Künstliche Intelligenz in der Schule

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.

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

S.J. Russell, P. Norvig: Künstliche Intelligenz. Ein moderner Ansatz. Pearson Education Germany, München, 2004.

1. Was ist Künstliche Intelligenz?

1.1. Wie intelligent sind Computer?

Was können Computer heute schon? Allgemein bekannt sind folgende Fähigkeiten:

- Sie können unheimlich schnell rechnen.
- Sie können riesige Datenmengen verwalten.
- Sie sind die Grundlage des Internets.

Wir Menschen haben uns schon lange daran gewöhnt, dass uns die Computer bei diesen Fähigkeiten weit überlegen sind. Alle diese Tätigkeiten lassen sich auch manuell erledigen (auch das Internet, das man als weltweites Dokumentenverwaltungssystem betrachten kann), allerdings nur prinzipiell, praktisch nicht, weil wir mit unseren begrenzten Fähigkeiten zu unseren Lebzeiten viele Probleme nicht lösen könnten, die Computer in wenigen Minuten oder gar nur Sekunden erledigen. Aber Computer können inzwischen schon mehr:

- Sie können Sprache verstehen.
- Sie können Schrift lesen.
- Sie können erkennen, was auf einem Bild dargestellt ist.
- Sie können Probleme lösen, für die sonst viel menschliches Erfahrungswissen notwendig ist.
- Sie können lernen, d.h. ihr Wissen selbstständig erweitern.

Zugegeben, diese Fähigkeiten sind nicht so gut ausgebildet wie die ersten drei genannten, und Menschen sind darin meist noch besser als Computer, aber immerhin gibt es auch hier schon sogar praktisch eingesetzte Systeme. Man muss sich also fragen, wo sind die Grenzen der Computer, oder gibt es solche überhaupt nicht und ist es nur eine Frage der Zeit, bis die Computer uns in jeder Hinsicht überlegen sind? Manche Wissenschaftler glauben das, z.B. Ray Kurzweil (siehe z.B. http://www.kurzweilai.net/meme/frame.html?m=7). Der allererste Computer wurde vor etwa 65 Jahren gebaut, wenn man bedenkt, was in dieser relativ kurzen Zeit daraus geworden ist und was heutige Maschinen leisten, dann scheint es nicht sehr sinnvoll zu sein über absolute Grenzen der Technik zu spekulieren.

1.2. Können Computer denken?

Als höchste geistige Tätigkeit des Menschen, wodurch er sich von allen Tierarten unterscheidet, betrachtet man allgemein seine Fähigkeit zu denken, d.h. Inhalte miteinander auf abstrakter Ebene in Verbindung zu bringen. Menschen können komplizierte Theorien entwickeln, und vielfach, vor allem in den Naturwissenschaften, sind diese auch für das Verständnis der Welt notwendig. Ein wesentlicher Grund dafür ist die Beschränktheit unserer Wahrnehmungsfähigkeit durch unsere Sinnesorgane und die Notwendigkeit für das Gehirn, aus dieser Eingabe ein einigermaßen sinnvolles Bild der Welt zu konstruieren. Das Gehirn baut sich komplexe Repräsentationen der Welt auf, und die brauchen wir um uns in der Welt zurecht zu finden.

Schaut man sich die im vorigen Abschnitt aufgezählten Punkte (Sprache verstehen, Schrift lesen usw.) an, dann muss man zugeben, dass dafür ein gewisses Maß an Denkfähigkeit erforderlich ist, das wir also dem Computer zubilligen müssen, jedenfalls von außen betrachtet. Selbstverständlich können Computer noch nicht selbstständig Forschung betreiben und noch keine komplizierten physikalischen oder mathematischen Theorien entwickeln. Sie können allerdings schon mit Hilfe

von Sensoren als Roboter Wahrnehmungen machen und diese intern verarbeiten und sie können sich eigene interne Repräsentationen aufbauen.

Von außen betrachtet besitzen also Computer so etwas wie Denkfähigkeit, und diese vergrößert sich in der Zukunft sicherlich noch erheblich. Man kann sich aber fragen, ob sie beim Denken intern das Gleiche tun wie Menschen. Früher nahmen das viele Leute naiv an. Inzwischen haben die Gehirnforscher sehr viel über Aufbau und Funktionsweise des Gehirns herausgefunden und jetzt sind die Unterschiede zwischen Gehirn und Maschine ziemlich gut zu sehen. Die sind allerdings sehr grundlegend. Man kann die Arbeitsweise des Gehirns auf dem Computer simulieren, wenn auch in sehr viel kleinerem Maßstab. Aber immerhin ist zu erkennen, dass man dabei ähnliche Effekte erzielt wie das Gehirn selbst.

Die Gehirnleistungen, die wir am ehesten mit dem Begriff „Denken“ verbinden, nämlich Sprache verstehen, Schrift lesen usw., könnten also auf einer entsprechenden technologischen Basis in ähnlicher Form und in ähnlichem Umfang von einem künstlichen System erbracht werden wie die Menschen es tun. Solche Systeme sind wahrscheinlich eines Tages realisierbar, denn im Ansatz und im kleinen Maßstab geht es schon heute.

1.3. Können Computer Gefühle haben?

Die meisten, wenn nicht alle Computerlaien würden hier antworten: niemals! Das geht uns Menschen doch zu sehr unter die Haut, denn ein Computer wird eben als Maschine betrachtet, und eine solche kann keine Gefühle haben. Dass die Computer zu solchen Denkleistungen imstande sind oder sein werden wie wir, damit können wir uns noch abfinden, aber Emotionen? Die betrachten wir als etwas genuin menschliches, oder vielleicht etwas allgemeiner als etwas, was höheren Lebewesen vorbehalten ist.

Es gibt aber Psychologen, die sehen das anders. Einer der bekanntesten von ihnen ist Dietrich Dörner (Uni Bamberg). Er geht davon aus, dass Emotionen ihren Sitz im Gehirn haben und deshalb auf Mechanismen beruhen, die auch sonst im Gehirn vorkommen. Wenn man nun Gehirnfunktionen auf dem Computer simulieren kann, dann müsste das auch mit Emotionen möglich sein. Er hat einige interessante Programme entwickelt, die das belegen sollen. Dabei möchte er insbesondere demonstrieren, wie weit Emotionen rationale Entscheidungen beeinflussen.

Natürlich gilt auch hier wieder, dass man vom Vorbild noch weit entfernt ist. Vor allem aber wird von anderen Forschern der Einwand gemacht, dass es sich bei den Emotionen um wesentlich andere Vorgänge handelt als bei den rein kognitiven Tätigkeiten. Bei den Emotionen sind nämlich noch andere Gehirnpartien im Spiel als nur das Großhirn (der Kortex), vor allem das limbische System, außerdem spielen andere Steuerungssysteme im Körper, wie das Hormonsystem und sogar das Immunsystem, eine wichtige Rolle. Diese Steuerungssysteme beeinflussen sich gegenseitig, und das trifft ganz besonders bei den Emotionen zu. Man kann zwar emotionales Verhalten simulieren, aber es ist dann eine philosophische Frage, ob ein Computer oder ein Roboter ein solches Verhalten hat oder „nur“ simuliert.

Diese Frage hat durchaus praktische Bedeutung. In der Robotertechnik interessiert man sich für die sichtbare Simulation von Emotionen, und zwar bei solchen Robotern, die mit Menschen in Kontakt treten sollen oder gar mit ihnen kommunizieren sollen. Das kann in der Zukunft eine Rolle spielen bei Robotern, die zur Unterstützung behinderter Personen verwendet werden sollen. Die Erfahrungen mit Robotern, insbesondere in großem Umfang mit den Roboterhunden, zeigt, dass Menschen relativ leicht Emotionen in ein Wesen projizieren, das im Verhalten und insbesondere in der Mimik (soweit vorhanden) etwas zeigt, was als emotionale Äußerung interpretiert werden kann. In dem

Forschungsprojekt Kismet wird untersucht, wie Menschen auf mimische Verhaltensweisen von Robotern reagieren, vgl. http://www.ai.mit.edu/projects/humanoid-robotics-group/kismet/kismet.html.

In diesen Zusammenhang gehört auch die Frage, ob künstliche Systeme so etwas wie Selbstbewusstsein haben können. Höheren Tieren wird Selbstbewusstsein zugebilligt, allerdings war man darüber in früheren Jahrhunderten anderer Meinung. Ob künstliche Systeme, z.B. Roboter, Selbstbewusstsein haben können ist deshalb schwer zu sagen, weil man sich in der Philosophie, Psychologie und verwandten Wissenschaften über den Begriff noch nicht im Klaren ist. Sollte Selbstbewusstsein rein als Leistung einer höheren Intelligenz definiert werden, dann wird man sie eines Tages auch künstlichen Systemen nicht absprechen können.

1.4. Können sich Computer sozial verhalten?

Konkreter gefragt heißt das: Können Computer miteinander kommunizieren, können sie kooperieren um gemeinsam Aufgaben zu lösen, können sie sich selbstständig in einer Gemeinschaft organisieren, können sie dabei eigene Verhaltensweisen ausbilden, ja sogar: können sie selbstständig eine Sprache ausbilden? Das alles leisten die Menschen in sozialen Gemeinschaften, und manche dieser Fähigkeiten findet man auch bei Tieren.

In der KI-Forschung gibt es ein Teilgebiet, das sich „Multiagentensysteme“ nennt, dort werden genau diese Fragen untersucht und es werden Entwicklungen betrieben, die nachweisen sollen, dass künstliche Systeme ähnliche Leistungen erbringen können. Kommunikation zwischen Computern ist heutzutage eine Selbstverständlichkeit, ohne sie gäbe es das Internet nicht. Dafür gibt es eine Menge Sprachen, die allerdings von den Entwicklern vorgegeben sind. Es gibt aber durchaus Parallelen zur Kommunikation zwischen Menschen: Erhält ein Computer ein Datenpaket, hat aber keinen Interpreter dafür, dann kann er mit den Daten nichts anfangen. Hat er aber einen geeigneten Interpreter, dann kann ist das Datenpaket für ihn eine Nachricht, auf die er reagieren kann. Das ist bei Menschen nicht anders, sie verstehen immer nur wenige oder nur eine Sprache, Sendungen in anderen Sprachen sind für sie wertlos, weil sie nicht den richtigen Interpreter für diese Sprachen besitzen.

Kooperation ist schon schwieriger, aber auch dafür gibt es schon zahlreiche Beispiele, vor allem beim parallelen und verteilten Rechnen. Auch das ist eine Form von Kooperation, wenn auch eine nach festen Regeln, bei denen den beteiligten Maschinen bestimmte Aufgaben zugeteilt werden. Man kann aber Computer mit mehr Selbstständigkeit oder *Autonomie* ausstatten und sie damit befähigen, Kooperation untereinander selbst zu organisieren. Das hängt eng mit der Selbstorganisation zusammen. Man stellt der ganzen Agentengesellschaft eine Aufgabe, und diese muss sich selber so koordinieren und die Aufgaben verteilen, dass die Aufgabe insgesamt gelöst wird.

Irgendwann und irgendwie haben es die Menschen geschafft, Formen des Zusammenlebens und der sprachlichen Verständigung zu entwickeln. Ansätze dazu gibt es allerdings schon im Tierreich. In der Multiagentenforschung interessiert man sich dafür, ob künstliche Systeme auch zu einer solchen Leistung imstande sind. Es gibt dazu einige interessante Ansätze, z.B. die *Talking Heads* von Luc Steels (http://talking-heads.csl.sony.fr/). Die Frage ist dabei vor allem, wie die Semantik der Sprache entstand. Diese Fragen sind auch im Zusammenhang mit der Kommunikation zwischen Menschen und Robotern von Interesse.

1.5. Womit beschäftigt sich die Künstliche Intelligenz?

In der KI versucht man, intelligente Fähigkeiten, wie sie am ausgeprägtesten beim Menschen zu beobachten sind, mit dem Computer zu simulieren. Die erste Voraussetzung dafür ist, menschliches Wissen in den Computer zu bringen um es dort verarbeiten zu können. Damit fangen die Schwierigkeiten bereits an. Es genügt nicht, nur irgendwelche Dokumente einzulesen, z.B. Bücher aller Art, das könnte man noch maschinell machen. Das darin enthaltene Wissen ist nicht unmittelbar verarbeitbar, vielmehr muss es in eine spezielle Form gebracht werden, damit dies möglich ist. Eine solche Form nennt man allgemein eine formale Sprache. Programmiersprachen sind die bekanntesten Beispiele für formale Sprachen. Für die Darstellung von Wissen im Computer werden aber andere Sprachen verwendet, z.B. Logik-Sprachen. Wichtig ist, dass es die Sprache erlaubt, Inhalte nach genauen Regeln miteinander zu verknüpfen um neue Inhalte zu gewinnen. Diese Regeln müssen ganz exakt sein, damit man sie im Computer in einem automatisch ablaufenden Programm verwenden kann. Bei Logik-Sprachen z.B. gibt es solche Regeln, sie heißen *Inferenzregeln*, und das Erschließen neuer Inhalte mit Hilfe der Regeln nennt man *logisches Schließen*.

Wir haben schon festgestellt, dass bei reinen Rechenprozessen die Computer uns Menschen weit überlegen sind. Typisch „intelligente“ Aufgaben haben aber einen anderen Charakter als reine Rechenaufgaben. Sie sind *Problemlösungsaufgaben*. Die Besonderheit dieser Art von Aufgaben ist, dass zu Beginn eine Beschreibung eines (Anfangs-) Zustands und eine Aufgabenbeschreibung vorliegen sowie eine Menge möglicher Operationen (z.B. in Form von Regeln). Der Problemlöser hat nun die Aufgabe einen Zustand zu finden, der eine Lösung der Aufgabe darstellt. Das kann er dadurch tun, dass er eine im Augenblick geeignete Operation auswählt, diese auf den gegenwärtigen Zustand anwendet und so zu einem neuen Zustand kommt. Dies muss er so lange machen, bis er einen Lösungszustand erreicht hat. Man nennt den ganzen Prozess *Problemlösen durch Suchen*. Er ist ein so allgemeines Verfahren, dass er im Prinzip immer angewendet werden kann, sobald ein Problem in einer formalen Beschreibung im Computer gegeben ist. Er hat aber den Nachteil, dass er sehr zeitaufwendig werden kann, deshalb versucht man für spezielle Probleme speziellere Verfahren zu verwenden.

Eine wichtige Aufgabe ist des Planen. Menschen können das sehr gut. Das Planen ist ein gutes Beispiel für allgemeines Problemlösen. Man kann also mit Suchverfahren ein Planungsproblem immer lösen. Um es aber effizienter tun zu können, hat man spezielle Verfahren für das Planen entwickelt. Eine wichtige Voraussetzung dafür ist die detailliertere Beschreibung des Planungsproblems als es mit einer allgemeinen Zustandsformulierung möglich ist. Logik-Sprachen bieten dazu die Möglichkeit. Was man mit einem spezielleren Verfahren erreicht ist, dass die Zahl der nutzlosen Schritte, die zu keiner Lösung führen, reduziert wird.

Wie in Abschnitt 1.4 erwähnt, ist die Untersuchung von Kommunikation und Kooperation zwischen Computern ein wichtiges Teilgebiet der KI. Das liegt zunächst einmal daran, dass Computersysteme oder *Agenten* mehr leisten können, wenn sie kooperieren, und dass viele Probleme von Natur aus verteilt sind, so dass ein verteilter Ansatz zur Lösung günstig erscheint. Das lässt sich besonders gut am Beispiel der „reaktiven Agenten“ zeigen, einem Typ besonders einfacher Agenten. Man kann sich die reaktiven Agenten ungefähr wie Ameisen vorstellen, also sehr einfache Tiere, bei denen das einzelne Exemplar nicht viel leisten kann, nicht einmal allein überlebensfähig ist, aber die Gemeinschaft erstaunliche Dinge zustande bringt. Ein weiterer wichtiger Grund für die Beschäftigung mit den Multiagentensystemen ist die Einsicht, dass Intelligenz ein soziales Phänomen ist. Die Intelligenz eines Menschen bildet sich nur im Kontakt mit anderen Menschen aus, wäre ein Mensch von Geburt an auf sich allein gestellt, dann würde sie sich nicht entwickeln, ein berühmtes Beispiel dafür sind die „Wolfskinder“.

Die publikumswirksamsten Entwicklungen der KI sind die in der Robotik, genauer gesagt in der mobilen Robotik. Die klassischen Roboter, die schon seit langem in der Industrie eingesetzt werden, sind herkömmlich programmiert, sie sind eigentlich nur Maschinen ohne „Intelligenz“. Die mobilen Roboter benötigen aber ein beträchtliches Maß an Intelligenz, damit sie sich selbstständig oder *autonom* in einer bestimmten Umgebung bewegen können. So entsteht leicht die Science fiction Vorstellung von Robotern, die menschenähnlich sind, womöglich sogar den Menschen überlegen, und die am Ende die Menschen überflüssig machen. Roboter können aber, wenn sie vernünftig eingesetzt werden, sehr nützliche Maschinen sein Hier geht es darum, was der Stand der Entwicklung ist und wie man einfache Roboter auch zu Lehrzwecken einsetzen kann.

Eine besonders wichtige Fähigkeit des Menschen ist die des Lernens. Menschen, oder genauer gesagt unser Gehirn, lernen ständig. Das wird uns nur meist nicht bewusst. Das Gehirn ist aber immer damit beschäftigt, Eindrücke von außen zu verarbeiten und dabei neue Inhalte herauszufiltern und sie in das schon vorhandene Wissen zu integrieren. Auch Computern kann man das Lernen beibringen, wenigstens bestimmte Arten davon. Eine Form des Lernens, die Menschen schon im Kleinkindalter praktizieren, ist das Lernen von Begriffen und Regeln, und diese Form beherrscht auch ein Computer wenn er entsprechend programmiert ist. Man nennt das dann *Maschinelles Lernen*. Der Lernprozess verläuft kontrolliert, deshalb spricht man hier von *überwachtem Lernen* oder *Lernen mit Lehrer*. Man kann sich das etwa so vorstellen wie bei einem Kleinkind, dem man erklären will, was ein Tisch, ein Stuhl oder ein Auto ist. Dazu zeigt man auf bestimmte Gegenstände und sagt „das ist ein ...“ bzw. „das ist kein ...“. Ganz ähnlich lernt ein Computer.

Es gibt noch eine andere Form des Lernens, die man mit Computern realisieren kann, und zwar mit (künstlichen) Neuronalen Netzen. Das sind letztlich auch Computerprogramme, in die man etwas eingibt und von denen man eine Ausgabe erhält. Allerdings sind das keine Begriffe, sondern in der Regel relativ abstrakte Kodierungen von Inhalten. Man spricht deshalb auch von „subsymbolischem Lernen“ im Gegensatz zu „symbolischem Lernen“ wie oben. Neuronale Netze führen im Prinzip Prozesse durch, die in ähnlicher Form auch im Gehirn ablaufen, nur sind sie dort noch komplexer. Deshalb interessieren sich die Hirnforscher für sie und verwenden sie als Experimentierwerkzeuge. Auch bei den Neuronalen Netzen gibt es überwachtes Lernen, aber es gibt auch andere Formen, die man allgemein als *unüberwachtes Lernen* bezeichnet.

Es gibt noch eine Reihe anderer Gebiete in der KI, z.B. das Verstehen von Sprache, das Erzeugen von Sprache, das Erkennen von Bildern, die Verarbeitung von unsicherem Wissen und andere. Sie können hier nicht alle behandelt werden, und lassen sich auch nicht so günstig für Unterrichtszwecke verwenden.

2. Probleme lösen durch Suchen

2.1. Wie formuliert man Probleme?

1. Beispiel: Ein Tourenproblem

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. Herr *A* hat eigentlich noch andere Interessen, 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.

2. Beispiel: Staubsauger-Roboter

Die Staubsaugerwelt bestehe aus zwei Plätzen. An jedem Platz kann sich Schmutz befinden oder nicht und der *Staubsauger-Roboter* (*SR*) befindet sich am einen oder am anderen Platz. In dieser Welt gibt es acht verschiedene Zustände, die graphisch in Abbildung 2.1 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 2.1 erfüllt.

Die Problembeschreibung für den Staubsauger-Roboter kann man auf zwei verschiedene Weisen machen.

1. Ein-Zustands-Problem

Der Problemlöser (d.h. hier: der Roboter) weiß, in welchem Zustand er sich befindet 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.

2. Mehr-Zustands-Problem

Der Problemlöser weiß nicht, in welchem Zustand er sich befindet, z.B. weil er keinen Sensor hat um dies festzustellen oder weil seine Sensoren den Zustand nicht mit Sicherheit bestimmen können, 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.

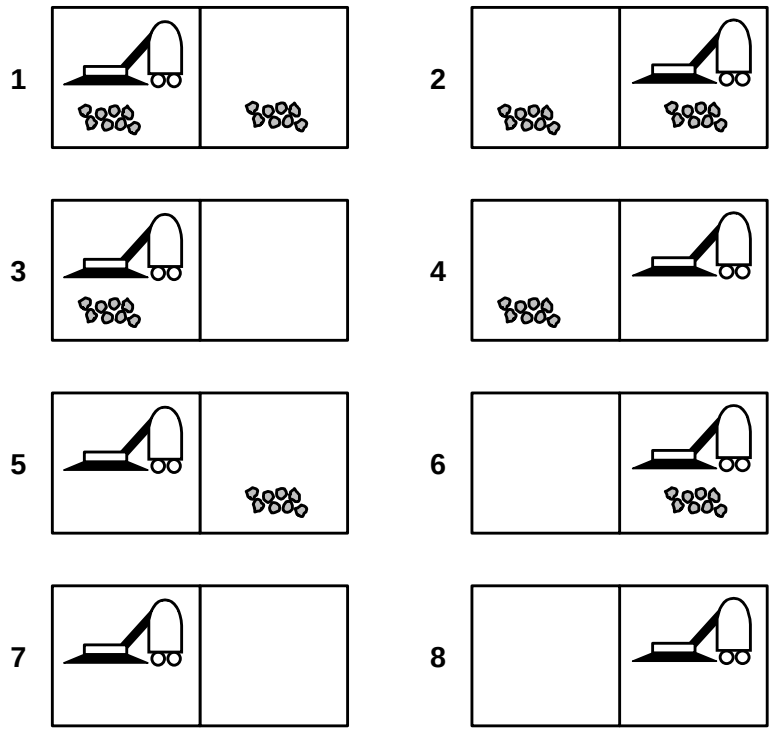

Abbildung 2.1

2.1.1. 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.

Die vier Bestandteile einer Problembeschreibung kann man als **Datentyp** *Problem* definieren, und der sieht dann so aus:

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 *Qualität* 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.

2.1.2. 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 2.2 zeigt eine vereinfachte Landkarte mit den Straßenverbindungen zwischen verschiedenen Orten.

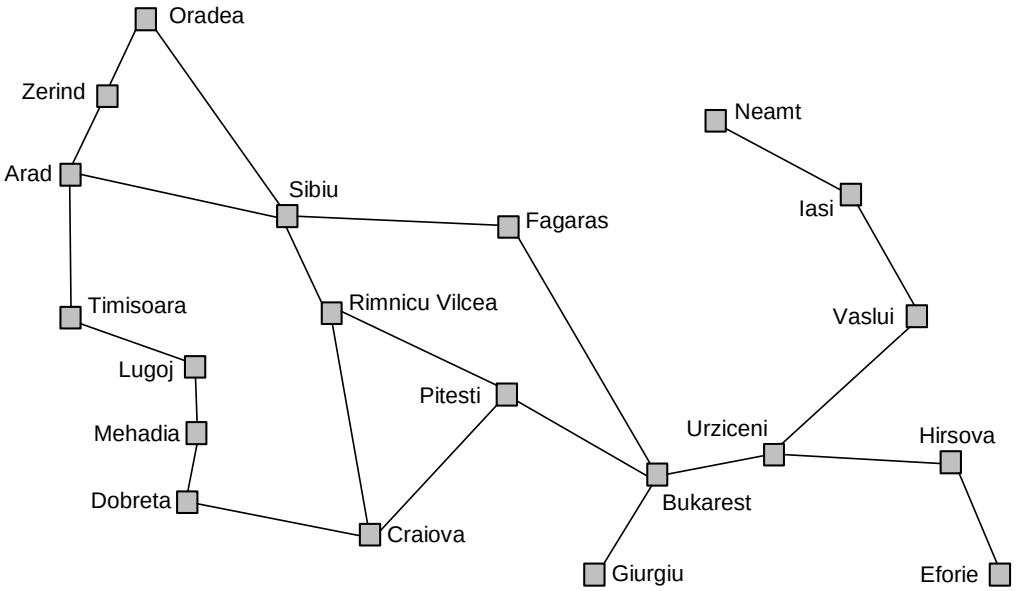

Abbildung 2.2

Das Problem heißt: „Fahre von Arad nach Bukarest unter Benutzung der Landkarte von Abbildung 2.2.“ 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 ent-

sprechen 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.

2.2. Einige Beispielprobleme

1. Das 8-Puzzle

Das 8-Puzzle ist das in Abbildung 2.3 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 2.3 dargestellte.

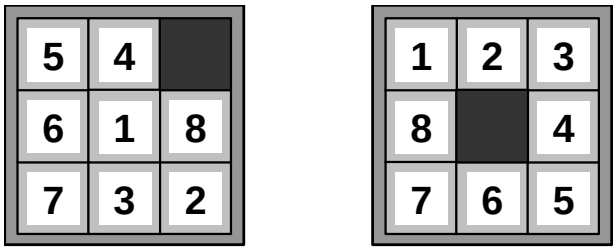

Abbildung 2.3

- **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 2.3 dargestellten Zielzustand.
- **Pfadkosten**: Jeder Schritt hat die Kosten 1, die Kosten eines Pfades sind damit gleich der Länge des Pfads.

2. Das 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 2.4 zeigt eine Anordnung der Damen, die die Bedingung verletzt.

- **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.

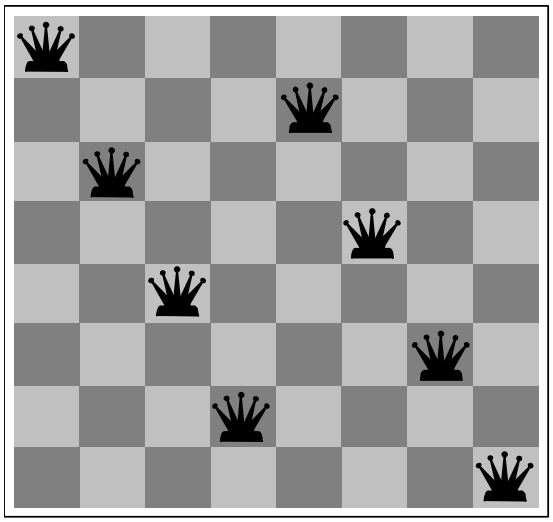

Abbildung 2.4

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.

3. 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.

4. 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 2.1.
- **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 2.5 zeigt den vollständigen Zustandsraum mit allen möglichen Pfaden.

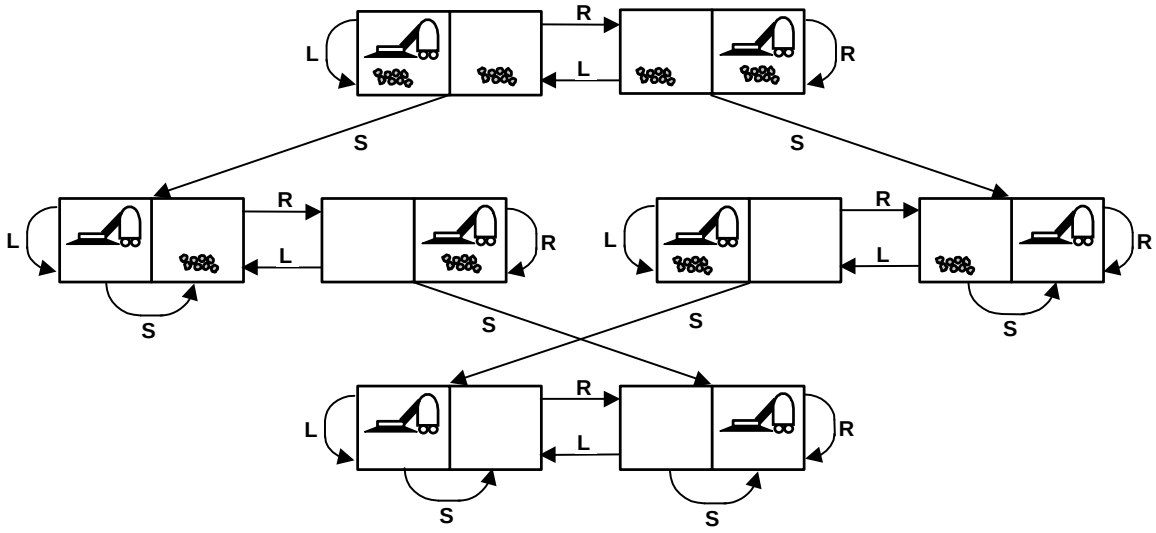

Abbildung 2.5

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

- **Zustandsmengen**: Teilmengen der acht Zustände von Abbildung 2.1.
- **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.

Der Startzustand ist die Menge aller Zustände, da der Roboter keinen Sensor hat um seinen Platz festzustellen. Abbildung 2.6 zeigt den Zustandsmengenraum für das Problem.

5. Das 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.

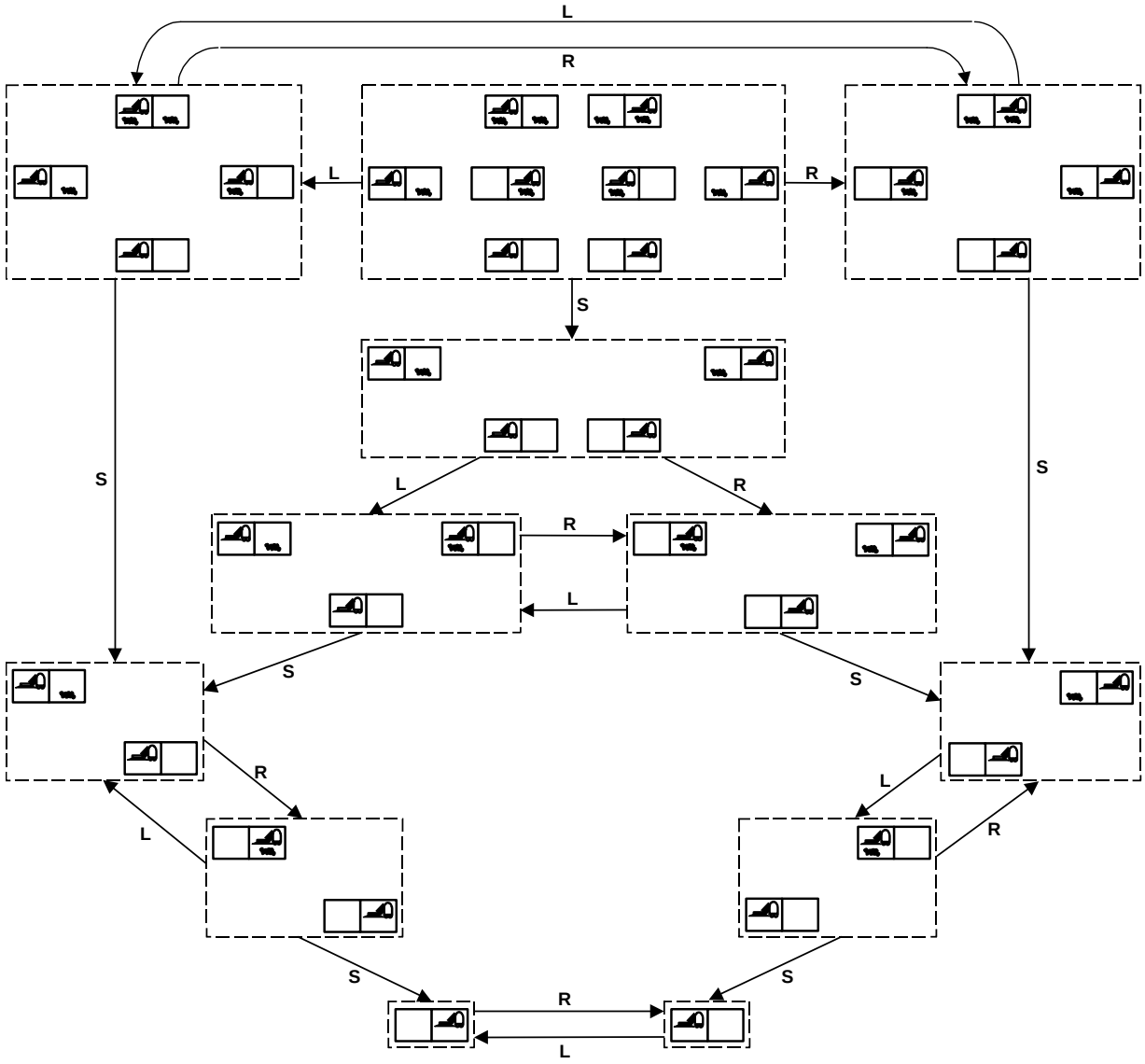

Abbildung 2.6

2.3. Suche nach Lösungen

2.3.1. 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.

2.3.2. 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
```

2.3.3. Strategien zum Suchen

Die Qualität einer Suchstrategie lässt sich nach vier Kriterien abschätzen:

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

Wie geht man nun vor, um eine Lösung zu finden? Die einfachste Möglichkeit ist, irgendeinen beliebigen Zustand aus dem Rand herauszugreifen, eine Aktion darauf anzuwenden und die sich daraus ergebenden Zustände daraufhin zu prüfen, ob sie Zielzustände sind. Diese Strategie wäre das Stochern mit der Stange im Nebel, denn sie verläuft rein zufällig. Damit kann man zwar auch zum Ziel kommen, und nach dem Zufallsgesetz ist es sogar garantiert, dass man einen Zielzustand erreicht, d.h. das Verfahren wäre vollständig, aber der Zeitaufwand kann sehr groß werden, er lässt sich nicht einmal abschätzen. Das Verfahren wäre mit Sicherheit nicht optimal.

Statt rein zufällig zu verfahren, kann man die Suche systematisch betreiben, allerdings ohne zu wissen, welche Richtung die günstigste ist. Deshalb nennt man diese Strategien *blind* oder *uninformiert*. Sie sind aber systematisch, d.h. sie arbeiten die Zustände nach einem bestimmten Schema ab.

Aus der Problembeschreibung kann man oft Hinweise dafür bekommen, welche Richtung für die Suche günstig ist, d.h. in welcher Richtung man hoffen kann, möglichst rasch zu einem Zielknoten zu kommen. Diese Richtung sollte man bevorzugen. Diese Hinweise werden in so genannten *heuristischen Funktionen* formuliert, deshalb nennt man diese Strategien *heuristisch* oder *informiert*.

2.4. Blinde Suchverfahren

2.4.1. 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.

2.4.2. 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.

2.4.3. 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 $O(b \cdot l)$.

2.4.4. 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.

2.5. 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 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

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

3. Planen

3.1. Ein einfacher Planungsalgorithmus

Wenn der Weltzustand vollständig bekannt ist kann ein Planer Wahrnehmungen nutzen, die von der Umgebung geliefert werden, um ein vollständiges und korrektes Modell des aktuellen Weltzustands aufzubauen. Ist ein Ziel gegeben, dann kann er einen geeigneten Planungsalgorithmus aufrufen um einen Aktionsplan zu erzeugen. Danach kann der Planer den Plan Aktion für Aktion ausführen. Der folgende Algorithmus implementiert einen einfachen Planer.

function EINFACHER-PLANER(*Wahrnehmung*) **returns** eine *Aktion*
  static: *KB*, eine Wissensbasis mit Aktionsbeschreibungen
    p, ein Plan, zu Beginn *NoPlan*
    t, ein Zähler für die Zeit, zu Beginn 0
  local variables: *G*, ein Ziel
    current, eine Beschreibung des aktuellen Zustands

  TELL(*KB*, MAKE-PERCEPT-SENTENCE(*Wahrnehmung*, *t*))
  current ← STATE-DESCRIPTION(*KB*, *t*)
  if *p* = *NoPlan* **then**
    G ← ASK(*KB*, MAKE-GOAL-QUERY(*t*))
    p ← IDEAL-PLANNER(*current*, *G*, *KB*)
  if *p* = *NoPlan* **or** *p* ist leer **then** *Aktion* ← *NoOp*
  else
    Aktion ← FIRST(*p*)
    p ← REST(*p*)
  TELL(*KB*, MAKE-ACTION-SENTENCE(*Aktion*, *t*))
  t ← *t* + 1
  return *Aktion*

3.2. Vom Problemlösen zum Planen

Drei *Schlüsselideen*, die dem Planen zugrundeliegen:

1. „Öffne“ die Repräsentation der Zustände, Ziele und Aktionen. Dies wird erreicht durch Verwendung einer geeigneten Repräsentationsform, z.B. der Logik erster Ordnung oder einer Teilmenge von ihr. Zustände und Ziele werden darin als Sätze repräsentiert und Aktionen als logische Beschreibungen von Vorbedingungen und Wirkungen. Dadurch ist es möglich, direkte Verbindungen zwischen Zuständen und Aktionen herzustellen.
2. Der Planer kann neue Aktionen zu seinem Plan hinzufügen wann immer dies erforderlich ist, nicht nur zu Beginn der Planung vom Anfangszustand aus. Planung und Ausführung müssen nicht streng getrennt sein. Indem offensichtliche oder wichtige Entscheidungen zuerst getroffen werden, werden der Verzweigungsfaktor und die Notwendigkeit zum Rücksetzen stark reduziert.
3. Die meisten Teile der Welt sind unabhängig von den meisten anderen Teilen. Deshalb kann man oft ein Ziel als Konjunktion mehrerer voneinander unabhängiger Teilziele formulieren und mit einer Divide-and-Conquer-Strategie lösen.

3.3. Grundlegende Repräsentationen für das Planen

3.3.1. Repräsentation von Zuständen und Zielen

In der STRIPS-Sprache (STanford Research Institute Problem Solver) werden Zustände durch Konjunktionen (Und-Verknüpfungen) sehr einfacher logischer Ausdrücke repräsentiert. Diese bestehen nur aus Prädikaten, angewandt auf Konstanten. Man nennt sie *Literale*.

Zustandsbeschreibungen müssen nicht vollständig sein. In einer unzugänglichen Welt hat ein Planer oft unvollständige Zustandsbeschreibungen. Eine solche entspricht einer Menge möglicher Zustände, für die der Planer erfolgreiche Pläne zu erstellen versucht. Viele Planungssysteme folgen der „Closed World Assumption“, nach der ein nicht erwähntes positives Literal als falsch angenommen wird.

Ziele werden ebenfalls durch Konjunktionen von Literalen repräsentiert. Diese Literale können auch Variable enthalten.

3.3.2. Repräsentation von Aktionen

Die STRIPS-Repräsentation für Aktionen besteht aus drei Komponenten:

- **Aktionsbeschreibung**: Eine Beschreibung dessen, was der Agent an die Umgebung ausgibt um etwas zu tun. Innerhalb des Planers dient sie nur als Name für eine mögliche Aktion.
- **Vorbedingung**: Eine Konjunktion von Atomen (positive Literale), die angibt, was gelten muss bevor der Operator angewandt werden kann.
- **Wirkung**: Eine Konjunktion von Literalen (positiv oder negativ), die beschreibt, wie sich die Situation verändert, wenn der Operator angewandt wird.

Ein Beispiel für die Syntax zur Formulierung eines STRIPS-Operators ist der folgende Operator für die Bewegung von einem Platz zu einem anderen:

$$Op(\text{ACTION}: Gehezu(dort), \text{PRECOND}: An(hier) \wedge Pfad(hier, dort), \text{EFFECT}: An(dort) \wedge \neg An(hier))$$

Abbildung 3.1 zeigt eine graphische Notation des Operators.

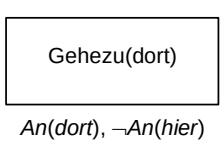

Abbildung 3.1

Ein Operator mit Variablen wird als **Operatorschema** bezeichnet, denn er beschreibt nicht nur eine einzige Aktion, sondern eine Menge von Aktionen, die sich durch Einsetzen verschiedener Konstanten für die Variablen als **Instanzen** des Schemas ergeben. Üblicherweise können nur Instanzen von Operatorschemata ausgeführt werden. Die Planungsalgorithmen sorgen dafür, dass alle Variablen in einem Operatorschema vor der Anwendung gebunden sind.

Ein Operator *o* ist **anwendbar** in einem Zustand *s*, wenn die Variablen in *o* so instanziiert werden können, dass alle Vorbedingungen von *o* erfüllt sind in *s*, d.h. wenn $Precond(o) \subseteq s$. Im resultierenden Zustand gelten alle positiven Literale in *Effect*(*o*), ebenso alle Literale, die schon in *s* gegolten

haben, außer denen, die negativ in *Effect*(*o*) sind. Ein Beispiel: In der Anfangssituation gelten die Literale

$$An(ZuHause),\ Pfad(ZuHause, Supermarkt),\ ...$$

Dann ist die Aktion *Gehezu*(*Supermarkt*) anwendbar und die Ergebnissituation enthält die Literale

$$\neg An(ZuHause),\ An(Supermarkt),\ Pfad(ZuHause, Supermarkt),\ ...$$

3.3.3. Repräsentation von Plänen

Nach dem **Least Commitment**-Prinzip beim Planen werden Entscheidungen, die im Planungsprozess zu treffen sind, so lange wie möglich zurückgestellt und erst dann getroffen, wenn dies unumgänglich ist. Ein Plan, in dem die Menge der Schritte nur teilweise geordnet ist, heißt **partiell geordnet**. Ist die Menge der Schritte vollständig geordnet, dann heißt der Plan **vollständig geordnet**. Entsteht ein vollständig geordneter Plan aus einem Plan *P* durch Hinzufügen von Constraints, dann heißt er eine **Linearisierung** von *P*. Ein Plan, in dem jede Variable an eine Konstante gebunden ist, heißt **voll instanziierter Plan**.

Ein Plan ist eine Datenstruktur bestehend aus vier Komponenten:

- Eine Menge von Planschritten. Jeder Schritt ist einer der Operatoren für das Problem.
- Eine Menge von Constraints, die die Schritte ordnen. Die Constraints haben die Form $S_i \prec S_j$, gelesen „S_i vor S_j“, was bedeutet, dass Schritt S_i irgendwann vor Schritt S_j kommen muss, aber nicht notwendigerweise unmittelbar vor S_j.
- Eine Menge von Constraints für Variablenbindungen. Die Constraints haben die Form $v = x$, wobei v eine Variable ist, die in einem Schritt vorkommt, und x eine Konstante oder eine andere Variable.
- Eine Menge **kausaler Kanten**. Eine kausale Kante wird geschrieben $S_i \xrightarrow{c} S_j$ und gelesen als „S_i erreicht c für S_j“. Kausale Kanten beschreiben den Zweck von Schritten in einem Plan. Bei der Kante $S_i \xrightarrow{c} S_j$ ist der Zweck von S_i die Vorbedingung c von S_j zu erfüllen.

Der Anfangsplan, der vor Anwendung einer Operation gegeben ist, beschreibt das ungelöste Problem. Er besteht nur aus zwei Schritten, genannt *Start* und *Ende*, und dem Ordnungsconstraint $Start \prec Ende$. Beide Schritte enthalten keine Aktion. Der *Start*-Schritt hat keine Vorbedingungen, aber eine Wirkung, nämlich die alle Aussagen hinzuzufügen, die im Anfangszustand wahr sind. Der *Ende*-Schritt hat den Zielzustand als Vorbedingung, aber keine Wirkung. Der Planungsprozess beginnt mit dem Anfangsplan und führt so lange Verfeinerungsschritte durch, bis ein vollständiger Plan entsteht.

Als Beispiel wird ein Anfangsplan für das Schuhe anziehen betrachtet. Er hat folgende Gestalt:

Plan(STEPS: { S_1: *Op*(ACTION: *Start*),
S_2: *Op*(ACTION: *Ende*, PRECOND: *RechterSchuhAn* $\wedge$ *LinkerSchuhAn*)},
ORDERINGS: {$S_1 \prec S_2$},
BINDINGS: {},
LINKS: {})

Abbildung 3.2(a) zeigt eine graphische Notation für Pläne, Abbildung 3.2(b) zeigt den Anfangsplan für das Schuhe-Anziehen-Problem.

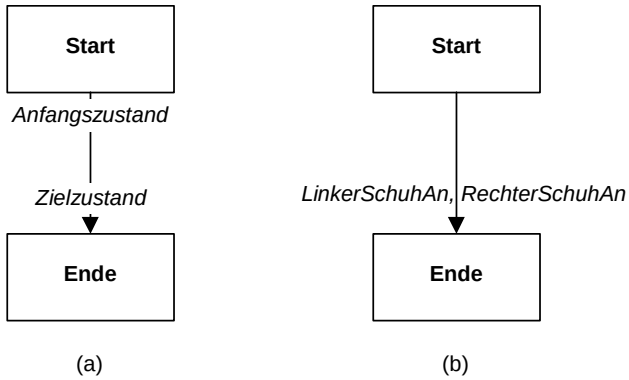

Abbildung 3.2

3.3.4. Lösungen

Eine Lösung ist ein Plan, den ein Planer ausführen kann und der das Erreichen des Ziels garantiert. Vollständig instanziierte, vollständig geordnete Pläne erfüllen diese Bedingung. Trotzdem ist es nicht sinnvoll nur solche Pläne zuzulassen aus drei Gründen:

1. Für viele Probleme, z.B. das Schuhe-Anziehen-Problem, ist es natürlicher, den partiell geordneten Plan als Lösung zu akzeptieren als eine beliebige Linearisierung auszuwählen.
2. Manche Planer können Aktionen parallel zueinander ausführen, für sie sind Pläne, die dies erlauben, vorteilhafter.
3. Partiell geordnete Pläne können leichter mit anderen Plänen zu größeren Plänen zusammengebaut werden.

Deshalb wird eine Lösung so definiert: Eine **Lösung** ist ein **vollständiger**, **konsistenter** Plan. Diese beiden Begriffe werden wie folgt definiert.

Ein Plan ist **vollständig**, wenn jede Vorbedingung jedes Schritts durch einen anderen Schritt erreicht wird. Ein Schritt **erreicht** eine Bedingung, wenn diese eine der Wirkungen des Schritts ist und kein anderer Schritt die Bedingung eventuell ungültig machen kann. Formaler gefasst:

Ein Schritt S_i erreicht eine Vorbedingung c eines Schritts S_j, wenn gilt:

(1) $S_i \prec S_j$ und $c \in \text{EFFECTS}(S_i)$
(2) Es gibt keinen Schritt S_k so dass $\neg c \in \text{EFFECTS}(S_k)$, wobei $S_i \prec S_k \prec S_j$ in irgendeiner Linearisierung des Plans gilt.

Ein Plan ist **konsistent**, wenn er keine Widersprüche in der Ordnung der Schritte und der Variablenbindung enthält. Ein Widerspruch liegt vor, wenn für zwei Schritte S_i und S_j sowohl $S_i \prec S_j$ als auch $S_j \prec S_i$ gilt oder wenn für eine Variable v und zwei verschiedene Konstanten A und B sowohl $v = A$ als auch $v = B$ gilt. Da sowohl $\prec$ als auch = transitive Relationen sind, ist z.B. auch ein Plan mit $S_1 \prec S_2$, $S_2 \prec S_3$ und $S_3 \prec S_1$ inkonsistent.

3.4. Ein Beispiel für partiell geordnetes Planen

Das betrachtete Beispiel ist das Milch-Bananen-Mix-Problem. Es werden zwei vereinfachende Annahmen gemacht: Die *Gehezu*-Aktion kann für Bewegungen zwischen zwei beliebigen Orten verwendet werden und bei der Definition der *Kaufe*-Aktion wird das Geld nicht betrachtet. Als Abkürzungen werden verwendet: *HWG* für *Haushaltswarengeschäft* und *SM* für *Supermarkt*. Der Anfangszustand und der Zielzustand sind wie folgt definiert:

Op(ACTION: *Start*, EFFECT: *An*(*ZuHause*) ∧ *Verkauft*(*HWG*, *Rührgerät*)
∧ *Verkauft*(*SM*, *Milch*) ∧ *Verkauft*(*SM*, *Bananen*))

Op(ACTION: *Ende*, PRECOND: *An*(*ZuHause*) ∧ *Hat*(*Rührgerät*)
∧ *Hat*(*Milch*) ∧ *Hat*(*Bananen*))

Die Aktionen *Gehezu* und *Kaufe* sind wie folgt definiert:

Op(ACTION: *Gehezu*(*dort*), PRECOND: *An*(*hier*), EFFECT: *An*(*dort*) ∧ ¬*An*(*hier*))

Op(ACTION: *Kaufe*(*x*), PRECOND: *An*(*geschäft*) ∧ *Verkauft*(*geschäft*, *x*), EFFECT: *Hat*(*x*))

Abbildung 3.3 zeigt den Anfangsplan.

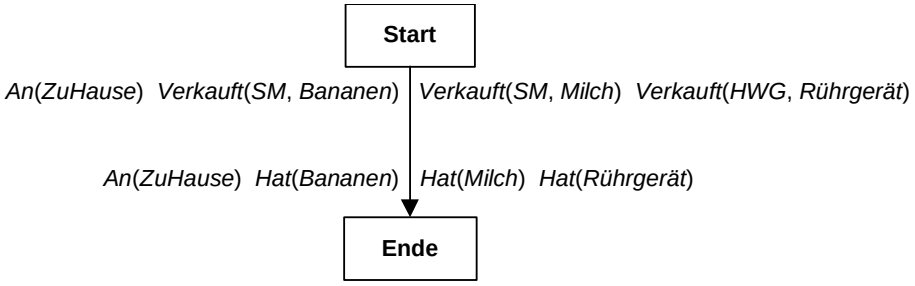

Abbildung 3.3

Abbildung 3.5 zeigt den vollständigen Plan mit einer Anordnung der Schritte, die den Ordnungsconstraints entspricht. Der Plan ist fast vollständig geordnet, nur die Reihenfolge der Aktionen *Kaufe*(*Milch*) und *Kaufe*(*Bananen*) ist nicht festgelegt, da es dafür keine Vorgaben in der Definition der Aktionen gibt.

3.5. Ein Algorithmus für partiell geordnetes Planen

Der folgende Algorithmus ist ein Planer für partiell geordnetes Planen, genannt POP (Partial-Order Planner). Er ist nichtdeterministisch, wie man an den Operationen *Choose* und *Select* erkennen kann.

function POP(*initial*, *goal*, *operators*) **returns** *plan*

plan ← MAKE-MINIMAL-PLAN(*initial*, *goal*)
loop do
  if SOLUTION?(*plan*) **then return** *plan*
  S_{need}, *c* ← SELECT-SUBGOAL(*plan*)
  CHOOSE-OPERATOR(*plan*, *operators*, S_{need}, *c*)
  RESOLVE-THREATS(*plan*)
end

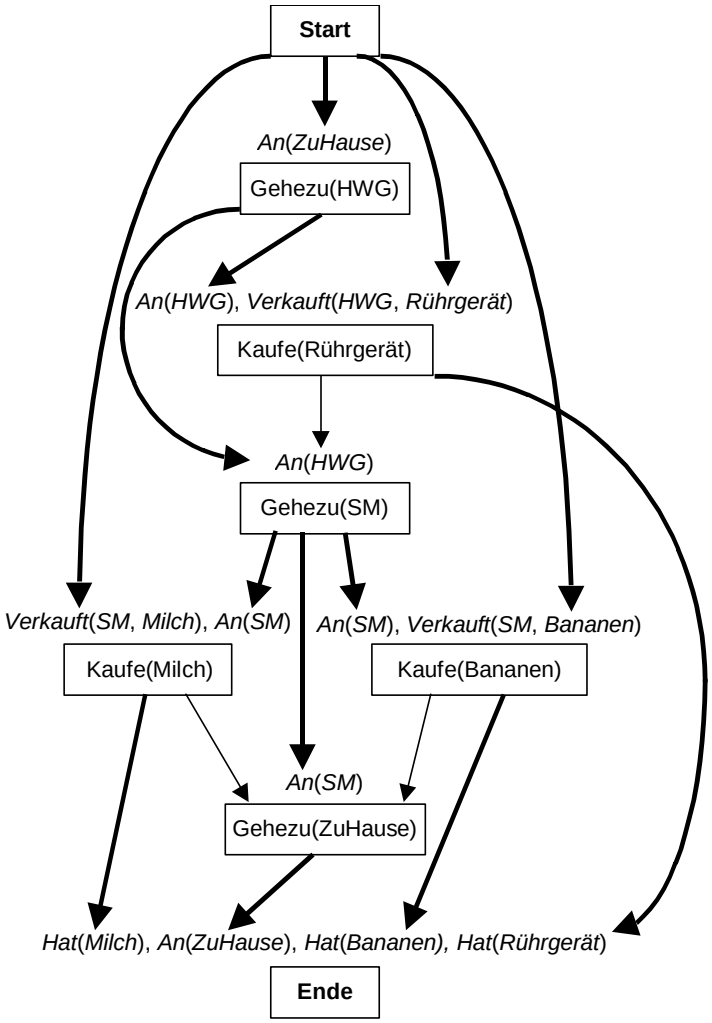

Abbildung 3.5

3.6. Vorgehensweise beim Planen am Beispiel der Blockwelt

Kurz zusammengefasst sind beim Knowledge Engineering für das Planen folgende Schritte durchzuführen:

- Lege den Gegenstandsbereich fest.
- Lege das Vokabular für Bedingungen (Literale), Operatoren und Objekte fest.
- Definiere Operatoren für den Anwendungsbereich.
- Definiere eine Beschreibung des speziellen Problems.
- Übergib die Problemformulierungen dem Planer und erhalte Pläne zurück.

Bis auf den letzten gelten diese Schritte für beliebige Anwendungsbereiche. Sie werden im Folgenden an zwei Anwendungsbereichen demonstriert.

Gegenstandsbereich: Der Bereich besteht aus einer Menge von Spielzeugblöcken, die auf einem Tisch liegen und einem Roboterarm, der die Blöcke ergreifen und bewegen kann. Die Blöcke können gestapelt sein, aber auf jeden Block passt höchstens ein anderer. Der Arm kann immer nur einen Block ergreifen, er kann ihn auf den Tisch oder auf einen anderen Block setzen. Das Ziel ist, Stapel bestimmter Blöcke zu bilden, z.B. Block A auf B oder C auf D und D auf E.

Vokabular: Die Objekte sind die Blöcke und der Tisch. Sie werden durch Konstanten repräsentiert. *On*(*b*, *x*) bezeichnet, dass Block *b* auf *x* liegt, das ein anderer Block oder der Tisch sein kann. Der Operator für die Bewegung eines Blocks ist *Move*(*b*, *x*, *y*). Dies bedeutet, dass Block *b* von der Position auf *x* zu der Position auf *y* bewegt wird. Eine der Vorbedingungen für die Bewegung eines Blocks ist, dass nichts auf ihm liegt. Dies wird durch die Formulierung *Clear*(*x*) ausgedrückt, die besagt, dass nichts auf x liegt.

Operatoren: Die formale Definition der Bewegungsoperation ist folgendermaßen:

Op(ACTION: *Move*(*b*, *x*, *y*),
PRECOND: $On(b, x) \wedge Clear(b) \wedge Clear(y)$,
EFFECT: $On(b, y) \wedge Clear(x) \wedge \neg On(b, x) \wedge \neg Clear(y)$)

Der Operator für die Bewegung eines Blocks auf den Tisch ist folgendermaßen definiert:

Op(ACTION: *MoveToTable*(*b*, *x*),
PRECOND: $On(b, x) \wedge Clear(b)$,
EFFECT: $On(b, Table) \wedge Clear(x) \wedge \neg On(b, x)$)

Clear(*x*) wird interpretiert als „es gibt einen freien Platz auf x für einen Block“. Unter dieser Interpretation ist *Clear*(*Table*) immer Bestandteil der Anfangssituation und es ist klar, dass *Move*(*b*, *Table*, *y*) die Wirkung *Clear*(*Table*) hat.