Heuristische Suchverfahren und ihre Anwendbarkeit auf physikalische Probleme

Besondere Lernleistung
von

Michael Goerz

Privates Gymnasium der Zisterzienserabtei Marienstatt

03.07.2002

Betreuung: Heike Niepel

Ich versichere hiermit, diese Arbeit selbstständig und ohne Zuhilfenahme anderer als der angegebenen Quellen und Hilfsmittel angefertigt zu haben.

Zusammenfassung

Diese Arbeit befasst sich erstens mit sogenannten heuristischen Algorithmen und zweitens mit ihrer Anwendbarkeit auf physikalische Probleme. Heuristische Algorithmen sind Näherungsalgorithmen, die zum Einsatz kommen, wenn ein Problem v.a. aufgrund seiner exponentiell anwachsender Komplexität nicht mit herkömmlichen Verfahren gelöst werden kann.
Im ersten Kapitel der Arbeit werden die Begriffe „Algorithmus“ und „Komplexität“ der theoretischen Informatik erklärt. Es werden die Komplexitätsklassen N – für polynomiale – und NP – für exponentielle Probleme eingeführt. Zudem wird das Prinzip heuristischer Verfahren erklärt. Kapitel II führt in die formale Modellierung von Problemen ein, die jedem Versuch, das Problem algorithmisch zu lösen, vorangehen muss. Es wird ein System eingeführt, das Probleme auf der Basis von Zuständen und Transformationen beschreibt. Zudem werden in dem Kapitel wichtige Begriffe der Graphentheorie erklärt. Die beiden folgenden Kapitel beschäftigen sich mit verschiedenen Algorithmen, sowohl systematisch-vollständigen, als auch heuristischen. Es werden die Verfahren Tiefensuche, Breitensuche, Zufällige Suche, Hillclimbing, Steepest-Ascent-Hillclimbing, Simulated Annealing, BF, BF*, Z, Z* und A* vorgestellt. Kapitel V beschäftigt sich mit der Frage, wie heuristische Zustands-Bewertungsfunktionen zustande kommen. Das Prinzip der Relaxation wird eingeführt und am Beispiel der Ungleichung von Tschebyscheff erklärt. Die letzten drei Kapitel der Arbeit legen Ansätze dar, wie heuristische Algorithmen in der Physik angewendet werden können. Es wird darauf eingegangen, physikalische Systeme nach dem in Kapitel II eingeführten Modell zu beschreiben und zu lösen; grundsätzliche Analogien zwischen physikalischen Theorien und den heuristischen Verfahren werden aufgezeigt, insbesondere am Fall des Simulated Annealing. Es wird beschrieben, wie Theoreme mit Hilfe der vorgestellten Algorithmen automatisch bewiesen werden können. Zuletzt wird die Verwendung numerischer Verfahren mit den heuristischen Algorithmen in Verbindung gebracht. Als Beispiel wird das Problem, in einem statischen System von 10 gravitationserzeugenden Körpern einen feldfreien Punkt zu finden, gelöst.

Inhalt

Vorwort

Grundlage für diese Arbeit ist mein starkes Interesse an Informatik und insbesondere für das Gebiet der künstlichen Intelligenz. Es fällt jedoch schwer, aus diesem Bereich ein geeignetes Thema für eine Besondere Lernleistung zu finden: der überwiegende Teil der Zusammenhänge würde den Rahmen erheblich sprengen. Dennoch habe ich mich bemüht, einen Komplex auszuwählen, der sich für die Ausarbeitung und Darstellung im vorliegenden Rahmen eignet. Bei der Themensuche ist mir die Diplomarbeit „Moderne Aspekte der Wissensverarbeitung“ von Gerald Reif in die Hände gefallen. Das Kapitel 4 dieser Arbeit, „Problemlösen durch Suchen“ hat in mir erstes Interesse geweckt. Ich habe dann mit Hilfe verschiedener Materialien die Möglichkeiten dieses Themenkreises abgesteckt. Die Bibliothek der State University of New York at Binghamton, zu der ich während einer USA-Reise in den Sommerferien 2001 längere Zeit Zugang hatte, ermöglichte mir die Einsicht in eine große Menge erstklassiger Fachliteratur, u. a. in das Buch „Heuristics: Intelligent Search Strategies for Computer Problem Solving“ von Judea Pearl, welches ein wesentliches Fundament meiner Arbeit bildet.

Da ich anfangs auch Interesse gehabt hatte, eine Arbeit aus dem physikalischen Fachbereich zu schreiben, beschloss ich, beides zu kombinieren und nicht nur die Thematik der heuristischen Verfahren auszuarbeiten, sondern auch ihre Möglichkeiten in der Physik zu untersuchen.

Obwohl das von mir gewählte Thema eher zu den einfacheren aus der Künstliche-Intelligenz-Forschung gehört, stellte es für mich eine Herausforderung dar, mit der Flut an neuen Begriffen, Ideen und Themenbereichen fertig zu werden. Insbesondere in der theoretischen Informatik mussten weite Strecken zurückgelegt werden, um eine Einordnung zu ermöglichen. Es galt, viele Zusammenhänge auf ein verständliches Niveau zu bringen und untereinander in Beziehung zu setzen, wobei ein Großteil der verwendeten Literatur nur in Englisch vorlag. Insbesondere die Übertragung des informatischen Teils auf den physikalischen war nicht einfach, da sich hierzu kaum Primärquellen finden ließen. In der Physik kam zudem hinzu, dass viele der Anwendungsbereiche heuristischer Verfahren weit außerhalb jedes gymnasialen Horizontes liegen. Diese Arbeit konnte daher in diesem Bereich oft nur Ansätze aufbauen.

Eine große Hilfe zum Verständnis einiger Begriffe und Zusammenhänge war das Buch „Das Geheimnis des kürzesten Weges“ von Peter Gritzmann und René Brandenberg, das speziell für Schüler geschrieben wurde und sehr gut auch recht komplizierte Thematiken erklärt.

Sehr bedanken möchte ich mich bei Frau Niepel, der Betreuerin dieser Arbeit, die immer für Fragen offen war und mir viel Freiraum für die eigene Ausarbeitung gelassen hat.

I) Einführung: Algorithmen und Komplexität; Heuristik

Der wohl zentralste Begriff der Informatik ist der des Algorithmus. Ein Algorithmus ist eine mathematisch genaue Anleitung zur Lösung eines Problems, die festgelegten Regeln folgen muss. Dabei versucht man, möglichst effiziente Verfahren zu finden..

Komplexität

Wie kann man die Qualität eines Algorithmus bewerten? Mit dieser Frage beschäftigt sich die theoretische Informatik (genauer: die Algorithmik). Mit dem Begriff „Komplexität" bezeichnet sie den Aufwand, d.h. die maximale Anzahl der nötigen Schritte eines Algorithmus in Abhängigkeit von der Anzahl der Eingabedaten (n). Derartige Betrachtungen sind insbesondere dann wichtig, wenn Algorithmen mit einer sehr großen Menge an Daten arbeiten, wie dies z.B. bei Such- und Sortieralgorithmen der Fall ist. Man teilt verschiedene Algorithmen gemäß ihrer Komplexität auch in Klassen (Ordnungen) ein, wobei man Koeffizienten weglässt. Bei den Sortierverfahren hat „Bubblesort" beispielsweise die Ordnung $O(n) = n^2$, der Algorithmus „Mergesort" die Ordnung $O(n) = n \cdot \log_2(n)$.

P und NP

Die meisten Algorithmen haben polynomialen Aufwand. Das heißt, dass sie im ungünstigsten Fall von der Ordnung $\mathrm{O}(n) = n^k \; ; k \in \mathbb{N}$ sind.

Es ist jedoch auch denkbar, das ein Algorithmus exponentiellen Aufwand haben kann: $O(n) = k^n$.

Solche Algorithmen sind ebenso problematisch wie interessant. Sie unterscheiden sich fundamental von den Algorithmen mit polynomialer Laufzeit. Bei ihnen zeigt sich einen Effekt, den man als „kombinatorische Explosion" bezeichnet. Ein Beispiel:

Angenommen, ein Computer braucht zur Durchführung eines Schrittes des Algorithmus eine tausendstel Sekunde. Der Algorithmus löst das Problem mit dem Aufwand von 2^n. Hat man nun 5 Eingangsdaten, braucht das Programm $2^5 / 1000 = 0,032s$, gibt man ihm 10 Daten braucht es 1,024s, bei 20 schon 1048,57s (17,5min) und bei nur 40 Daten sind es unglaubliche 35 Jahre! In der Realität müssen selbstverständlich in der Regel wesentlich mehr als 40 Daten verarbeitet werden. Es ist leicht ersichtlich, dass hier ein Problem liegt: ein solcher Algorithmus ist für die Praxis völlig untauglich! Es gibt aber eine Vielzahl an wichtigen praktischen Problemen, für die sich nur ein Algorithmus mit exponentieller Laufzeit

finden lässt. Das wohl bekannteste Problem ist das des Handelreisenden, der alle Großstädte der USA auf dem kürzesten Weg durchreisen muss.

Weil sich Probleme, deren Algorithmen polynomiale bzw. exponentielle Laufzeiten besitzen in ihrer Schwierigkeit so stark unterscheiden, und weil diese Unterscheidung so wichtig für die theoretische Informatik ist, definiert man sie als zwei elementare Problemklasse: N und NP.

„[P ist] die Klasse aller Entscheidungsprobleme, die in polynomialer Laufzeit durch deterministische Algorithmen gelöst werden können. Diese deterministischen Algorithmen haben die Eigenschaft, dass es nach jeder Anweisung genau eine weitere gibt, die als nächste ausgeführt wird. Nach der letzten Anweisung endet der Algorithmus

Die Klasse NP definieren wir mit Hilfe von nichtdeterministischen Algorithmen. Diese Algorithmen unterscheiden sich von deterministischen dadurch, dass sie mehrfache Kopien von sich selbst erzeugen können, die verschiedene Alternativen gleichzeitig verfolgen. [...] Wenn irgendeine Kopie eine Lösung findet, meldet sie Erfolg, und das Entscheidungsproblem ist gelöst. Wenn keine der Kopien eine Lösung findet, ist das Entscheidungsproblem negativ zu beantworten. Der Algorithmus endet, wenn alle Kopien abgearbeitet sind oder wenn eine der Kopien eine Lösung gefunden hat. [...] Die Klasse NP enthält nun alle Entscheidungsprobleme, die sich mit *nichtdeterministischen* Algorithmen in *polynomialer* Zeit lösen lassen“ (Gasper, S.240)

Die Definition baut auf einem abstrakten Automatenmodell auf. Ein PC kann selbstverständlich keine nichtdeterministischen Algorithmen ausführen, auch wenn gewisse Methoden benutzt werden können, um derartige Vorgänge bis zu einem gewissen Grad zu simulieren. Um ein Problem der Klasse NP zu lösen braucht ein PC immer exponentielle Laufzeit.

Vorsicht: Es gibt viele Probleme, die bei der Vorgehensweise „Ausprobieren aller Möglichkeiten“ einen exponentiellen Algorithmus benötigen. Allerdings lässt sich mit einigen Überlegungen oft ein polynomialer Algorithmus finden. Solche Probleme gehören nicht zu der Klasse NP. Es ist übrigens nicht bewiesen, dass sich für alle die Probleme, die man heute der Klasse NP zurechnet, vielleicht nicht doch auch ein Algorithmus mit polynomialer Laufzeit finden lässt, auch wenn dies niemand glaubt (Man nennt dies die P=NP - Frage)

Möglichkeiten, an NP-Probleme heranzugehen

Es ist also nicht möglich, NP-Probleme zu lösen, ohne sich nicht in der kombinatorischen Explosion zu verfangen. Dennoch muss man diese Probleme nicht unbeantwortet lassen. Es gibt nämlich Methoden, mit denen man dennoch versuchen kann, Lösungen zu entwickeln.

Sie beruhen auf einem naheliegenden Prinzip: bei der Definition der NP-Probleme wurde gesagt, dass die Algorithmen, die diese in polynomialer Zeit lösen können, sich selbst teilen und gleichzeitig verschiedene Alternativen zur Lösung betrachten müssen. Dies ist in der Praxis nicht möglich, ein Computer kann immer nur einen bzw. eine sehr begrenzte Menge an Befehlen gleichzeitig ausführen. Man könnte aber, immer wenn es zur Teilung kommt, nur den einen Zweig ausführen, der am wahrscheinlichsten zu einer Lösung führt. Dieses Vorgehen nennt man heuristisch. Auf diese Art und Weise wird selten eine optimale Lösung gefunden, jedoch oft eine hinreichend gute.

Was sind Heuristiken?

Heuristiken sind „Algorithmen, die meistens, aber nicht immer funktionieren oder eine ungefähre Lösung ausgeben“ (vgl. NIST, „heuristic“). Diese Vorgehensweise ist dem menschlichen Denken sehr nahe. Sie gehört deshalb auch in den Forschungsbereich der künstlichen Intelligenz. Beispielsweise stelle man sich einen Schachspieler vor, der in einer bestimmten Spielsituation seinen nächsten Zug planen muss. Um eine Strategie zu entwickeln, die ihn sicher ans Ziel führt (d.h., seinen Gegner schachmatt setzt), müsste er eigentlich für jeden möglichen Zug jede mögliche Reaktion seines Gegners und auf jede dieser Reaktion wieder alle Zugmöglichkeiten seinerseits usw. in seine Planung einbeziehen. Das führt sofort zur kombinatorischen Explosion. Besser wäre es, mehrere heuristische Funktionen zu entwickeln, die einem Zug eine bestimmte Qualität zuordnen. Eine Idee für eine solche Funktion wäre beispielsweise die Zahl nicht gedeckten eigenen Figuren. Ein Zug, der die Zahl der ungedeckten Figuren vergrößert, ist in der Regel schlechter als einer, der dies nicht tut. Mit Hilfe komplexer Funktionen könnte man mit relativ geringem Rechenaufwand schon einen möglichst guten Zug finden. Allerdings lässt sich mit dieser Methode nicht mehr garantieren, dass der Zug zu einem Sieg führt. Der menschlicher Spieler verwendet ähnliche Methoden, wenn auch nicht auf diese mathematische Art und Weise und meist mehr oder weniger unbewusst. Zumindest als ungeübter Schachspieler würde er sich zu einem nicht geringen Teil auf seine „Intuition“ verlassen, er würde den Zug wählen, der zu einer scheinbar stärkeren Position auf dem Spielfeld führt. Dieses Konzept der Intuition ist nicht sehr weit von der Heuristik entfernt.

Heuristiken helfen also bei Problemen, die ein exponentielles Wachstum an Möglichkeiten zeigen, sodass die Kapazität oder der Rechenaufwand die Anwendbarkeit klassischer Algorithmen übersteigen, indem sie den Bereich der zu betrachtenden Möglichkeiten einschränken.

II) Modellierung von Problemen

Bevor ein Problem algorithmisch angegangen werden kann, muss es erst schematisiert werden. Die Wahl eines geeigneten Modells bietet vor allem den Vorteil, dass gleiche Algorithmen auf ganz unterschiedliche Probleme angewandt werden können, die jedoch mit dem selben Modell beschrieben werden können. Bei NP-Problemen (aber auch bei sehr vielen anderen) lässt sich gut mit Graphen und Bäumen arbeiten.

Graphen

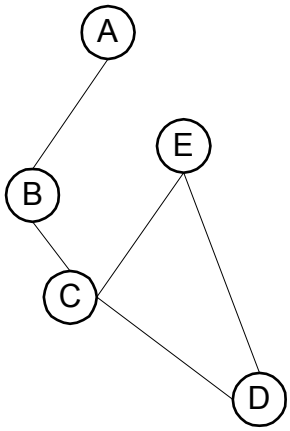

Abb. 1 ein ungerichteter Graph

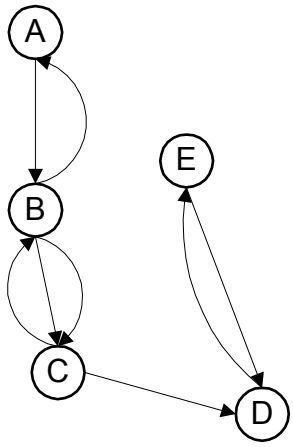

Abb. 2 ein gerichteter Graph

Graphen sind Strukturen, die aus Knoten und Kanten bestehen. Abb. 1 zeigt einen Graphen. Die Knoten sind mit den Buchstaben A bis E beschriftet, die Kanten sind die geraden Linien, die zwischen den Knoten verlaufen. Mathematisch wird ein Graph folgendermaßen beschrieben:

„Graph $G = (V, E)$
G besteht aus
V, einer endlichen Menge von Knoten, sowie aus
E, einer Menge 2-elementiger Teilmengen von V, den Kanten.“
(Gritzmann, S.24)

Graphen können zusammenhängen, müssen aber nicht. Zwischen zwei Knoten können auch mehrere Kanten verlaufen. Den Graphen bezeichnet man in diesem Fall als Multigraphen. Die Kanten können gerichtet sein (sie heißen dann Bögen), außerdem kann man einer Kante

ein Kantengewicht (eine beliebige, meistens natürliche Zahl) zuordnen. Ein gerichteter Graph, dessen Bögen Bogengewichte zugeordnet sind, nennt man Digraph (engl. „*di*rected *graph*“). Abb. 2 zeigt einen gerichteten Multigraphen.

Bäume

Bäume sind eine besondere Art von Graphen. Sie sind zusammenhängend und enthalten keine Kreise (Abb. 3). Kreisfrei ist ein Graph dann, wenn es nicht möglich ist, einen Weg durch den Graphen zu finden, der zweimal durch einen Knoten geht, ohne eine Kante mehrmals zu benutzen. Bäume können auch gerichtet sein und Kantengewichte besitzen (also Digraphen sein, Abb. 4). Logischerweise kann ein Multigraph nie ein Baum sein, da er ja schon per Definition Kreise besitzt. Bei den gerichteten Bäumen benutzt man zudem folgende Begriffe, um bestimmte Verhältnisse der Knoten untereinander auszudrücken:
Den Anfangsknoten des Baums nennt man die Wurzel (root), jeder Knoten (außer der Wurzel) hat einen Vorgänger (parent), von dem aus ein Bogen zu ihm führt, der Knoten selbst ist in Bezug auf seinen Vorgänger dessen Nachfolger (child). Ein Endknoten des Baums, der keine Nachfolger mehr hat, heißt Blatt (leaf). Die Anzahl der Kanten ist übrigens immer um eins geringer als die Anzahl der Knoten.

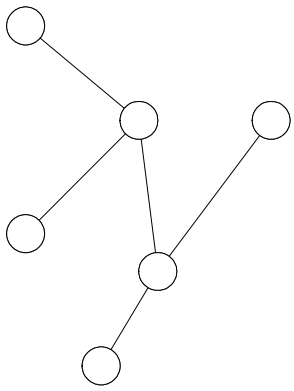

Abb. 3 ein ungerichteter Baum

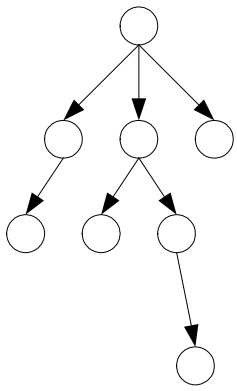

Abb. 4 ein gerichteter Baum

Bäume und NP-Probleme

Bäume spielen eine besondere Rolle bei der Modellierung von NP-Problemen, bzw. von Problemen, die mit Hilfe von Heuristiken gelöst werden sollen. Das sieht man schon an der Definition von NP: „Diese Algorithmen unterscheiden sich von deterministischen dadurch, dass sie mehrfache Kopien von sich selbst erzeugen können, die verschiedene Alternativen gleichzeitig verfolgen.“ Wenn man diese Vorgehensweise visualisiert, entsteht ein Baum, bei dem jeder Knoten eine Alternative im weiteren Verlauf der Problemlösung darstellt.

Formelle Problemlösung

Um ein Problem, insbesondere ein NP-Problem mit Hilfe von Computern und unter der Zuhilfenahme einer Baummodellierung zu lösen, benötigt man drei Elemente:

1) Ein Zustandscode (code)

   Der Code ist eine Datenstruktur, die es erlaubt, jeden möglichen Zustand (state) während der Suche nach einer Lösung zu beschreiben. Je nach Problem kann diese Datenstruktur sehr einfach oder auch sehr komplex sein Um die Baumstruktur widerzuspiegeln, muss der Code Informationen über den schon gelösten Teil wie auch über den noch zu lösenden Teil des Problems enthalten. Dazu muss ein Zustand immer auch Unterprobleme beschreiben, indem er auf irgendeine Art und Weise auf ihm untergeordnete Zustände verweist (seine Nachfolger im Graph). Der Zustand selbst stellt einen Knoten im Baum dar.

   Die Beschreibung eines Zustandes muss es weiterhin ermöglichen, Transformationen (siehe 2) auf ihn anzuwenden; gegebenenfalls muss sie auch Informationen für die Steuerungsstrategie (siehe 3) enthalten. Jedes Problem hat einen definierten Anfangszustand, und einen oder mehrere Endzustände, die einfach als solche identifiziert werden können, wenn sie angetroffen werden (dieser Endzustand selbst muss aber nicht die Lösung des Problems sein, manchmal besteht die Lösung auch aus dem Pfad zwischen Anfangszustand und Endzustand).

2) Ein Transformationssystem (Produktionssystem, production system)

   Es muss eine Menge von Operationen (production rules) geben, die einen Zustand in einen anderen, ihm untergeordneten, eventuell auch übergeordneten Zustand transformieren. Durch Anwendung aller möglichen Operationen auf einen Zustand lässt sich jeder seiner Nachfolger erzeugen. Die möglichen Transformationen bauen aus den Zuständen einen Baum auf. Eine Operation wird formell durch die Angabe einer Bedingung und einer zugehörigen Transformation angegeben („Wenn der Zustand diese oder jene Bedingung erfüllt, kann er auf diese oder jene Weise verändert werden“.)

   Die Bedingungen müssen ausreichend allgemein formuliert sein (z.B. beim Schachspiel die Regeln für die möglichen Züge der einzelnen Figuren: Läufer diagonal, Turm senkrecht und waagerecht, ...). Manchmal können nicht auf jeden Zustand alle Transformationen angewendet werden, deshalb muss das Transformationssystem in der Lage sein zu

entscheiden, inwieweit dies der Fall ist.

Es müssen außerdem Prozeduren im Transformationssystem vorhanden sein, die eine bestimmte Operation konkret auf die Datenstruktur eines Zustands anwenden..

3) Eine Steuerungsstrategie (controll strategy)

Es muss eine Methode geben zu entscheiden, wie und in welcher Reihenfolge die Transformationen angewandt werden, um einen gewünschten Zustand, die Lösung des Problems, zu erreichen. Erst die Steuerungsstrategie erzeugt aus den Zuständen einen Baum bzw. einen Pfad durch diesen Baum. Eine Steuerungsstrategie kann systematisch-vollständig sein oder, wie bei heuristischen Verfahren, selektiv; in diesem Fall müssen im Zustandscode Informationen enthalten sein, die die Steuerungsstrategie leiten. Die Steuerungsstrategie wird manchmal auch zum Transformationssystem gezählt.

Beispiel: Solitaire-Spiel

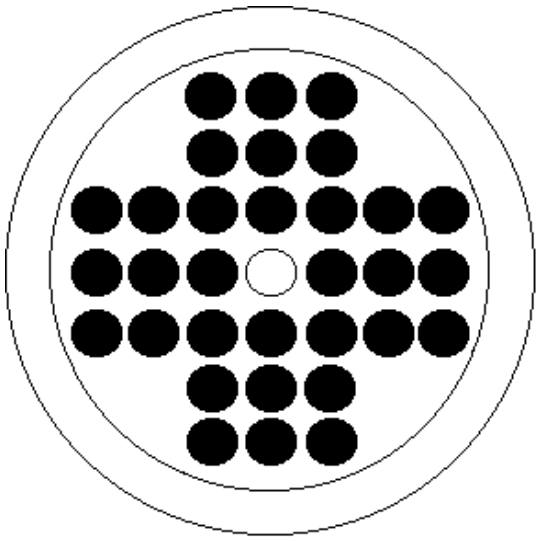

Abb. 5 das Solitaire-Brett im Anfangszustand

Bei dem bekannten Solitaire-Spiel liegen auf einem Spielbrett wie links dargestellt 32 Kugeln (schwarz), in der Mitte ist ein Feld freigelassen. Es wird gezogen , indem man waagerecht oder senkrecht mit einer Kugel über eine zweite Kugel auf ein freies Feld springt und die übersprungene Kugel entfernt. Ziel des Spiels ist es, am Ende möglichst wenig Kugeln übrig zu haben, gewonnen ist das Spiel, wenn am Ende nur noch eine Kugel in der Mitte liegt.

Für die Simulation. Dazu soll nun ein geeigneter (nicht unbedingt der für den Computer effizienteste) Code und ein Transformationssystem gefunden werden.

Es ist sinnvoll, eine vollständige Beschreibung der Brettposition als Zustand anzusehen. Als Datenstruktur lässt sich dies z.B. in einer Matrix (einem zweidimensionalen Array) darstellen, wobei besetzte Brett-Positionen mit einer 1, unbesetzte mit einer 0 markiert werden. Das Feld muss für die Matrix-Darstellung zu einem Quadrat erweitert werden, den hinzukommenden Positionen wird der Wert –1 zugewiesen, um sie als ungültig zu markieren. Der Code für den Anfangszustand lautet damit:

$$Z_0 = \begin{bmatrix} -1 & -1 & 1 & 1 & 1 & -1 & -1 \\ -1 & -1 & 1 & 1 & 1 & -1 & -1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ -1 & -1 & 1 & 1 & 1 & -1 & -1 \\ -1 & -1 & 1 & 1 & 1 & -1 & -1 \end{bmatrix}$$

Es ist leicht nachvollziehbar, dass dieser Code alle oben genannten Forderderungen erfüllt:
Die Anzahl der Nullen in der Matrix gibt Informationen darüber, bis zu welchen Grad das Problem gelöst ist, da ja bei jedem Zug ein Stein entfernt wird (der Matrix eine 0 hinzugefügt wird). Auf die gleiche Art und Weise gibt die Anzahl und die Positionen der 1 an, welcher Teil noch zu lösen ist. Mit Hilfe der notwendigen Operationen ist es möglich, Unterprobleme zu erzeugen (jeder Zustand kann als eigenes Problem angesehen werden: „Wie kommt man von diesem Zustand zu einem Endzustand?“; deshalb stellt jeder Nachfolger ein Unterproblem seines Vorgängers dar). Die Datenstruktur erlaubt es außerdem, auf einfache Art zu bestimmen, welche Züge durchgeführt werden können.

Auch die Menge der Operationen ist daher einfach zu beschreiben: hierzu müssen nur die schon in der Spielbeschreibung genannte Regel für das Ziehen formell richtig ausgedrückt werden:

$$O = \left\{ \begin{array}{l} \left(\begin{array}{l} ((Z(m,n)=1) \wedge (\mathrm{Z}(m+1,n)=1) \wedge (\mathrm{Z}(m+2,n)=0)) \\ \qquad \rightarrow Z(m,n) := 0; Z(m+1,n) := 0; Z(m+2,n) := 1 \end{array} \right); \\ \left(\begin{array}{l} ((Z(m,n)=1) \wedge (\mathrm{Z}(m-1,n)=1) \wedge (\mathrm{Z}(m-2,n)=0)) \\ \qquad \rightarrow Z(m,n) := 0; Z(m-1,n) := 0; Z(m-2,n) := 1 \end{array} \right); \\ \left(\begin{array}{l} ((Z(m,n)=1) \wedge (\mathrm{Z}(m,n+1)=1) \wedge (\mathrm{Z}(m,n+2)=0)) \\ \qquad \rightarrow Z(m,n) := 0; Z(m,n+1) := 0; Z(m,n+2) := 1 \end{array} \right); \\ \left(\begin{array}{l} ((Z(m,n)=1) \wedge (\mathrm{Z}(m,n-1)=1) \wedge (\mathrm{Z}(m,n-2)=0)) \\ \qquad \rightarrow Z(m,n) := 0; Z(m,n-1) := 0; Z(m,n-2) := 1 \end{array} \right) \end{array} \right\}$$

$Z(m,n)$ steht dabei für das Element der Matrix, das sich in Spalte m und Zeile n befindet.
Die Menge beschreibt den Sprung eines Spielsteins rechts, links, unten und oben.

Wendet man diese Operationen der Reihe nach an, erhält man folgenden Baum:

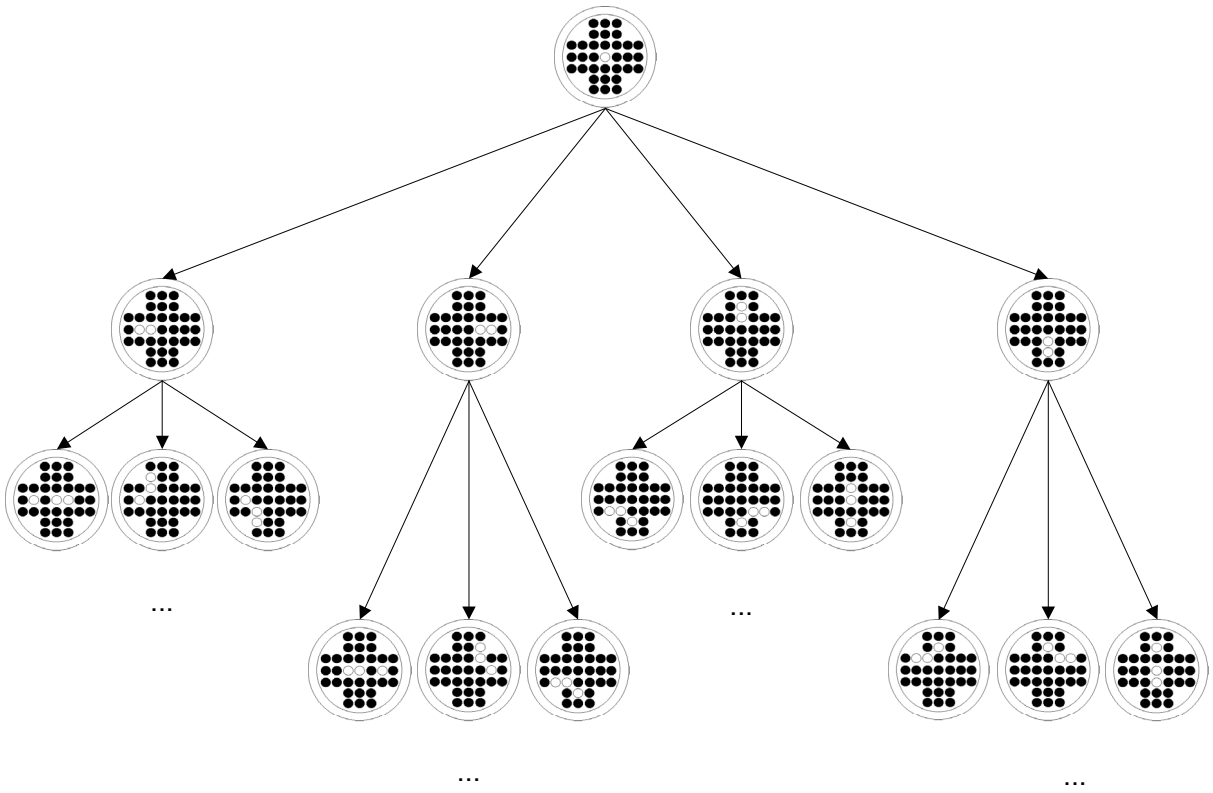

Abb. 6 der Zustandsbaum des Solitaire-Spiels

Die Zustände, die hier durch graphische Abbildung des Spielbrettes dargestellt sind, müssten natürlich durch den entsprechenden Code ersetzt werden.

III) Uninformierte Steuerungsstrategien

Um heuristischen Steuerungsstrategien, die noch zusätzliche Informationen benutzen, besser verstehen zu können, sollten zunächst zwei elementare Verfahren – die Tiefen- und die Breitensuche – vorgestellt werden, die keine derartigen Informationen verwenden. Diese sind grundlegend für jedes Suchen in Baumstrukturen. An dieser Stelle müssen zwei weitere Begriffe eingeführt werden: Expansion und Exploration.
Ein Knoten wird dann als exploriert bezeichnet, wenn einer seiner Nachfolger generiert worden ist. Er ist expandiert, wenn alle seine Nachfolger generiert sind.

Tiefensuche (Depth-First) und Backtracking

Bei der Tiefensuche werden Knoten in tieferen Schichten bevorzugt. Ausgehend von der Wurzel wird je ein Knoten expandiert und einer der erzeugten Nachfolger betrachtet (rekursiv). Dieser Vorgang wiederholt sich bis entweder eine Lösung gefunden ist oder das Verfahren an einem Blatt angelangt ist bzw. aus irgendeinem anderen Grund keine Nachfolger mehr generiert werden können. In diesem Fall wird zum Vorgänger des aktuellen Knotens zurückgegangen und mit einem noch nicht betrachteten Nachfolger weitergemacht. Wenn bereits alle Nachfolger betrachtet wurden, geht das Verfahren einen weiteren Schritt zurück.

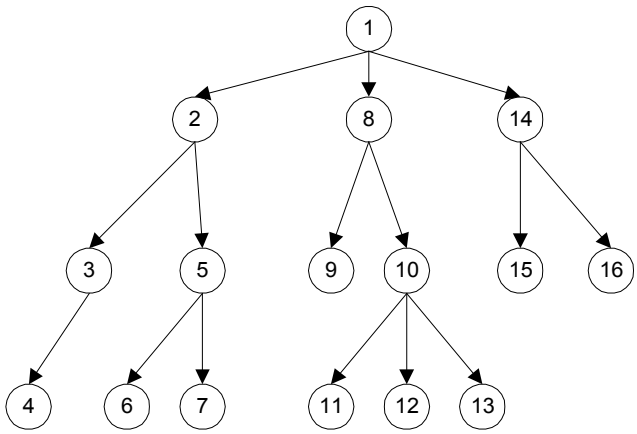

Abb. 7 Tiefensuche, die Knoten sind in der Reihenfolge ihrer Betrachtung nummeriert

In diesem Baum sind die Knoten in der Reihenfolge nummeriert, in der die Tiefensuche sie bearbeiten würde. Knoten mit gemeinsamen Vorgängern könnten jedoch auch in anderer Reihenfolge betrachtet werden (Der Knoten 8 könnte beispielsweise auch als zweites betrachtet werden, der Knoten 2 dann als achtes.)

Eine Variante der Tiefensuche ist das Backtracking, bei dem ein Knoten, der gerade betrachtet wird, nicht expandiert, sondern nur exploriert. Es wird also nur einer der Nachfolger generiert. Dieser wird dann als nächstes betrachtet. Es wird auch immer nur ein generierter Nachfolger gespeichert: wenn ein Limit erreicht wird und das Verfahren zu einem schon explorierten Knoten zurückkehrt, um ihn erneut zu explorieren, wird der vorher generierte Nachfolger dieses Knotens (und dessen sämtliche Nachfolger) aus dem Speicher gelöscht. Das Backtracking spart also enorm Speicherplatz, da nicht so viele Knoten im Speicher gehalten werden müssen. Dafür muss aber eine Information gespeichert werden, welche Operation angewendet wurde, um einen Knoten zu generieren, damit beim Zurückgehen derselbe Knoten nicht noch einmal betrachtet wird.
Es ist auch möglich, die Tiefensuche mit einem Tiefenlimit durchzuführen, bei dem der Algorithmus nur bis zu dieser bestimmten Tiefe sucht und sich dann erst einmal zurückzieht. Beim Backtracking ist dies nicht mehr so einfach möglich, da ja Pfade, auf denen zurückgegangen wird, gelöscht werden und also später keine Knoten mehr betrachtet werden können, die tiefer als das Tiefenlimit liegen.
Das Hauptproblem der Tiefensuche liegt darin, dass oft sehr tief in den Baum vorgedrungen wird und sich erst sehr spät herausstellt, dass der gewählte Pfad in eine Sackgasse führt.

Breitensuche (Breadth-First)

Die Breitensuche ist praktisch das Gegenteil der Tiefensuche. Hier wird Knoten, die sich in der gleichen Ebene befinden, der Vorzug gegeben. Die Knoten eines Baums würden damit in der folgenden Reihenfolge bearbeitet:

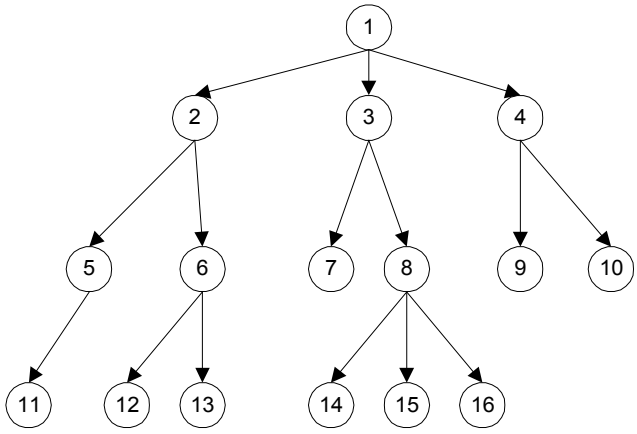

Abb. 8 Breitensuche, die Knoten sind in der Reihenfolge ihrer Betrachtung nummeriert

Knoten werden expandiert, die generierten Nachfolger werden jedoch nicht sofort betrachtet, sondern auf eine Warteliste gesetzt, solange nicht alle Knoten mit einem gemeinsamen Vorgänger expandiert sind. Erst dann wird mit dem ersten Knoten auf der Warteliste weitergemacht. Die Breitensuche eignet sich besonders dann, wenn die Knoten durchschnittlich viele Nachfolger haben, sodass es wahrscheinlich ist, schon in geringer Tiefe eine Lösung zu finden; allerdings müssen immer alle schon generierten Knoten im Speicher gehalten werden, der Speicherbedarf ist also sehr groß, es gibt keine Möglichkeit, ihn zu verringern (im Gegensatz zur Tiefensuche, bei der dies durch Backtracking geschieht).

Zufällige Suche

Bei der zufälligen Suche wird immer ein zufällig ausgewählter Knoten, von dem noch keine Nachfolger erzeugt wurden, expandiert. Dieses Verfahren springt zwischen den Knoten des Suchbaumes, so wie dies auch die Suchverfahren der Best-First-Familie tun werden (diese springen jedoch gezielt).

IV) Informierte Steuerungsstrategien

Informierte Steuerungsstrategien benutzen zusätzliche Informationen, um „bessere“ Knoten zuerst zu betrachten. Um zu entscheiden, ob ein Zustand besser als ein anderer ist, benutzt man eine Bewertungsfunktion $f(s)$. Bei NP-Problemen ist diese Funktion heuristisch, also eine ungefähre Abschätzung.

Als Information zur Grundlage der Bewertungsfunktion werden dabei oft Kantengewichte eingeführt, die die „Kosten“ für Erzeugung eines Zustandes angeben. Es entsteht dann ein Optimierungsproblem; Ziel ist es nun, einen möglichst kostenarmen Weg zwischen dem Anfangszustand und einem Endzustand zu ermitteln. Es gibt eine Kostenfunktion $c(s)$, die für einen Zustand die Mindestkosten ausgibt, die auf dem Weg vom Anfangszustand bis zu diesem Zustand entstehen. Um eine Bewertungsfunktion aufzustellen braucht man aber noch eine Abschätzung der minimalen Restkosten bis zum Erreichen eines Endzustandes, gegeben durch die Funktion $h(s)$. Die Bewertungsfunktion $f(s)$ berechnet sich dann als $c(s)+h(s)$; $h(s)$ liefert die heuristische Komponente der Funktion. Je niedriger $f(s)$ ist, desto besser ist der Knoten.

Es gibt aber auch andere Möglichkeiten, $f(s)$ zu evaluieren, die nur auf einer Analyse des in Frage stehenden Zustandes beruhen. Ein Beispiel dafür sind die meisten Schachprogramme. Diese weisen der Bewertungsfunktion oft einen Wert in sog. Bauerneinheiten zu, der die relative Stärke der Figurenanordnung beider Parteien aus der Sicht von Weiß beschreibt. Zum Beispiel sagt der Wert -1 aus, dass Schwarz einen Vorteil hat, der ungefähr so groß ist, als hätte es einen Bauern mehr als Weiß. Der Wert 0 beschreibt eine ausgewogene Situation.

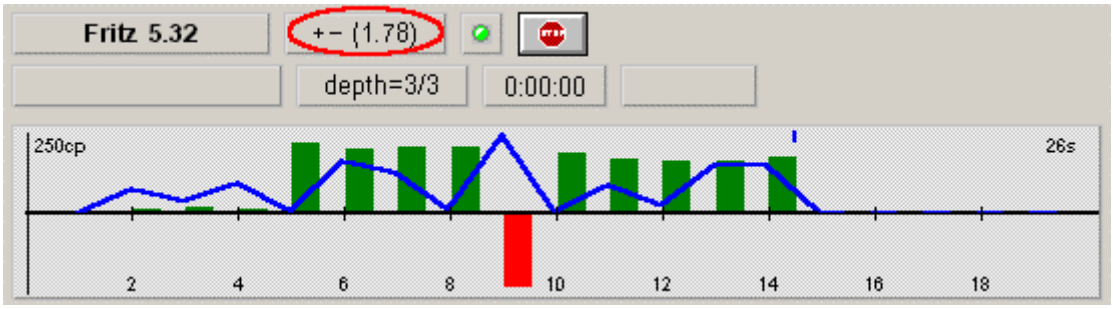

Abb. 9 das Bewertungsfenster der Schachsoftware Fritz 5, die Stellungsbewertung ist rot eingekreist

Hillclimbing, Steepest Ascent und Simulated Annealing

Das Hillclimbing-Verfahren besteht darin, immer "einen Schritt in die richtige Richtung" zu machen.

Ausgehend vom Startknoten wird je ein Knoten exploriert. Ist der generierte Nachfolger besser als seine Vorgänger (hat er eine bessere Bewertungsfunktion), wird er als nächstes betrachtet. Ist er nicht besser, wird ein weiterer Nachfolger des Knotens generiert und ggf. als nächstes betrachtet, bis eine Lösung gefunden ist oder das Verfahren in eine Sackgasse gerät, weil entweder keine Nachfolger mehr zu generieren sind oder weil keiner der Nachfolger eines Knotens besser ist als der Knoten selbst (man sich etwa in einem lokalen Minimum befindet). Wenn dies geschieht, muss die Suche abgebrochen werden und ein ganz neuer Versuch gestartet werden. Beim Hillclimbing lassen sich die Schritte nicht rückgängig machen, worin auch dessen größter Schwachpunkt liegt. Besonders gut verwenden lässt sich das Hillclimbing, wenn bei dem Problem jeder Zustand aus jedem anderen (mit Zwischenschritten) erzeugen lässt, also keine Sackgassen dadurch entstehen können, dass keine Nachfolger mehr generiert werden können. Ein Beispiel für ein solches Problem ist das Umformen von Gleichungen bzw. der Beweis von mathematischen Theoremen.

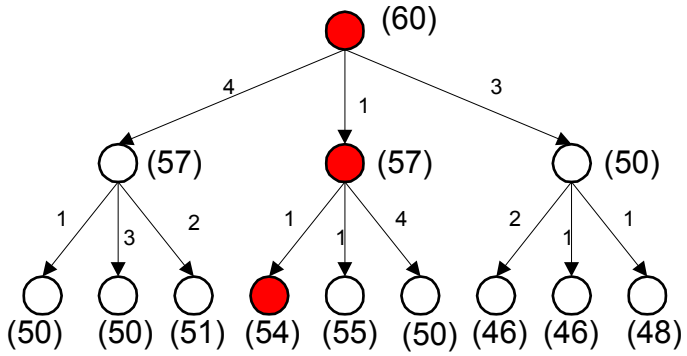

Abb. 10 das einfache Hillclimbing, in Klammern sind die heuristischen Bewertungen der Zustände angegeben

Bei der Variante Steepest Ascent wird jeder Knoten ganz expandiert und unter seinen Nachfolgern der beste ausgewählt, sofern er besser als sein Vorgänger ist. Steepest Ascent beschleunigt das Hillclimbing etwas und ist deshalb in der Regel vorzuziehen. Wenn die Menge der möglichen Transformationen und damit der Aufwand der Expansion sehr groß ist, ist manchmal auch das einfache Hillclimbing vorteilhafter.

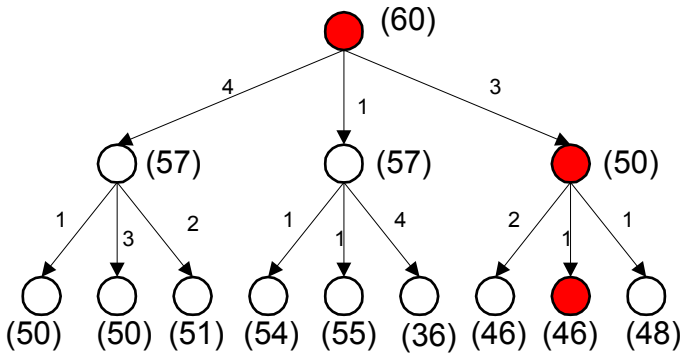

Abb. 11 das Steepest-Ascent-Hillclimbing, in Klammern sind die heuristischen Bewertungen der Zustände angegeben

Man kann versuchen, mit verschiedenen Tricks wie z.B. durch eine Kombination mit dem Backtracking versuchen, dem Problem der lokalen Maxima zu begegnen, dennoch ist das Backtracking bei vielen Problemen nicht sehr erfolgreich, bei einigen wenigen jedoch sehr.
Eine weitere Variante, die versucht, auf eben dieses Problem der lokalen Maxima einzugehen ist das Simulated Annealing. Sie erlaubt es auch, dass ein Zustand als nächstes betrachtet wird, der schlechter als sein Vorgänger. Ein solcher Zustand wird mit einer bestimmten Wahrscheinlichkeit akzeptiert, die sich nach der Formel

$$p = e^{-\frac{f(s_1)-f(s_2)}{T}}$$

berechnet, wobei e die Eulersche Zahl ist und $f(s_1)$ bzw. $f(s_2)$ der Wert des Vorgänger-Zustandes bzw. des neu generierten Zustandes. T ist ein Wert, der zu Beginn der Suche willkürlich festgesetzt werden kann. Im Laufe der Zeit bzw. der voranschreitenden Lösung des Problems wird T immer kleiner. Wann und um wie viel T verkleinert wird, kann ebenfalls willkürlich festgesetzt werden. T bewirkt durch seine Veränderung während des Suchvorgangs, dass, während es am Anfang noch relativ wahrscheinlich ist, dass ein schlechterer Zustand akzeptiert wird, dies später immer unwahrscheinlicher wird.
Der Erfolg des Simulated Annealing hängt stark davon ab, welcher Anfangswert für T gewählt wurde und nach welcher Regel T abnimmt. Die optimalen Parameter sind sehr schwer zu ermitteln, in der Regel muss man durch Ausprobieren geeignete Werte finden.
Es ist sehr sinnvoll, das Simulated Annealing mit dem Steepest Ascent zu kombinieren. Die Funktion wird dann in absteigender Reihenfolge vom besten bis zum schlechtesten Nachfolger angewendet. Dies vermeidet, dass ein Zustand weiterbetrachtet wird, der schlechter ist als sein Vorgänger, obwohl ein besserer vorhanden ist.

Best-First-Algorithmen

Während das Hillclimbing im Prinzip eine Tiefensuche darstellt, die durch die Bewertungsfunktion gelenkt wird, sind die Verfahren der Best-First-Familie eher an die Breitensuche angelehnt; sie ähneln aber auch der zufälligen Suche, da sie den Suchbaum nicht kontinuierlich durchsuchen, sondern zwischen den bisher generierten Blättern (noch nicht expandierten Knoten) springen. Dieses Springen erfolgt aber nicht zufällig, sondern es wird immer der beste Knoten, gemessen an einer beliebigen Bewertungsfunktion, als nächstes betrachtet und dieser Knoten wird im Gegensatz zum Hillclimbing nicht nur aus den Nachfolgern des zuletzt betrachteten Zustands ausgewählt, sondern aus allen noch nicht explorierten Knoten. Dieser Grundalgorithmus BF hatte eine Reihe von Varianten, die u.a. die Namen BF*, Z, Z* und A* tragen.

Heuristische Verfahren liefern normalerweise nur Näherungslösungen. Durch eine geringfügige Änderung des BF-Grundalgorithmus lässt sich jedoch auch ein Algorithmus, BF*, erstellen, der eine optimale Lösung ausgibt, jedoch auf Kosten eines erheblich größeren Aufwandes (der Stern im Namen eines Algorithmus drückt immer aus, dass er optimale Lösungen liefert). Bei BF* wird die Suche noch weitergeführt, nachdem schon eine Lösung gefunden ist, und zwar solange, bis sich erweist, dass diese schon gefundene Lösung tatsächlich auch die optimale Lösung des Problems darstellt. Der Erweis wird durch die charakteristische Eigenschaft des BF-Algorithmus erbracht, als nächstes immer zum besten verfügbaren Zustand zu springen. Das heißt: angenommen der Algorithmus findet eine Lösung, bricht an dieser Stelle aber nicht ab. Es wird wieder unter allen noch nicht explorierten Knoten (zu denen man auch die schon gefundene Lösung zählt) gesucht und der Beste als nächstes betrachtet. Wenn noch ein offener Knoten existiert, dessen Bewertungsfunktion besser ist als die der schon gefunden Lösung, springt der Algorithmus dorthin und führt die Suche weiter durch. Vielleicht findet sich auf diesem Weg eine zweite Lösung. Aber: hat von den offenen Knoten keiner eine bessere Bewertungsfunktion als die schon gefundene Lösung, muss der Algorithmus wieder zu dieser Lösung zurückgehen. Dann ist eindeutig bewiesen, dass diese Lösung die beste sein muss (vorrausgesetzt, die verwendete Heuristik liefert eine angemessen gute Näherung), denn es ist unmöglich, auf anderem Weg eine bessere Lösung zu finden, da alle diese Wege eine schlechtere Bewertungsfunktion besitzen. Bei BF* wird der Abbruch der Suche also solange verzögert, bis eine schon gefundene Lösung expandiert werden müsste.

Die anderen Varianten von BF bestehen nur darin, die Bewertungsfunktion so einzuschränken, dass sich das Errechnen der Bewertungsfunktion mit weniger Aufwand gestaltet. Schon

in der Einleitung dieses Kapitels wurde erklärt, dass es sehr gebräuchlich ist, für die Bewertungsfunktion Kantengewichte als „Kosten“ einzuführen. Die Bewertungsfunktion wird auf irgendeine beliebige Art und Weise aus diesen Kantengewichten ermittelt. Unter Umständen kann dies jedoch sehr aufwendig sein. Zu bevorzugen wäre eine Vorgehensweise, die bei der Errechnung der Kosten eines Zustandes die Kosten seiner Vorgänger mit einbezieht. Eine solche rekursive Funktion – in der Informatik als Rollback-Funktion bekannt - würde Rechenaufwand sparen, da auf gespeicherte Zwischenwerte (die Kosten der Vorgänger) zurückgegriffen werden kann. Führt man BF mit einer solchen Rollbackfunktion durch, nennt man den Algorithmus Z (Z* wenn man BF* spezialisiert).
Es gibt noch eine weitere Spezialisierung von Z*, die man A* nennt. Diese wurde schon in der Einleitung erwähnt: es handelt sich um eine „additive“ Rollback-Funktion, die sich zusammensetzt aus der Summe der Kosten aller Vorgänger und einer heuristischen Abschätzung der Restkosten.

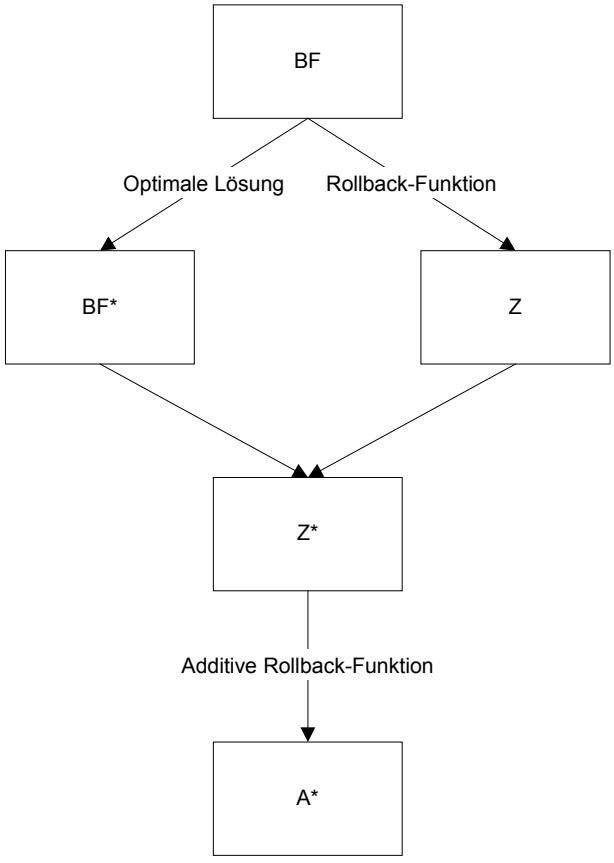

Abb. 12 Die Verwandtschaft der Best-First-Algorithmen (vgl. Pearl, S. 63)

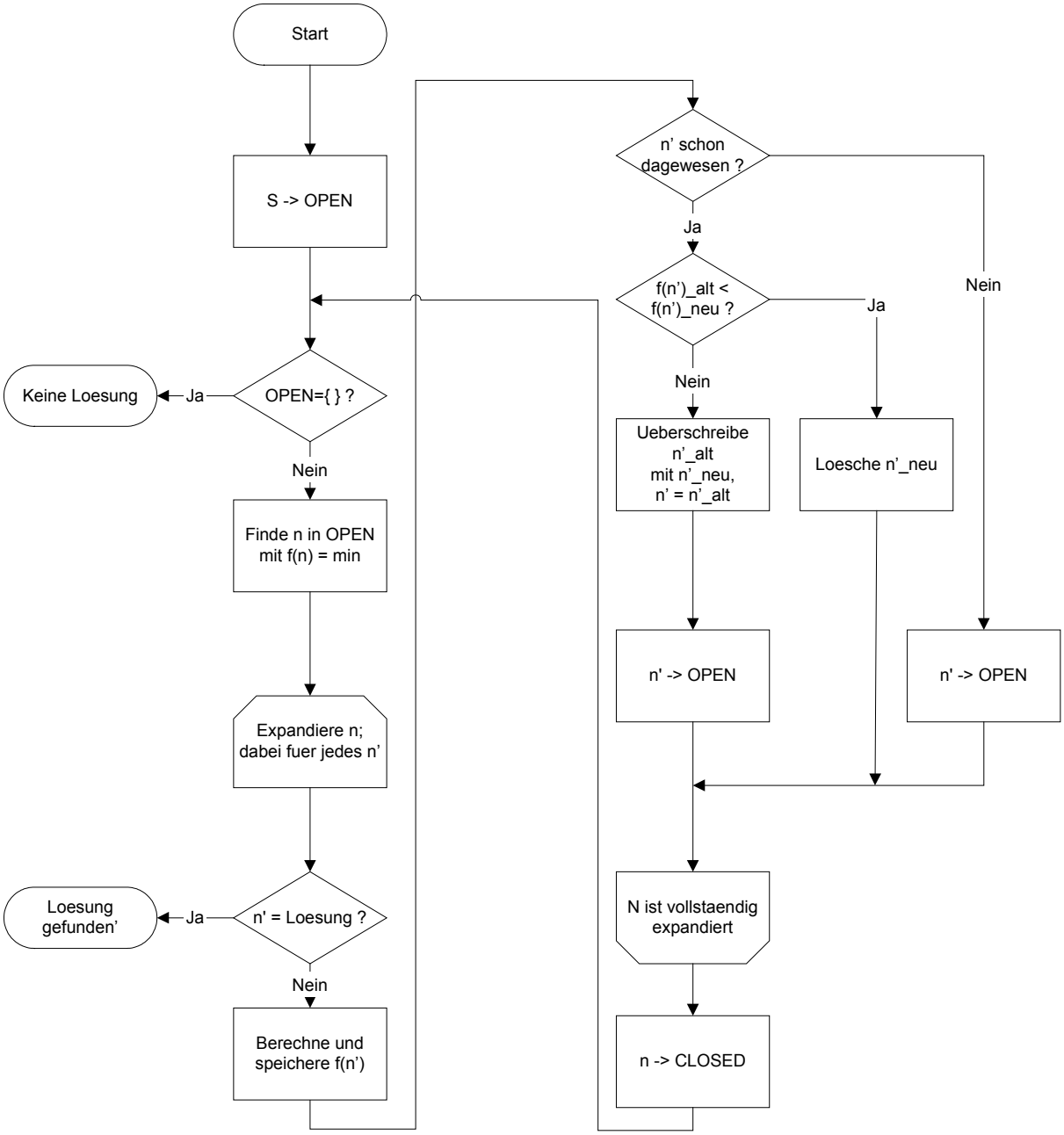

Abb. 13 Flowchart für den Best-First-Algorithmus

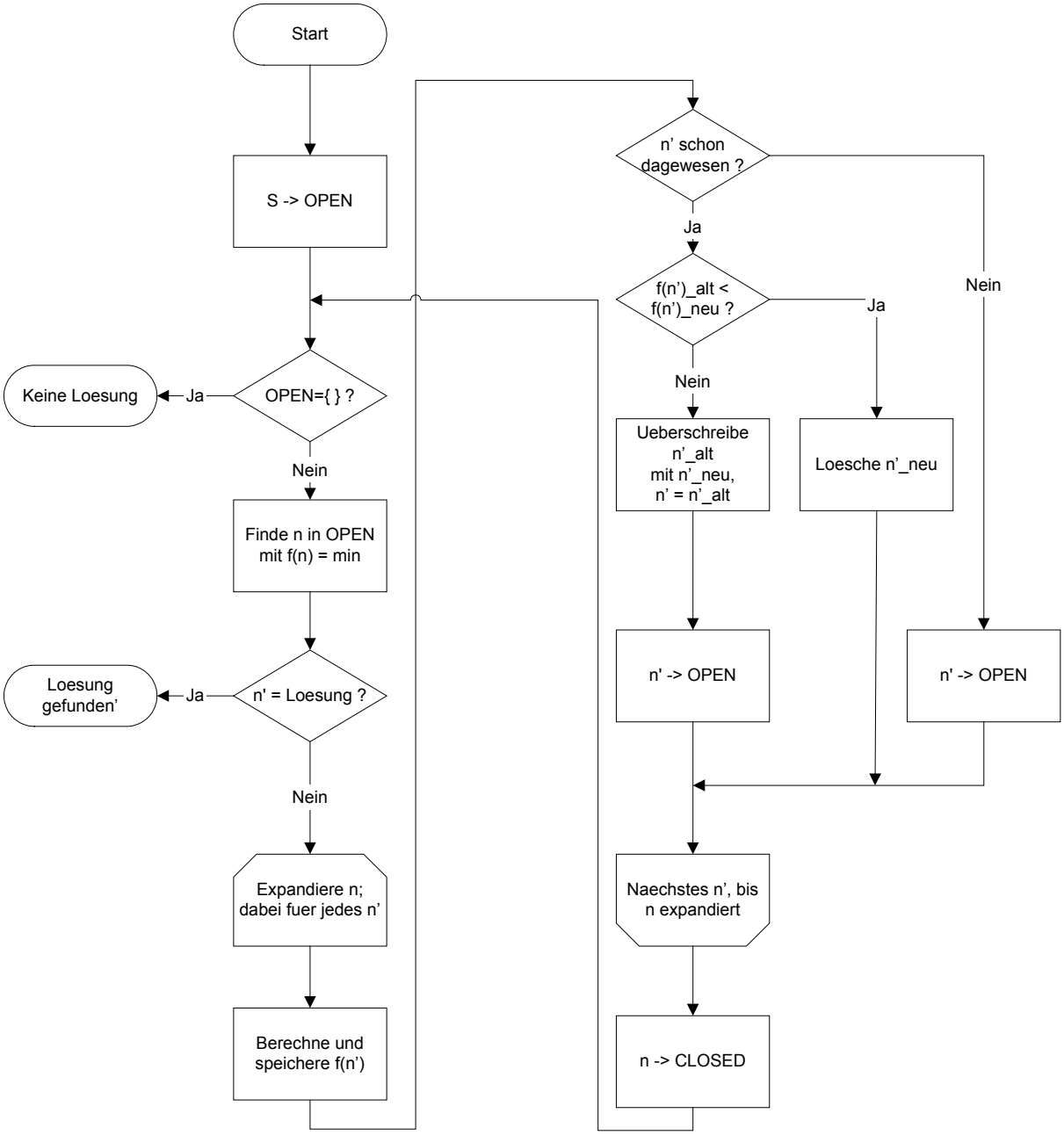

Abb. 14 Flowchart für den BF*-Algorithmus

Vergleich der Verfahren, Kombinierungsmöglichkeiten

Heuristische Lösungseinsätze müssen immer Kompromisse eingehen zwischen Qualität, Laufzeit und Ressourcen (Speicher). Die vorgestellten Verfahren legen ihre Schwerpunkte in diesen Bereichen verschieden. Hillclimbing und seine Varianten zeichnet sich durch eine geringe Laufzeit aus. Es steuert zielsicher auf ein Lösung zu. Nachteilig kann sich diese Zielstrebigkeit auf die Qualität auswirken. Hillclimbing-Verfahren geraten leicht in eine Sackgasse oder verpassen eine gute Lösung. Ganz im Gegensatz dazu steht stehen die Best-

First-Algorithmen, die mit Sicherheit eine Lösung finden (bei Bedarf sogar eine optimale), dies jedoch insbesondere mit einem sehr hohen Speicheraufwand bezahlen, denn es muss ja der gesamte bisher generierte Suchbaum im Speicher gehalten werden, um immer am besten Ende weitermachen zu können. Sucht man mit BF* nach einer optimalen Lösung, kann zudem die Laufzeit stark anwachsen. Das Backtracking-Prinzip, das bei den uninformierten Steuerungsstrategien vorgestellt wurde, findet ebenfalls mit Sicherheit eine Lösung und benötigt nur wenig Speicher (es muss immer nur der aktuelle Suchpfad im Speicher sein), dies muss jedoch mit langen Laufzeiten ausgeglichen werden.

In der Realität lassen sich oft von vorneherein Aussagen über mögliche Inhomogenitäten des Suchraums (die möglichen Zustände und ihre Struktur) machen. In solchen Fällen kann es sinnvoll sein, die obigen Verfahren und Prinzipien zu kombinieren. Derartige Kombinationen sind problemlos möglich. So könnte beispielsweise zu Beginn des Problems ein speichersparender Backtracking-Algorithmus schnell tiefer in den Suchbereich eindringen und dann Best First ab einer gewissen Tiefe eine umfassendere und genauere Suche durchführen. Viele derartige Kombinationen sind denkbar.

V) Die heuristische Abschätzungsfunktion

Kern der heuristisch informierten Verfahren ist die Abschätzung der Qualität eines Zustandes (oft durch eine Abschätzung der Restkosten) mittels einer Funktion. Diese Funktion ist bisher etwas obskur geblieben. Wie lässt sich eine solche Funktion finden, die Aussagen über nicht bekannte Eigenschaften von Zuständen machen kann?

Heuristiken – der kreative Teil der Problemlösung

Das Finden (Heuristik kommt von gr. heuriskein – finden) derartiger Funktionen ist in der Tat schwierig. Es benötigt tiefes Fachwissen über das zu lösende Problem. Oft ist es zudem so, dass es einfach zu findende Heuristiken für ein Problem gibt, die sich dann jedoch oft als zu ungenau erweisen, während wirklich gute Abschätzungen nur nach eingehender Analyse entstehen. Allgemein beruhen sie auf dem Ansatz, ein gelockertes Modell des Problems einzuführen, das einige Regeln des ursprünglichen Modells vernachlässigt oder vereinfacht, etwa die Produktionsregeln oder die Anforderungen an eine Lösung. Dabei ist darauf zu achten, dass in heuristischen Verfahren verwendete Heuristiken immer „optimistisch“ sein müssen, d. h. sie dürfen den wirklichen Wert nie überschätzen.

Die Graphentheorie, ein Teilgebiet der Mathematik, hat eine Reihe allgemeiner Modelle entworfen, die auf eine Vielzahl praktischer NP-Probleme anwendbar sind. Für diese Problemmodelle sind Heuristiken bekannt. Das wohl bekannteste dieser Probleme ist das Traveling-Salesman-Problem.

Das Traveling-Salesman-Problem (TSP)

„Das TSP ist wohl das berühmteste aller graphentheoretischen Problem; geradezu der Prototyp eines NP-schweren Problems. Viele der heute zur Verfügung stehenden algorithmischen Methoden sind ursprünglich am Rundreiseproblem entwickelt worden“ (Gritzmann, S. 284f.)

Das Problem befasst sich damit, wie ein Handelsreisender alle Punkte auf einer Karte besuchen kann und dabei möglichst wenig Weg zurücklegt. Dahinter steckt selbstverständlich ein abstrakteres Problem, nämlich in einem kompletten gewichteten Graphen (ein gewichteter Graph, dessen Knoten alle verbunden sind) einen Rundweg zu finden, der insgesamt möglichst wenig Kosten verursacht (Summe der Kantengewichte). Es gibt viele praktische Probleme, die mit diesem Modell beschrieben werden können. Beispiele sind Paketdienste, die ihre Lieferwagen möglichst kostensparend einsetzen müssen, Maschinen, die zur Her-

stellung von Schaltkreisen möglichst effizient Löcher in Platinen bohren müssen oder Telekommunikationsfirmen, die mit einem Minimum an Kabeln ein Ringnetz aufbauen müssen.

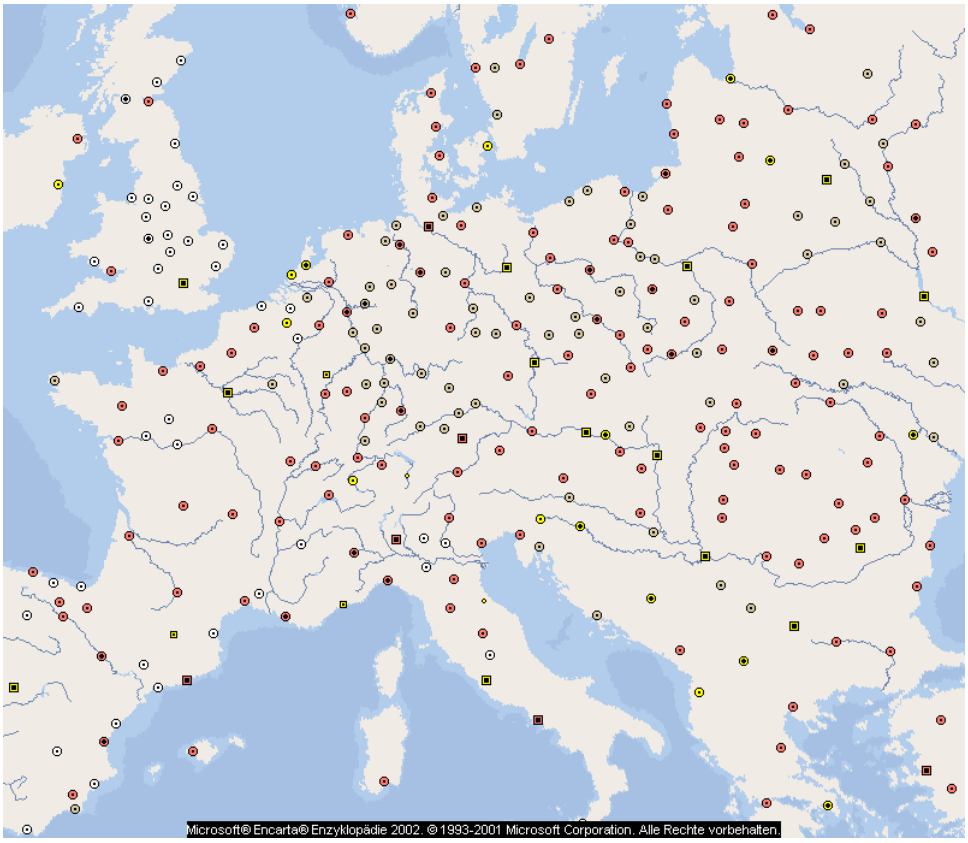

Abb. 15 Städte über 100 000 Einwohner in Mitteleuropa als Grundlage für eine TSP-Tour

Städte ab 100 000 Einwohner

Um das Problem systematisch zu lösen, müsste man ausgehend von dem Startpunkt alle möglichen Wege generieren, indem man parallel immer zu jedem noch nicht besuchten Knoten weitergeht. Der resultierende Algorithmus wäre damit von der Ordnung n!, also NP. Bei der Suche nach einer Heuristik, die diesen Aufwand verkürzen würde, ist die nahe liegendste Möglichkeit, immer zur nächsten Stadt weiterzureisen. Dies ist sicherlich eine funktionierende Heuristik, doch schon eine kurze Betrachtung zeigt, dass sie von mangelhafter Güte ist. Die Qualität des gesamten restlichen Weges wird nur auf der Grundlage der nächsten Station bemessen! Eine systematische Vorgehensweise führt da weiter:

Ausgegangen wird von einem vollständigen Graphen. Das Ziel ist ein Graph, der

a) insgesamt alle diese Knoten miteinander verbindet

b) eine Tour ergibt

Diese Bedingungen kann man nun lockern. Es erscheint sinnvoll, das Merkmal a) aufrecht zu erhalten. Eine Abschätzung der Kosten, alle Knoten eines Graphen zu besuchen, sollte auch alle Knoten mit einbeziehen (die erste, naive Nächste-Nachbar-Heuristik hat dies nicht beachtet). Dagegen kann leicht vernachlässigt werden, dass der gefundene Graph eine Tour ergeben soll. Die Restkosten könnten also sinnvoll durch die Gesamtkosten eines Graphen abgeschätzt werden, der alle restlichen Knoten verbindet, aber nicht unbedingt eine Tour ergibt. Um eine optimistische Abschätzung zu garantieren, sollte der Graph minimale Gesamtkosten haben. Auf diese Beschreibung passt in der Graphentheorie ein sogenannter minimaler 1-Baum („Eins-Baum“). Die Bezeichnung „Baum“ ist etwas irreführend, es handelt sich nämlich keineswegs um einen Baum. Stattdessen ist dieser Graph von einem Baum abgeleitet – dem minimalen Spannbaum (der Baum, der alle Knoten minimal verbindet). Der 1-Baum entsteht, indem man dem Spannbaum noch eine Kante hinzufügt, und zwar diejenige, die von einem definierten Anfangsknoten zu dem nächsten Knoten führt, der im Spannbaum noch nicht mit dem Anfangsknoten verbunden war. Die Tatsache, das der 1-Baum kein Baum, sondern allgemein ein Graph ist, ist insofern wichtig, als dass für die Heuristik ein Graph und nicht ein Baum konstruiert werden soll (Ebenso wie die Lösung des TSP ein Graph - genauer: eine Tour - und kein Baum ist). Ein 1-Baum lässt sich algorithmisch sehr einfach ermitteln.

Auf der Grundlage dieser Überlegungen lässt sich eine heuristische Funktion angeben, die unter Einbeziehung des schon ermittelten Pfades eine Bewertung für die nächsten Kandidaten ergibt: die Summe aller Kantengewichte eines minimalen 1-Baums, der sich noch konstruieren lässt, wenn man die Kanten der bisher ermittelten Teillösung als festgesetzt annimmt.

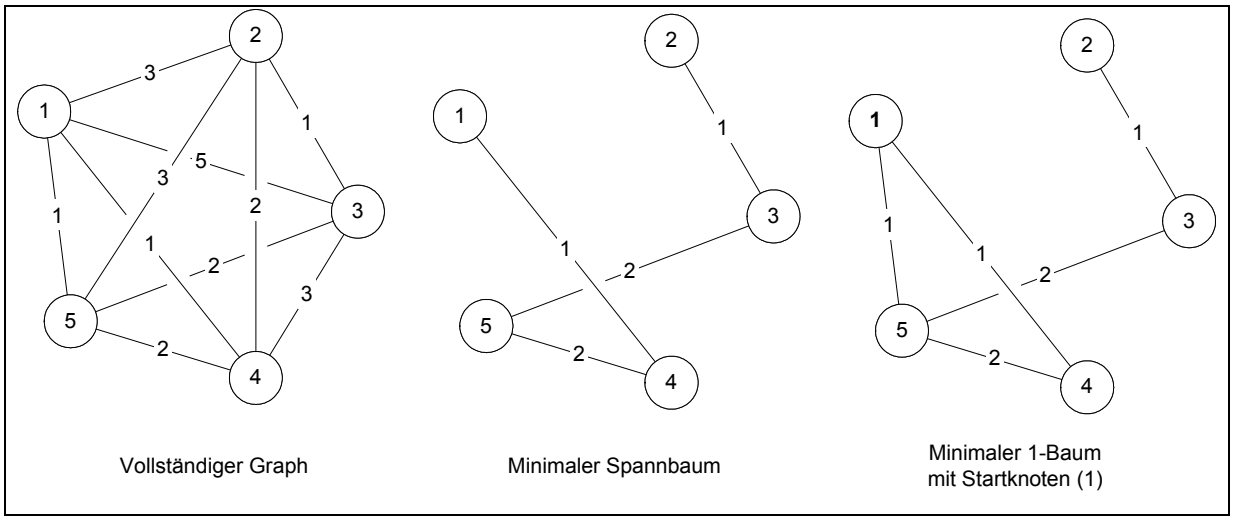

Abb. 16 Vollständiger Graph, Minimaler Spannbaum und Minimaler 1-Baum

Die Ungleichung von Tschebyscheff als Heuristik

Ein Paradebeispiel für eine optimistische Heuristik, das schon aus dem Mathematikunterricht bekannt ist, ist die Ungleichung von Tschebyscheff.

Diese Ungleichung schätzt die Wahrscheinlichkeit ab, dass in einer Wahrscheinlichkeitsverteilung die Zufallsvariable X um mehr als c von dem Erwartungswert μ abweicht. Mit σ ist die Standardabweichung der Verteilung bezeichnet.

Bei der Herleitung der Formel lassen sich beispielhaft einige Prinzipien von Heuristiken verdeutlichen:

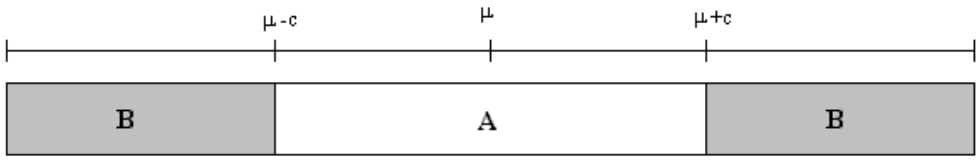

Abb. 17 eine Wahrscheinlichkeitsverteilung, unterteilt in die Bereiche A und B

Die Wertemenge von X wird in zwei Bereiche unterteilt, den Bereich A, der innerhalb des Bereiches liegen soll, der sich um maximal c von μ unterscheidet und den Bereich B (die restlichen Zufallswerte); Die Wahrscheinlichkeit von B soll abgeschätzt werden.

Dazu entwickelt man zunächst eine Abschätzung der Varianz, die man dann zu einer Abschätzung von P(B) umschreiben kann.

Die Varianz der Verteilung wird bekanntlich nach der Formel

$$\mathrm{VAR}(X) = \sigma^2 = \sum_{i=1}^{n} (x_i - \mu)^2 \cdot \mathrm{P}(X = x_i)^2$$

ermittelt.

Geschrieben für die geteilte Wertemenge lautet das (mit $X \in \mathrm{A} = x_1 \ldots x_k$ und $X \in \mathrm{B} = \mathrm{x}_{k+1} \ldots x_n$):

$$\sigma^2 = \sum_{i=1}^{k} (x_i - \mu)^2 \cdot \mathrm{P}(X = x_i) + \sum_{i=k+1}^{n} (x_i - \mu)^2 \cdot \mathrm{P}(X = x_i)$$

An dieser Stelle kann ein gelockertes Modell eingeführt werden: Der Anteil von A an der Varianz lässt man unter den Tisch fallen, wodurch eine (nicht optimistische!) Abschätzung entsteht:

$$\sigma^2 \geq \sum_{i=k+1}^{n} \left(x_i - \mu\right)^2 \cdot \mathrm{P}\left(X = x_i\right)$$

Es wird sofort noch einmal vereinfacht; aus der Definition der Menge B folgt:

$\left|x - \mu\right| \geq c$, für alle $x \in \mathrm{B}$,

setzt man dies ein und teilt anschließend durch c^2, erhält man

$$\frac{\sigma^2}{c^2} \geq \sum_{i=k+1}^{n} \mathrm{P}\left(X = x_i\right) = \mathrm{P}\left(\mathrm{B}\right),$$

also die Ungleichung von Tschebyscheff

$$\mathrm{P}\left(\mathrm{B}\right) \leq \frac{\sigma^2}{c^2}.$$

Als heuristische Funktion im Rahmen eines Algorithmus würde man schreiben:

$$\mathrm{h}\left(\mathrm{P}\left(\mathrm{B}\right)\right) = \frac{\sigma^2}{c^2}$$

Diese Heuristik erfüllt wie gefordert die Bedingung, optimistisch zu sein.

VI) Heuristiken und Physik 1 – Physik im algorithmischen Modell

Es lässt sich eine gewisse grundsätzliche Verbindung ziehen zwischen der Methode der Problemmodellierung, die in dieser Arbeit eingeführt wurde, und Modellen der Physik. Der Begriff des Zustands liegt auch in der Physik nahe, viele Zusammenhänge können definiert werden mit Hilfe eines Zustandsraumes – der Menge aller Zustände, die das System einnehmen kann – und geeigneten Produktionsregeln: „There is a close correspondence between physical processes and computations. [...] Theoretical models describe physical processes by computations that transform initial data according to algorithms representing physical laws“. (Wolfram) Da mit Hilfe dieses Prinzip physikalische Prozesse algorithmisch bearbeitet werden sollen, ist es oft nötig, bestimmte Größen (wie etwa die Zeit) zu diskretisieren, um eine endliche oder mindestens abzählbar unendliche Menge an möglichen Zuständen zu definieren.

Makroskopische Systeme

Die Produktionsregeln sind im makroskopischen Bereich oft linear. Beispielsweise lässt sich ein fahrender Wagen, der zum Zeitpunkt t_0 and der Stelle s_0 startet und sich mit der Beschleunigung $v = 2$ m/s fortbewegt, auch nach diesem Prinzip auffassen: Ein Zustand des Wagens lässt sich mit einem Paar [Position auf der Bewegungsgeraden (s); Zeitpunkt (t)] beschreiben, der Anfangszustand ist [0; 0]. Die Produktionsregel lautet:

$[s; t] \rightarrow [s + v \cdot t_d; t + t_d]$,

wobei t_d das diskrete Zeitintervall ist.

Auf den ersten Blick mag dies nach einem einfachen s-t-Diagramm aussehen, dies ist jedoch irreführend, vielmehr geht es hier um eine algorithmische Betrachtungsweise, die so in der Physik nicht direkt vorkommt.

Mikroskopische Systeme

Im mikroskopischen Bereich sind die Produktionsregeln oft nicht mehr linear. Hier muss oft in Wahrscheinlichkeiten gerechnet werden. Die Thermodynamik ist ein Beispiel dafür: Sie trifft stochastische Aussagen über das Verhalten einer Menge von Molekülen eines Stoffes. Noch extremer ist dies auf der atomaren und subatomaren Ebene in der Quantenphysik, wo keine deterministischen Gesetze mehr greifen.

In diesen Systemen wird der Begriff „Zustand“ etwas anders gebraucht als im Makrokosmos. Man spricht oft von den inneren Zuständen eines Systems, was sich insbesondere auf

energetische Betrachtungen bezieht. Wie gesagt ist der Wechsel zwischen diesen Zuständen nicht mehr linear, sondern oft unvorhersagbar und nur noch stochastisch erfassbar. Will man ein solches Problem vollständig, das heißt unter Einbeziehung aller möglichen Entwicklungen des Anfangszustandes, am Computer simulieren, erinnert dies sehr an die Betrachtung NP-schwerer Probleme. Denn auch hier entsteht ein exponentiell wachsender Baum an Folgezuständen.

Festkörperphysik

In der Festkörperphysik beschäftigt sich vor allem mit Stoffen, die in bestimmten Strukturen, vor allem Kristallen, auftreten. Diese Strukturen sind eng mit den energetischen inneren Zustände des Systems verbunden. Ein Stoff in kristalliner Form ist energetisch gesehen stabil. Der Stoff kann jedoch auch in nicht stabilen Zuständen auftreten, wenn er sich im Übergang zwischen zwei stabilen Zuständen befindet. Dies ist in direkter Analogie zu den eingeführten algorithmischen Verfahren. Die stabilen Zustände, welche den Physiker interessieren, können als Endzustände gesehen werden, die gesucht werden. Vorgänge der Festkörperphysik können also mit heuristischen Verfahren simuliert und betrachtet werden.

Simulated Annealing

Das in Kapitel IV vorgestellte Verfahren Simulated Annealing mag dort etwas merkwürdig erschienen sein, die Formel, die die Wahrscheinlichkeit eines Rückschrittes angibt, scheint aus der Luft gegriffen. Dieser Algorithmus ist jedoch von einem Phänomen der Festkörperphysik abgeschaut! Der englische Begriff Annealing bezeichnet das langsame Abkühlen flüssiger Metalle in den Festzustand. Während des Abkühlungsprozesses wechselt der Stoff von einer höheren energetischen Konfiguration zu einer niedrigeren; am Ende dieses Prozesses steht ein stabiler energetisch minimaler Zustand. Allerdings ist es auch möglich, dass der Stoff während des Abkühlens einmal von einem niedrigeren in einen höheren Zustand wechselt, und zwar geschieht dies mit der Wahrscheinlichkeit

$p = e^{-\Delta E / kT}$,

wobei k die Bolzmann-Konstante ist , ΔE der Energieunterschied und T die Temperatur. Von dieser Formel ist das Simulated Annealing direkt abgeleitet. Kühlt ein Metall zu schnell ab, ist sein Endzustand nicht minimal (nur lokal, nicht global); die Aufgabe, Parameter des Kühlvorganges so zu bestimmen, dass ein minimaler Endzustand entsteht, ist nicht trivial. Das Simulated Annealing kann verwendet werden, um solche Parameter zu ermitteln.

VII) Heuristiken und Physik 2 – Theoreme beweisen

Die Physik ist zu einem großen Teil angewandte Mathematik. Physikalische Zusammenhänge werden in Formeln und Theoreme gefasst. Die theoretische Physik befasst sich mit dem Aufstellen und Beweisen solcher Formeln und Theoreme.
Zumindest theoretisch kann dieser Vorgang auch automatisiert werden, unter Zuhilfenahme der vorgestellten Algorithmen.

Automatisiertes Beweisen

Angenommen man hat zwei mathematische Ausdrücke und will beweisen, dass sie äquivalent sind, indem man sie ineinander umformt. Die beiden Terme könnten als Anfangs- und Endzustand des Problems verstanden werden, die durch geeignete Produktionsregeln – Termumformungen – ineinander überführt werden können.
Auch hier also lassen sich die Begrifflichkeiten von Zuständen, Produktionsregeln, Baumstruktur, etc. anbringen und damit Lösungen durch die betreffenden Algorithmen erzielen. Der Term an sich ist also eine Beschreibung des Zustandes und die unzähligen Rechenregeln, die auf ihn angewendet werden können, sind die Produktionsregeln. Prinzipiell ist die Menge der Produktionsregeln unendlich groß, dies stellt jedoch kein Hindernis dar, wenn auch bei der Wahl des Algorithmus hierauf Rücksicht genommen werden muss (der Algorithmus darf natürlich nicht mehr darauf beruhen, alle möglichen Folgezustände zu generieren und aus diesen heuristisch den besten auszuwählen, vielmehr muss er zunächst eine begrenzte Zahl „sinnvoller“ Produktionsregeln anwenden und damit auf Vollständigkeit verzichten). Aus logischer Sicht ist diese Betrachtung schlüssig. Praktisch gesehen ergeben sich jedoch eine Vielzahl von Schwierigkeiten, nicht nur, dass die Zahl der Produktionsregeln prinzipiell unendlich ist; einen Zustandscode, der allgemein mathematische Ausdrücke fasst, ist äußerst schwer zu finden.

Boolesche Algebra

Einfacher demonstrieren lässt sich das automatische Beweisen in der booleschen Algebra, das ja ein in sich geschlossenes Gebiet analog zur herkömmlichen Algebra bildet. Im Unterschied zu dieser ist hier die Zahl der möglichen Produktionsregeln begrenzt, überschaubar und wohl bekannt: Die Negation der Negation, das Kommutativgesetz, das Assoziativgesetz, das Distributivgesetz, die Idempotenzgesetze, die Komplementgesetze, die 0-1-Gesetze, die Absorptionsgesetze und die De Morgansche Regel. Als Beispiel:

$(b \vee a) \wedge \left(\overline{b \wedge a} \vee \bar{c}\right) \wedge \left(\overline{c \wedge \bar{b}} \vee \bar{a}\right)$ soll umgeformt werden zu $(b \vee a) \wedge \left(\bar{a} \vee \bar{b}\right)$, der Minimalform.

Anfanszustand: $(b \vee a) \wedge \left(\overline{b \wedge a} \vee \bar{c}\right) \wedge \left(\overline{c \wedge \bar{b}} \vee \bar{a}\right)$

Produktionsregel: $\overline{b \wedge a} \rightarrow \bar{b} \vee \bar{a}$ (De Morgansche Regel: $\overline{a \wedge b} \rightarrow \bar{a} \vee \bar{b}$)

$\Leftrightarrow (b \vee a) \wedge \left(\bar{b} \vee \bar{a} \vee \bar{c}\right) \wedge \left(\overline{c \wedge \bar{b}} \vee \bar{a}\right)$

Produktionsregel: $\overline{c \wedge \bar{b}} \rightarrow \bar{c} \vee b$ (De Morgansche Regel: $\overline{a \wedge b} \rightarrow \bar{a} \vee \bar{b}$)

$\Leftrightarrow (b \vee a) \wedge \left(\bar{b} \vee \bar{a} \vee \bar{c}\right) \wedge \left(\bar{c} \vee b \vee \bar{a}\right)$

Produktionsregel: $\left(\bar{b} \vee \bar{a} \vee \bar{c}\right) \wedge \left(\bar{c} \vee b \vee \bar{a}\right) \rightarrow \left(\bar{a} \vee \left(\bar{b} \vee \bar{c}\right) \wedge \left(b \vee \bar{c}\right)\right)$

(Distributivgesetz: $(a \vee b) \wedge (a \vee c) \rightarrow (a \vee (b \wedge c))$)

$\Leftrightarrow (b \vee a) \wedge \left(\bar{a} \vee \left(\bar{b} \vee \bar{c}\right) \wedge \left(b \vee \bar{c}\right)\right)$

Produktionsregel: $\bar{a} \vee \left(\bar{b} \vee \bar{c}\right) \wedge \left(b \vee \bar{c}\right) \rightarrow \bar{a} \vee \bar{c}$

(Absorptionsgesetz 4: $(a \vee b) \wedge \left(a \vee \bar{b}\right) \rightarrow a$)

$\Leftrightarrow (b \vee a) \wedge \left(\bar{a} \vee \bar{c}\right)$ (Endzustand)

Dies ist natürlich der ideale Gang, Aufgabe eines Algorithmus wäre es, diesen Weg zu finden, indem die richtigen Produktionsregeln zum richtigen Zeitpunkt ausgewählt werden.

VIII) Heuristiken und Physik 3 – Numerische Verfahren

Viele Probleme der Mathematik wie der Physik können durch numerische Verfahren gelöst werden. Diese Verfahren sind bis zu einem gewissen Grad verwandt mit den heuristischen Algorithmen.

Minimieren einer Zielfunktion

Numerische Verfahren werden insbesondere dazu benutzt, Extremstellen von Funktionen zu ermitteln. Wenn man davon ausgeht, dass ein Minimum gefunden werden soll, entspricht dies genau der Zielsetzung der heuristischen Algorithmen: auch sie minimieren eine Zielfunktion (die Kosten). Die heuristische Funktion ist bei den numerische Verfahren meist schon durch die Aufgabenstellung gegeben, die zu minimierende Funktion selbst gibt an, ob ein zur Debatte stehender Folgezustand in die richtige Richtung geht. Von den vorgestellten Algorithmen ist das Hillclimbing-Verfahren mit seinen Varianten am deutlichsten analog hierzu.

Ebenso wie das Hillclimbing-Verfahren laufen auch alle numerischen Verfahren zur Funktionsminimierung Gefahr, an einem lokalen Minimum hängen zu bleiben. Im Gegensatz zu einigen numerischen Verfahren ist das Hillclimbing wie alle heuristische Algorithmen zudem noch diskret, woraus das Problem entsteht, dass das gewählte Raster nicht fein genug ist und die Lösung „zwischen“ die möglichen Zustände fällt. Man kann diesem Problem bis zu einem gewissen Grad begegnen, indem man das Raster um so feiner macht, je näher man dem Endzustand kommt.

Beispiel: Feldfreier Punkt in einem System von gravitationserzeugenden Körpern

Beispielhaft soll hier ein physikalisches Problem mit einem der vorgestellten Algorithmen gelöst werden: In einem System von 10 Planeten, die unregelmäßig im Raum verteilt sind, soll ein Punkt gefunden werden, an dem sich die Gravitationsfelder gegenseitig aufheben. Auf einen Körper, der sich an diesem Punkt befände, würde keine Kraft wirken.

Das Problem lässt sich am besten über die Vektorrechnung lösen. Jeder Planet wirkt auf einen bestimmten Punkt mit einem bestimmten Gravitationsfeldstärke-Vektor. Die Vektoren aller Planeten addieren sich zu der Gesamtfeldstärke an dieser Stelle.

Der Betrag eines von einem einzelnen Planeten erzeugten Feldstärke-Vektors lässt sich bekanntlich mit der Formel

$$\left|\overrightarrow{G^*}\right| = \frac{k \cdot m}{\left|\vec{d}\right|^2},$$

wobei k die Gravitationskonstante und der Vektor $\vec{d}$ der Verschiebungsvektor von dem Planeten zu dem Punkt ist.

Für die Richtung des Feldstärke-Vektors gilt, dass er dieselbe Richtung haben muss, wie der Verschiebungs-Vektor. Insgesamt lässt er sich also angeben, indem man den mit der obigen Formel errechneten Betrag mit dem Einheits-Verschiebungs-Vektor multipliziert:

$$\vec{G} = \frac{1}{\left|\vec{d}\right|} \cdot \vec{d} \cdot \frac{k \cdot m}{\left|\vec{d}\right|^2}$$

Die Gesamt-Feldstärke an einem Ort wird daher ermittelt:

$$\vec{G} = \sum_{i=1}^{n} \left(\overrightarrow{d_i} \cdot \frac{k \cdot m_i}{\left|\overrightarrow{d_i}\right|^3} \right)$$

Gemäß der Aufgabenstellung soll die Lösung dieser Gleichung der Vektor $\begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$ sein.

Das Problem soll nun in ein geeignetes Modell gefasst werden, um es algorithmisch zu lösen.

a) der Zustandscode

   Ein Zustand ist hinreichend beschrieben durch den Ortsvektor des zu betrachtenden Punktes und der Betrag der an diesem Punkt existierenden Feldstärke-Vektors.

   Der Zustandscode lautet damit:

   $$\left[\vec{x}; \mathrm{G}\left(\vec{x}\right)\right]$$

b) der Anfangszustand

   Der Anfangszustand ist prinzipiell beliebig:

   $$z_0 = \left[\overrightarrow{x_0}; \mathrm{G}\left(\overrightarrow{x_0}\right)\right]$$

c) der Zielzustand

   Der Betrag des Feldstärkevektors soll 0 sein:

   $$z_f = \left[\overrightarrow{x_f}; 0\right]$$

d) die Produktionsregeln

   Ein Zustand lässt sich ändern, indem man den Punkt x verschiebt. x kann dabei gegen-

über jeder Achse positiv oder negativ um einen festen Betrag verschoben werden. Damit ergeben sich sechs Produktionsregeln, die sich alle auf jeden möglichen Zustand anwenden lassen:

1) $\left[\vec{x}; \mathrm{G}\left(\vec{x}\right)\right] \rightarrow \left[\vec{x} + \begin{pmatrix} c \\ 0 \\ 0 \end{pmatrix}; \mathrm{G}\left(\vec{x} + \begin{pmatrix} c \\ 0 \\ 0 \end{pmatrix}\right)\right]$

2) $\left[\vec{x}; \mathrm{G}\left(\vec{x}\right)\right] \rightarrow \left[\vec{x} - \begin{pmatrix} c \\ 0 \\ 0 \end{pmatrix}; \mathrm{G}\left(\vec{x} - \begin{pmatrix} c \\ 0 \\ 0 \end{pmatrix}\right)\right]$

3) $\left[\vec{x}; \mathrm{G}\left(\vec{x}\right)\right] \rightarrow \left[\vec{x} + \begin{pmatrix} 0 \\ c \\ 0 \end{pmatrix}; \mathrm{G}\left(\vec{x} + \begin{pmatrix} 0 \\ c \\ 0 \end{pmatrix}\right)\right]$

4) $\left[\vec{x}; \mathrm{G}\left(\vec{x}\right)\right] \rightarrow \left[\vec{x} - \begin{pmatrix} 0 \\ c \\ 0 \end{pmatrix}; \mathrm{G}\left(\vec{x} - \begin{pmatrix} 0 \\ c \\ 0 \end{pmatrix}\right)\right]$

5) $\left[\vec{x}; \mathrm{G}\left(\vec{x}\right)\right] \rightarrow \left[\vec{x} + \begin{pmatrix} 0 \\ 0 \\ c \end{pmatrix}; \mathrm{G}\left(\vec{x} + \begin{pmatrix} 0 \\ 0 \\ c \end{pmatrix}\right)\right]$

6) $\left[\vec{x}; \mathrm{G}\left(\vec{x}\right)\right] \rightarrow \left[\vec{x} - \begin{pmatrix} 0 \\ 0 \\ c \end{pmatrix}; \mathrm{G}\left(\vec{x} - \begin{pmatrix} 0 \\ 0 \\ c \end{pmatrix}\right)\right]$

e) die heuristische Abschätzungsfunktion
   die zu minimierende Funktion, G, gibt selbst Aufschluss über die Qualität eines Zustandes: Je niedriger die Funktion ist, desto näher ist der Zustand am Endzustand. G bleibt dennoch heuristisch: es ist nicht garantiert, dass sie zum Ziel führt. So würde sie zum Beispiel vom Ziel wegführen, wenn der Anfangszustand zu weit außerhalb angesetzt wurde. Der Algorithmus wird zudem nur bis zu einer gewissen Genauigkeit an das Ziel herankommen.

Mit diesem Modell kann man nun aus den vorgestellten Verfahren ein geeignetes auswählen. Am geeignetesten erscheint dabei ein Hillclimbing-Algorithmus (siehe oben), aus folgenden Gründen:
Alle Transformationen sind reversibel und es gibt unendlich viele Zustände. Das Hillclimbing kann daher nie in eine Sackgasse geraten, weil keine Nachfolger mehr erzeugt werden

können. Es ist eine sehr informative Heuristik vorhanden, die es unwahrscheinlich macht, an lokalen Minima hängen zu bleiben. Zu beachten ist jedoch, dass ein Anfangszustand gewählt werden muss, der einigermaßen zentral unter den Planeten liegt, da ansonsten das Hillclimbing immer weiter weg driftet. Es kann das Steepest-Ascent-Prinzip angewendet werden.

Mit diesen Überlegungen lässt sich das folgende Pascal-Programm implementieren, das das Problem löst:

```
program gravitation;
{$N+}


uses crt;


const
k=10;
c_0=2.5;{Iterationsschritt}

type
tvector = array[1..3] of double;
tplaneten_koord=array[1..10,1..3] of double;
tplaneten_masse=array[1..10] of double;
tzustand=record
 pos:tvector;
 g:double;
end;{record}


var
i:integer;
planetenkoord:tplaneten_koord;
masse:tplaneten_masse;
zustand:tzustand;
nachfolger:tzustand;
bestnachfolger:tzustand;
schritte:longint;
c:double;

{==================Vektorrechnung=========================}
function vbetrag(x:tvector):double;
{errechnet den Betrag eines Vektors}
var
help:double;
begin
help:=sqrt(x[1]*x[1]+x[2]*x[2]+x[3]*x[3]);
vbetrag:=help;
end;

procedure vscale(var x:tvector;r:double);
{mulitpliziert einen Vektor mit einem konstanten Betrag}
begin
```

```
x[1]:=x[1]*r;
x[2]:=x[2]*r;
x[3]:=x[3]*r;
end;

procedure vadd(x1,x2:tvector;var res:tvector);
{addiert zwei Vektoren}
begin
res[1]:=x1[1]+x2[1];
res[2]:=x1[2]+x2[2];
res[3]:=x1[3]+x2[3];
end;

{============================================================}
procedure init;

{Die Daten der 10 Planeten}
begin
planetenkoord[1,1]:=52;
planetenkoord[1,2]:=12;
planetenkoord[1,3]:=-26;
masse[1]:= 100;

planetenkoord[2,1]:=10;
planetenkoord[2,2]:=-33;
planetenkoord[2,3]:=-20;
masse[2]:= 100;

planetenkoord[3,1]:=-44;
planetenkoord[3,2]:=10;
planetenkoord[3,3]:=10;
masse[3]:= 76;

planetenkoord[4,1]:=30;
planetenkoord[4,2]:=-53;
planetenkoord[4,3]:=0;
masse[4]:= 130;

planetenkoord[5,1]:=15;
planetenkoord[5,2]:=25;
planetenkoord[5,3]:=35;
masse[5]:= 80;

planetenkoord[6,1]:=-50;
planetenkoord[6,2]:=-31;
planetenkoord[6,3]:=-19;
masse[6]:= 950;

planetenkoord[7,1]:=20;
planetenkoord[7,2]:=-1;
planetenkoord[7,3]:=15;
masse[7]:= 95;

planetenkoord[8,1]:=37;
planetenkoord[8,2]:=-37;
planetenkoord[8,3]:=26;
masse[8]:= 115;

planetenkoord[9,1]:=9;
planetenkoord[9,2]:=-37;
planetenkoord[9,3]:=-17;
masse[9]:= 89;
```

```
planetenkoord[10,1]:=39;
planetenkoord[10,2]:=44;
planetenkoord[10,3]:=-5;
masse[10]:= 100;
end;



procedure entf(planet:integer;x:tvector; var res:tvector);
{Berechnet den Vektor von einem Planeten zu einem Punkt}
var
help:tvector;
begin
help[1]:=planetenkoord[planet,1];
help[2]:=planetenkoord[planet,2];
help[3]:=planetenkoord[planet,3];
vscale(help,-1);
vadd(x,help,res);
end;


function G(x:tvector):double;
{Liefert den Betrag der Feldstaerke an einem Punkt}
var
i:integer;
help:tvector;
entfv:tvector;
begin
help[1]:=0;
help[2]:=0;
help[3]:=0;
for i:= 1 to 10 do begin
entf(i,x,entfv);
vsca-
le(entfv,(k*masse[i]/((vbetrag(entfv)*(vbetrag(entfv)*(vbetrag(entfv)))))))
;
vadd(help,entfv,help);
end;{for}
G:=vbetrag(help);
end;

procedure gennachf(z:tzustand;regel:integer;var nachf:tzustand);
{Die Produktionsregeln}
begin
nachf:=z;
case regel of
1: begin
    nachf.pos[1]:=z.pos[1]+c;
    nachf.pos[2]:=z.pos[2];
    nachf.pos[3]:=z.pos[3];
   end;
2: begin
    nachf.pos[1]:=z.pos[1]-c;
    nachf.pos[2]:=z.pos[2];
    nachf.pos[3]:=z.pos[3];
   end;
3: begin
    nachf.pos[1]:=z.pos[1];
    nachf.pos[2]:=z.pos[2]+c;
    nachf.pos[3]:=z.pos[3];
   end;
4: begin
```

```
    nachf.pos[1]:=z.pos[1];
    nachf.pos[2]:=z.pos[2]-c;
    nachf.pos[3]:=z.pos[3];
   end;
5: begin
    nachf.pos[1]:=z.pos[1];
    nachf.pos[2]:=z.pos[2];
    nachf.pos[3]:=z.pos[3]+c;
   end;
6: begin
    nachf.pos[1]:=z.pos[1];
    nachf.pos[2]:=z.pos[2];
    nachf.pos[3]:=z.pos[3]-c;
   end;
end;{case}
nachf.g:=G(nachf.pos)
end;


begin {Hauptprogramm}

{Initialisieren}
init;
schritte:=0;
clrscr;
{Anfangszustand}
zustand.pos[1]:=0;
zustand.pos[2]:=0;
zustand.pos[3]:=0;
zustand.g:=G(zustand.pos);

{Anfangszustand ausgeben}
write(zustand.pos[1],' ',zustand.pos[2],' ',zustand.pos[3]);
writeln;
writeln(zustand.g);

{Steepest Ascent Hillclimbing}
repeat
 schritte:=schritte+1;
 gennachf(zustand,1,bestnachfolger);
 c:=c_0/schritte;{Groesse des Iterationsschritt}
 for i:=2 to 6 do begin {Nachfolger generieren}
  gennachf(zustand,i,nachfolger);
  if nachfolger.g<bestnachfolger.g then begin
   bestnachfolger:=nachfolger;
  end;{if}
 end;{for}
 if zustand.g>bestnachfolger.g then begin
  zustand:=bestnachfolger;
 end;{if}
until bestnachfolger.g > zustand.g;

{Enzustand ausgeben}
writeln;
write(zustand.pos[1],' ',zustand.pos[2],' ',zustand.pos[3]);
writeln;
writeln(zustand.g);
writeln;
writeln(schritte);
readln;
end.
```

Das Programm löst das Problem für 10 willkürlich verteilte Planeten an den Positionen (52|12|-26), (10|-33|-20), (-44|10|10), (30|-53|0), (15|25|35), (-50|-31|-19), (20|-1|15), (37|-37|26), (9|-37|-17), (39|44|-5)

Die Massen sind ebenfalls willkürlich gesetzt.

Der Anfangspunkt ist (0|0|0)

Als Gravitationskonstante wurde 10 verwendet, was die Feldstärken in einen sinnvollen Bereich bringt. Dies ist insofern legitim, als dass keine Skala bei den Raumachsen angegeben wurde.

Um den Algorithmus etwas zu beschleunigen, wurden keine konstanten Iterationsschritte verwendet; das Raster wird fortschreitend feiner.

Nach dem Durchlauf liefert das Programm folgendes Ergebnis:

Anfangszustand:[(0|0|0); 2,2429]

Endzustand:[(13,169|-13,852|5,535); $1{,}5 \cdot 10^{-6} \approx 0$]

Iterationsschritte: 156 369

Quellen

[Dresden] Lernprogramm zur Planung und Optimierung von Telekommunikationsnetzen; http://www.ifn.et.tu-dresden.de/TK/deutsch/lernprogramme.htm (16.03.2002)

[Gasper] Gasper, F. et al; Technische und theoretische Informatik; Bayrischer Schulbuchverlag; München 1992

[Gritzmann] Gritzmann, Peter; Brandenberg, René; Das Geheimnis des kürzesten Weges; Springer; Berlin 2002

[Jerusalem] o. V.; Discrete Math and Theoretical Computer Science; http://www.cs.huji.ac.il/~avi/discomblurb.htm (1.5.2002)

[NIST] Dictionary of Algorithms, Data Structures, and Problems; \ http://www.nist.gov/dads/ (16.03.2002)

[Pearl] Pearl, Judea; Heuristics: Intelligent Search Strategies for Computer Problem Solving; Addison-Wesley, Reading (MA) 1984

[Pool] Pool, Robert; When Easy Maths Turns Hard; http://domino.research.ibm.com/comm/wwwr_thinkresearch.nsf/pages/computing100.html (1.5.2002)

[Reif] Reif, Gerald; Moderne Aspekte der Wissensverarbeitung; Diplomarbeit an der TU Graz; 20. Januar 2000

[Rich] Rich, Elaine; Knight, Kevin; Artificial Intelligence; McGraw-Hill, 1983

[Wolfram] Wolfram, Stephen.; Undecidablility and Intractability in Theoretical Physics; http://www.stephenwolfram.com/publications/articles/physics/85-undecidability/2/text.html (1.5.2002)