.

Quantoren und Negation

Die beiden Quantoren lassen sich mittels Negation ineinander überführen. Den De Morganschen Regeln für die Konjunktion und Disjunktion entsprechen Regeln für den All- und Existenzquantor:

$$\begin{array}{llll} \forall x\ \neg P & \equiv \neg\exists x\ P & \neg P \wedge \neg Q & \equiv \neg(P \vee Q) \\ \neg\forall x\ P & \equiv \exists x\ \neg P & \neg(P \wedge Q) & \equiv \neg P \vee \neg Q \\ \forall x\ P & \equiv \neg\exists x\ \neg P & P \wedge Q & \equiv \neg(\neg P \vee \neg Q) \\ \exists x\ P & \equiv \neg\forall x\ \neg P & P \vee Q & \equiv \neg(\neg P \wedge \neg Q) \end{array}$$

Gleichheit

Zur Bildung von Sätzen in der Logik erster Ordnung kann auch das Gleichheitszeichen als eine Art spezielles Prädikat verwendet werden. Sind t_1 und t_2 Terme, dann ist $t_1 = t_2$ ein Satz. Er ist wahr, wenn t_1 und t_2 unter einer Interpretation auf das gleiche Objekt referieren. Die Gleichheit wird also als Identitätsrelation interpretiert. Diese ist die zweistellige Relation, in der alle Paare in der ersten und zweiten Komponente dasselbe Objekt enthalten.

Mit Hilfe der Gleichheit und der Negation kann ausgedrückt werden, dass zwei Objekte nicht identisch sind. Ein Beispiel ist der Satz

$$\exists x, y\ Schwester(x, Micky) \wedge Schwester(y, Micky) \wedge \neg(x = y)$$

Er besagt, dass Micky (mindestens) zwei Schwestern hat.

4.2. Zwei Anwendungsbereiche für die Logik erster Ordnung

4.2.1. Die Verwandtschaftsdomäne

In der Verwandtschaftsdomäne gibt es folgende Objekte, Prädikate und Funktionen:

Objekte:	Personen
Einstellige Prädikate:	*Männlich, Weiblich*
Zweistellige Prädikate:	*Elternteil, Geschwister, Bruder, Schwester, Kind, Tochter, Sohn, Gatte, Ehefrau, Ehemann, Großelternteil, Enkel, Vetter, Base, Tante, Onkel.*
Funktionen:	*Mutter, Vater*

Zwischen den Prädikaten und Funktionen bestehen bestimmte Beziehungen, die sich mit Hilfe logischer Sätze formulieren lassen. Beispiele hierfür sind:

$$\forall m, k\ Mutter(k) = m \Leftrightarrow Weiblich(m) \wedge Elternteil(m, k)$$

$$\forall f, e\ Ehemann(e, f) \Leftrightarrow Männlich(e) \wedge Gatte(e, f)$$

$$\forall x\ Männlich(x) \Leftrightarrow \neg Weiblich(x)$$

4.2.2. Axiome, Definitionen und Theoreme

Sätze in einer Wissensbasis werden als **Axiome** betrachtet, d.h. als Aussagen, von denen man annimmt, dass sie gültig sind. Mittels **Definitionen** kann man aus Axiomen neue Konzepte erzeugen und aus Axiomen und Definitionen lassen sich **Theoreme** ableiten. Beim Aufbau einer Wissensbasis gibt es zwei Probleme:

- Reichen die Axiome und Definitionen zur Beschreibung einer Domäne aus?
- Ist die Domäne durch die Axiome und Definitionen überspezifiziert?

Zur Lösung des ersten Problems kann man versuchen eine Menge von Prädikaten anzunehmen, die man als elementar betrachtet, und alle übrigen benötigten Prädikate mittels dieser zu definieren. Allerdings ist es nicht in allen Domänen möglich eine sinnvolle Menge von elementaren Prädikaten zu bestimmen.

Das zweite Problem ist gewissermaßen die Umkehrung des ersten. In mathematischen Domänen versucht man Mengen voneinander **unabhängiger** Axiome zu definieren, d.h. solche, bei denen kein Axiom aus den übrigen ableitbar ist. In der KI nimmt man Überspezifikation in Kauf, denn sie erhöht die Effizienz von Ableitungen.

Wird ein Prädikat durch andere Prädikate **definiert**, so schreibt man meistens

$$\forall x, y\ P(x, y) \equiv ...$$

um auszudrücken, welche Eigenschaften Objekte haben müssen um *P* zu erfüllen.

4.2.3. Die Mengendomäne

In der Mengendomäne gibt es folgende Konstanten, Prädikate und Funktionen:

Konstanten: *LeereMenge*
Prädikate: *Element, Teilmenge, Menge*
Funktionen: *Durchschnitt, Vereinigung, Einfügen*

Die folgenden acht Axiome definieren die Eigenschaft *Menge*.

$$\forall m\ Menge(m) \Leftrightarrow (m = LeereMenge) \vee (\exists\ x, m'\ Menge(m') \wedge m = Einfügen(x, m'))$$

Für Mengen- und Listenoperationen werden im Folgenden die in der Mathematik üblichen abkürzenden Notationen verwendet. Die einzige Notation, die hier benutzt wird und nicht allgemein üblich ist, ist $\{a|m\}$ für *Einfügen*(*a*, *m*). Sie sind keine echten Erweiterungen des Axiomensystems für Mengen bzw. Listen, sie sind aber als Abkürzungen praktisch. Solche nur notationellen Erweiterungen werden oft als **syntaktischer Zucker** bezeichnet. Für Listen werden die in Prolog üblichen Notationen verwendet.

$$\begin{aligned}
\varnothing &= \text{LeereMenge} & [] &= Nil\\
\{x\} &= Einfügen(x, LeereMenge) & [x] &= Cons(x, Nil)\\
\{x, y\} &= Einfügen(x, Einfügen(y, LeereMenge)) & [x, y] &= Cons(x, Cons(y, Nil))\\
\{x, y|m\} &= Einfügen(x, Einfügen(y, m)) & [x, y|l] &= Cons(x, Cons(y, l))\\
m \cup n &= Vereinigung(m, n)\\
m \cap n &= Durchschnitt(m, n)\\
x \in m &= Element(x, m)\\
n \subseteq m &= Teilmenge(n, m)
\end{aligned}$$

Arithmetische und Vergleichsausdrücke werden in der üblichen Infixnotation geschrieben.

4.2.4. Fragen und Antworten

Die Axiome und Fakten einer Domäne werden mittels TELL in die Wissensbasis *KB* eingefügt. Mittels ASK können dann logische Konsequenzen aus den eingefügten Sätzen abgefragt werden. Ein Aufruf von TELL zum Einfügen eines Satzes hat z.B. folgende Form:

$$\text{TELL}(KB, (\forall m, k\ Mutter(k) = m \Leftrightarrow Weiblich(m) \wedge Elternteil(m, k)))$$

entsprechend für die weiteren Axiome. Fakten können z.B. in folgender Form eingegeben werden:

$$\text{TELL}(KB, (Weiblich(Tifanny) \wedge Elternteil(Tifanny, Micky) \wedge Elternteil(Micky, Felix)))$$

Eine Abfrage mit ASK hat folgende Gestalt:

$$\text{ASK}(KB, Großelternteil(Tifanny, Felix))$$

Sätze, die mittels TELL zur Wissensbasis hinzugefügt werden, werden **Zusicherungen** genannt und Sätze, die mittels ASK abgefragt werden, werden **Fragen** oder **Ziele** genannt.

Eine Frage, die mit einem existenziell quantifizierten Satz formuliert ist, erwartet als Antwort ein oder mehrere Objekte, so dass bei Einsetzen dieser Objekte für die Variable jeweils ein aus der Wissensbasis ableitbarer Satz entsteht. Heißt die Frage z.B.

$$\text{ASK}(KB, \exists x\ Kind(x, Micky))$$

dann sollte die Antwort *Felix* sein. Die Standardform der Antwort auf eine Frage dieser Art ist eine Liste von Variable/Term-Paaren, genannt **Substitution** oder **Bindungsliste**. Im Beispiel besteht die Liste nur aus einem Paar, nämlich $\{x/Felix\}$.

4.3. Zusammenfassung

In diesem Kapitel wurde gezeigt, wie die Logik erster Ordnung als Wissensrepräsentationssprache benutzt werden kann. Die wichtigsten Punkte sind:

- Die **Logik erster Ordnung** ist eine allgemeine Repräsentationssprache, deren ontologische Beschreibungsmöglichkeiten die Darstellung von Objekten und Relationen in der Welt einschließen.
- Objekte werden durch **Konstantensymbole**, Relationen durch **Prädikatsymbole** benannt. **Komplexe Terme** benennen Objekte unter Benutzung von **Funktionssysmbolen**. Die Interpretation spezifiziert, worauf sich die Symbole beziehen.
- Ein **atomarer Satz** besteht aus einem Prädikatsymbol, das auf einen oder mehrere Terme angewandt ist. Er ist wahr, wenn die Relation, die durch das Prädikatsymbol benannt ist, zwischen den Objekten gilt, die durch die Terme benannt sind. **Komplexe Sätze** werden aus atomaren mittels Junktoren aufgebaut und **quantifizierte Sätze** erlauben die Formulierung allgemeiner Regeln.