Veranstaltung: Seminar im Grundstudium
“Künstliche Intelligenz”
Prof. Dr. W. Bibel

S. Russel, P. Norvig:

Artificial Intelligence

Kapitel 4:

Informed Search Methods

Zusammenfassung des Vortrages von:
Patrick Seiler, Mat.: 628961

Kapitel 4

Das Kapitel 3 hat gezeigt, daß die uninformierten Suchstrategieen unglaublich uneffizient sind.

Dieses Kapitel führt uns nun in die informierten Suchstrategieen ein. Dabei wird problemspezifisches Wissen benutzt, um Lösungen effizienter zu finden. Dieses Kapitel zeigt auch, wie Optimierungsprobleme gelöst werden können.

Inhaltsverzeichnis

4.1 Best-First Suchalgorithmen 3

4.1.1 Generelle Suche 3
4.1.2 Gefräßige (Greedy) Suche 3
4.1.3 A* Suche 4

4.2 Heuristische Funktionen 5

4.2.1 8er Puzzle 5
4.2.2 Auswirkung der heuristischen Genauigkeit auf die Geschwindigkeit 5
4.2.3 Finden von heuristischen Funktionen 6

4.3 Speichergebundene Suche 6

4.3.1 IDA* Suche 6
4.3.2 SMA* Suche 6

4.4 Verbesserte Iterative Algorithmen 7

4.4.1 Bergsteigende Suche. 7
4.4.2 Simuliertes Ausglühen/Härten 8

4.5 Zusammenfassung 9

4.6 Historisches 10

4.1 Best-First Suchalgorithmen

4.1.1 Generelle Suche

Die generelle Suche benötigt problemspezifisches Wissen für die Entscheidung welcher Knoten als nächstes besucht werden soll. Das Wissen wird in eine s.g. Evaluierungs Funktion (evaluation function) gesteckt.
Diese Evaluierungs Funktion ist meist schwierig zu erstellen, nur wenn sie allwissend ist, dann ist auch der erste Knoten der beste Knoten. Manchmal kommt die Suche vom Rechten Weg ab, wenn die Evaluierungs Funktion nicht gut genug erstellt wurde. Sind die Knoten so angeordnet, daß der, mit der besten Evaluierungs Funktion zuerst bearbeitet wird, dann erhält man die Bestes-zuerst Suche.

Da viele generelle Suchalgorithmen mit verschiedenen Suchfunktionen existieren, gibt es viele Bestes-zuerst Suchalgorithmen mit verschiedenen Evaluierungs Funktionen. Diese stellen eine eigene Familie von Algorithmen dar.

Das Ziel bei diese Suche ist, die Lösung mit den geringsten Kosten zu finden. Dazu werden die Kosten abgeschätzt und versucht zu minimieren. Entscheident hierbei sind die Pfadkosten *g*. Die Suche findet aber keinen direkten Weg zum Ziel. Möglichkeiten die sich hiebei ergeben, lauten

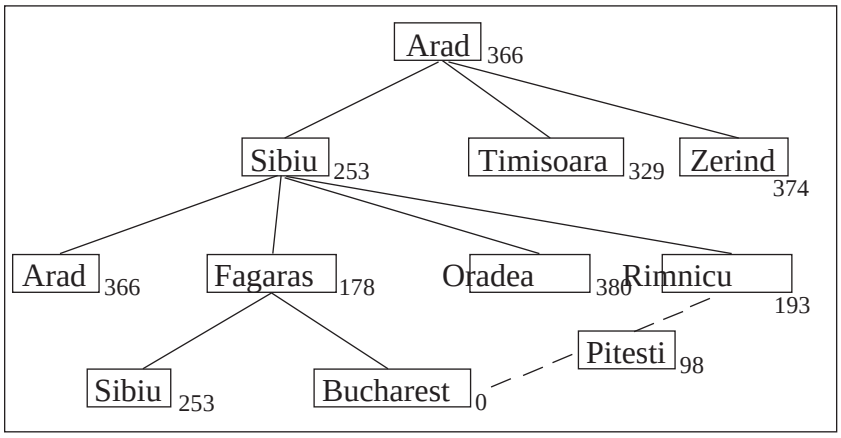
1. *bearbeite den Knoten, der näher zum Ziel liegt*
2. *bearbeite den Knoten, mit den niedrigsten (Pfad-)Kosten*

Eine Kombination dieser beiden Möglichkeiten ist die beste Lösung.

4.1.2 Gefräßige (Greedy) Suche

Die einfachste Bestes-zuerst Strategie ist die Minimierung der abgeschätzten Kosten, d.h. der Knoten dessen Status am nächsten dem Status des Ziels ist, wird zuerst bearbeitet. Das Problem hierbei ist, das dies nur eine Schätzung ist.

Gefräßige Suche benutzt *h(n)* zum Wählen des Nachfolgers.
Die heuristische Funktion, *h(n)*, stellt die abgeschätzten Kosten des billigsten Pfades vom Knoten *n* zum Ziel dar. Zum erstellen einer heuristischen Funktion kann jede beliebige Funktion benutzt werden, verlangt wird nur, daß *h(n)=0* wenn *n* der Zielknoten ist.
Heuristische Funktionen sind sehr problemspezifisch, d.h. jedes Problem kann seine eigene Heuristik besitzen, natürlich auch mehr als eine.



Das Wegeproblem von Arad nach Bukarest wurde schon im 3. Kapitel angesprochen (siehe Grafik, Knoten sind beschriftet mit *h(n)*).
Der direkte-Weg-Abstand (Straight Line Distance) *hsld(n)* ist eine gute Möglichkeit für Wegeprobleme, dies ist sozusagen eine Luftlinienverbindung zwischen *n* und dem Ziel. Benötigt werden hierzu die Koordinaten der Städte und die Entfernungen der einzelnen Städte untereinander, um eine gute heuristische Funktion erstellen zu können.
Hierbei wird nur die minimale Sucharbeit geleistet, d.h. es wird kein Knoten besucht, der nicht auf dem Lösungsweg liegt. Leider war die Suche in diesem Fall nicht optimal, der gefundene Weg ist 32 Meilen länger als der (gestrichelte) über Rimnicu und Pitesti.

Allgemein kann man sagen, findet die gefräßige Suche Lösungen normalerweise recht schnell, ist aber dafür nicht immer optimal. Probleme die sich hierbei ergeben sind Startfehler, Sackgassen und unnötige Knotenuntersuchungen. Greedy Suche findet nicht immer Lösungen, im allgemeinen hat diese Suchmethode die gleichen Probleme wie alle Tiefensuche Algorithmen, wie z.B. Endlosschleifen.

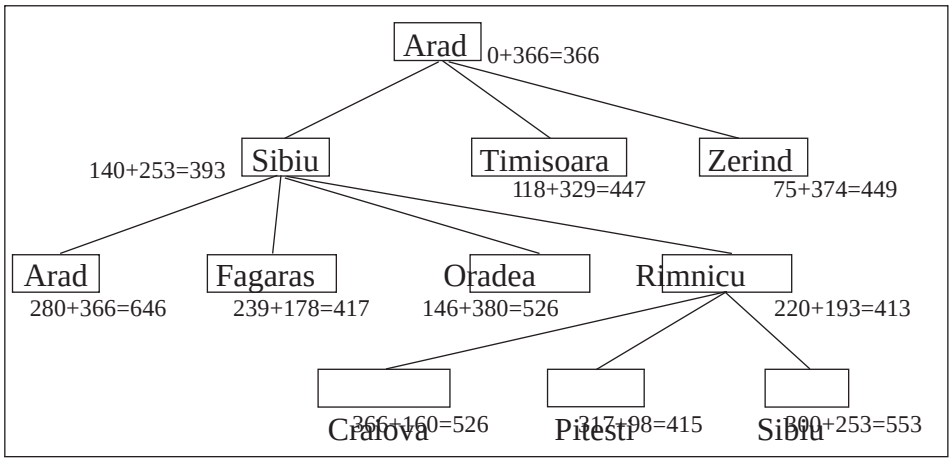
4.1.3 A*-Suche

Die uninformierte-Kosten Suche *g(n)*, schon bekannt aus dem Kapitel 3, beachtet zwar die Kosten des Pfades, ist optimal und abgeschlossen, ist aber auch sehr ineffizient.
Die billigste Lösung der Suche ist eine Kombination von *g(n)* und einer heuristischen Funktion *h(n)*, durch eine Addition von *g* und *h*. Diese profitiert von den Vorteilen beider.

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

f(n) stellt dabei die Funktion der geschätzten Kosten der billigsten Lösung über den Knoten *n* dar. Die Strategie für diese Kombination lautet hierbei: nehme den Knoten mit dem niedrigstem Wert von *f(n)*.

Eine zulässige Heuristik besitzt die einzige Einschränkung, daß die Wahl einer Funktion *h* so getroffen werden muß, daß die Kosten zum Erreichen des Ziels nicht zu hoch werden. Bestes Beispiel ist unsere Funktion *Hsld(n)*. Fast alle zulässigen heuristischen Funktionen zeigen das gleiche Verhalten, sie sind monoton steigend; d.h., von der Wurzel zum Ziel werden die *F*-Kosten nicht geringer. Die wenigen nicht monotonen lassen sich leicht ändern.



Wie an der Grafik zu erkennen ist, findet der A* Such- algorithmus jetzt den optimalen Weg nach Bucharest (die letzte Station der Suche ist nicht mehr eingezeichnet und die Knoten sind beschriftet mit f=g+h). Der gefundene Weg führt jetzt über Rimnicu und Pitesti nach Bucharest und ist daher 32 Meilen kürzer als bei gefräßiger Suche.

Zur Komplexität kann man sagen, daß A* keine Antwort auf alle Suchprobleme ist, obwohl der Algorithmus vollständig, optimal und effizient ist. Eine Aufwandsbetrachtung für diesen Algorithmus liefert:

$$|h(n) - h^*(n)| \quad \Theta(\log h^*(n))$$

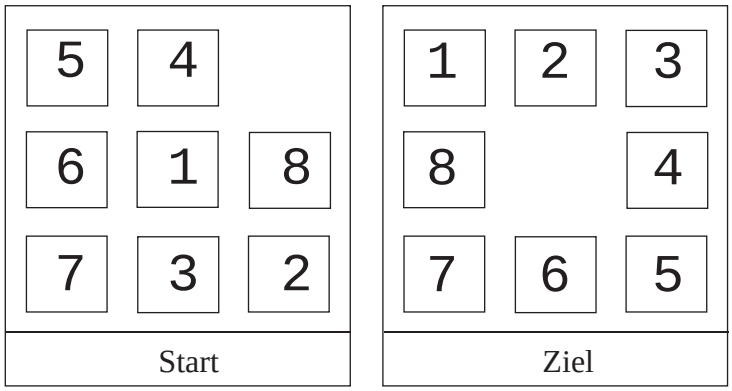
h(n) sind hierbei die wahren Kosten vom Knoten n zum Ziel.
Abschließend kann man feststellen, A* ist optimal-effizient, d.h. es gibt keinen Algorithmus, der mit weniger Knotenuntersuchungen auskommt. Das Problem bei der A* Suche ist auch nicht die benötigte Zeit, sondern der Speicherverbrauch.

4.2 Heuristische Funktionen

4.2.1 8er Puzzle

Das 8er Puzzle stellt eines der ersten heuristischen Suchprobleme dar (siehe Bild).

Eine typische Lösung dieses Puzzles benötigt ca. 20 Schritte, natürlich abhängig von der Startposition. Durchschnittlich gibt es drei Möglichkeiten zum Schieben der Teile, falls das mittlere bewegt werden soll, sind es vier, in den Ecken drei, sonst weniger.
Eine Suche in die Tiefe benötigt daher ca. 3^{20} Schritte ($=3{,}5*10^9$), und das, obwohl es nur 9! (=362.888) verschiedene Möglichkeiten gibt um neun Quadrate anzuordnen. Das ist keine Meisterleistung, daher macht man sich auf die Suche nach einer guten heuristischen Funktion.



Zwei Möglichkeiten die das Buch an dieser Stelle bietet, lauten:

h1 = Zahl der Teile an falscher Position = 8

dies ist eine zulässige Heuristik, da jedes Teil welches an einer falschen Position ist, mindestens einmal bewegt werden muß,

h2 = Summe des Abstandes der Teile zum Ziel = 15

wird auch der Manhatten Abstand genannt. Die zweite Heuristik ist auch zulässig, da jede Bewegung nur ein Teil und ein Stück näher zum Ziel bringt.
Gesucht ist nun eine Aussage über die Nützlichkeit der Heuristik in Bezug auf die Geschwindigkeit.

4.2.2 Auswirkung der heuristischen Genauigkeit auf die Geschwindigkeit

Um bei eigenen Problemen auch eigene Heuristiken zu verwenden, muß man entscheiden können, ob und wie gut eine erstellte Heuristik ist. Eine gute Aussage über die Nützlichkeit der Heuristik ist der effektive Verzweigungsfaktor b*

$$b^* - N = 1 + b^* + (b^*)^2 + ... + (b^*)^d$$

N sind die untersuchten Knoten, d die Lösungstiefe und b* ist der Verzweigungsfaktor. Ein Beispiel um dies verständlich zu machen: wenn *d=5* und *N=52* dann beträgt der effektive Verzweigungsfaktor *b*=1,91*. b* ist relativ konstant für einzelne Heuristiken. Gut designte heuristische Funktionen besitzen einen Faktor b* der ungefähr bei 1 liegt. Um wieder auf unser Beispiel zurückzukommen:

uninformierte Tiefensuche: b* 2,75

A*(h1): b* 1,46

A*(h2): b* 1,31

h2 ist also immer besser als *h1*, denn es braucht durchschnittlich weniger Knoten als die Heuristik *h1*. Man kann sagen, daß *h2*, *h1* domniert. Allgemein ist es besser heuristische Funktionen mit großen Werten auszuwählen: *h2=15 > h1=8.*

4.2.3 Finden von heuristischen Funktionen

Nachdem die zwei Beispiele nun durch das Buch gegeben waren fragt man sich, wie kommt man selbst auf eigene Heuristiken? Dazu hat man die entspannten Probleme eingeführt. Bei einer Regeländerung (z.B. alle Teile können überallhin bewegt werden) liefert *h1* die Anzahl der Schritte für die kürzeste Lösung.
Entspannte Probleme sind Probleme mit weniger Einschränkungen. "Ein Teil kann bewegt werden von *A* nach *B*, wenn *A* an *B* angrenzt und *B* ist leer", bedeutet dann:

(a) Ein Teil kann bewegt werden, wenn *A* an *B* angrenzt

(b) Ein Teil kann bewegt werden, wenn *B* leer ist

(c) Ein Teil kann bewegt werden von *A* nach *B*

Solche Probleme beschreibt man eigentlich in sogenannten formalen Sprachen, wie noch in den Kapiteln 7 und 11 zu sehen ist. *Absolver* ist ein Automat, der erfunden wurde, um Heuristiken automatisch aus entspannten Problemen zu generieren. Die erste benutzbare Heuristik für den Zauberwürfel Rubik's Cube wurde damit entdeckt.

4.3 Speichergebundene Suche

4.3.1 IDA* Suche

Ungeachtet der intelligenten Suchalgorithmen die gefunden wurden, gibt es noch Probleme, die wirklich schwer zu lösen sind. Eines dieser Probleme ist, den vorhandenen Speicher optimal zu nutzen. Daher kommen jetzt zwei Algorithmen, die Speicher sparen.
Kapitel 3 zeigt, daß wiederholende Tiefensuche (ID) gut zum Speichersparen geeignet ist, wiederholende A* Tiefensuche (IDA*) baut darauf auf, mit zusätzlichen heuristischen Informationen. Auch für diesen Algorithmus gilt, jede Wiederholung ist eine Tiefensuche, wie bei jedem ID-Suchalgorithmus. Aber, die Tiefensuche wurde abgeändert, um ein f-Kosten Limit statt eines Tiefen Limits zu benutzen.

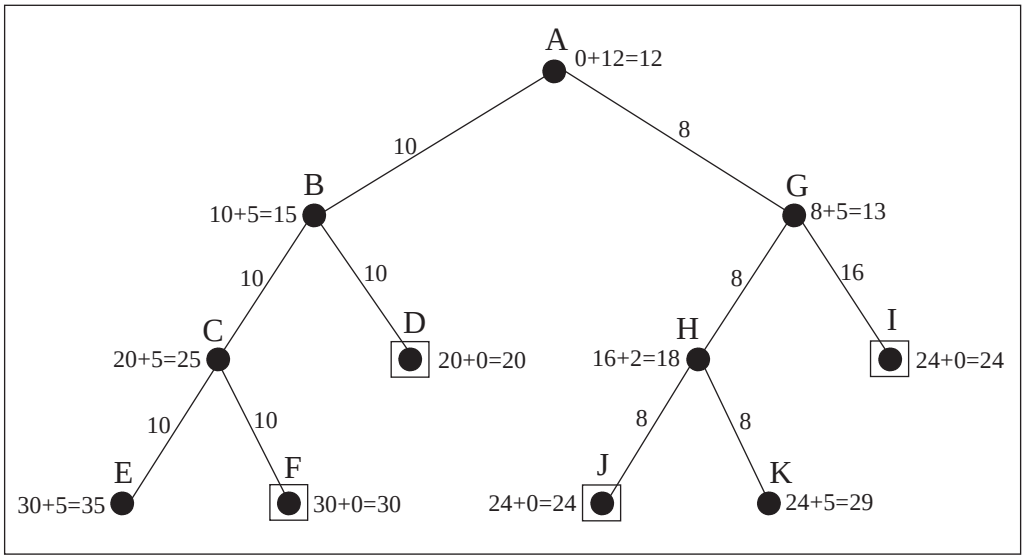
Das Problem des Handlungsreisenden ist mit $\Theta(N^2)$ zu komplex für IDA*. Eine Möglichkeit dies zu Umgehen, ist das Steigern des f-Kosten-Limits um einen Betrag bei jeder Iteration, so daß die Anzahl der Wiederholungen proportional zu 1/ ist. Dieser Algorithmus zur Reduzierung der Suchkosten nennt sich -Zulässig.

4.3.2 SMA* Suche

Die SMA* Suche, oder auch vereinfachte Speichergebundene A* Suche, benutzt das komplette freie RAM. Daraus ergibt sich, daß je mehr Speicher vorhanden ist, desto mehr Knoten durch den Algorithmus gesichert werden können (um so weniger müssen neu gebildet werden). Die fünf allgemeinen Punkte, welche die SMA* Suche charakterisieren, lauten:

- benutzt jeglichen Speicher der angeboten wird, um soviel wie nur möglich Knoten und Zustände zu speichern.
- vermeidet Wiederholungen sowie die Speichersituation dies erlaubt und zwar dadurch, daß alles im Speicher gehalten wird.
- Vollständigkeit, wenn genug Speicher vorhanden ist, um den einfachsten Lösungsweg aufzunehmen.
- optimal, wenn genug Speicher vorhanden ist, sonst liefert er die beste Lösung, die der Speicher zuläßt.
- wenn der komplette Suchbaum in den Speicher geht, dann ist SMA* optimal-effizient.

Sobald Speicher für Nachfolgerknoten benötigt wird, wendet der SMA* Algorithmus die Technik der vergessenen Knoten an, d.h. es wird ein Vorgängerknoten geopfert, der sehr hohe f-Kosten besitzt. Um zu verhindern, daß solche weggeworfenen Knoten nochmal durchsucht werden, wird im Vorgängerknoten Information über den besten Weg (über den vergessenen Knoten) gesichert.



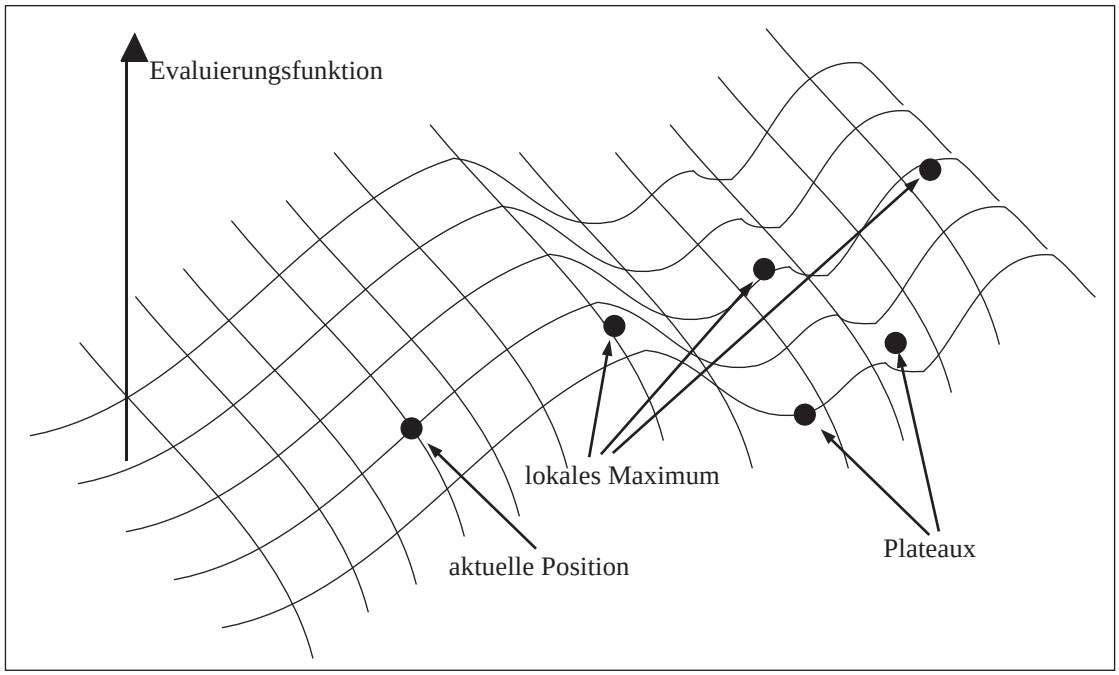
In dem folgendem Beispiel ist *A* der Anfang der Suche, die Knoten sind beschriftet mit *f=g+h* und die Kanten mit den Kosten von Knoten zu Knoten. Im ganzen gibt es hier vier Zielknoten *D,F,I* und *J*. Das Ziel der Suche lautet, den Knoten mit den niedrigesten Kosten zu finden. Mit der einzigen Einschränkung, daß nur Speicher für drei Knoten vorhanden ist. Daß diese Einschränkung von großer Auswirkung ist, erkennt man daran, daß die Zielknoten *F* und *J* nur über vier Knoten von A aus erreichbar sind, d.h. bei vorhandenem Speicher für nur 3 Knoten, können diese beiden überhaupt nicht untersucht werden.
Wie SMA* Suche genau funktioniert, wird in dem Buch auf den Seiten 108-110 detailliert beschrieben. Hier sei nur gesagt, daß der Knoten *D* als beste Lösung gefunden wird.

In dem Beispiel ist genug Speicher für die beste Lösung vorhanden. Wenn *J* Pfadkosten von *19* anstatt *24* besitzen würde, dann wäre zu wenig Speicher für die beste Lösung vorhanden, dadurch würde *D* nur die beste erreichbare Lösung darstellen. Bei genügend Speicher kann SMA* viel kompliziertere Probleme lösen, als A*, ohne dabei zu viele unnötige Knoten zu bearbeiten. Eine Speichereinschränkung kann allerdings die Lösung erschweren, z.B. erhöht sich dann die Zeit der Bearbeitung wegen wiederholtem Durchsuchen der gleichen Knoten. Dabei ist es schwer, ein Mittelmaß zwischen dem Speicherverbrauch und der Bearbeitungszeit zu finden.

4.4 Verbesserte Iterative Algorithmen

4.4.1 Bergsteigende Suche

Die Hill-Climbing- oder Bergsteigende-Suche, gehört zu einer neuen Problemklasse, bei dieser ist der Weg, der zum Ziel führt uninteressant, nur das Ziel selbst zählt. Ein Beispiel für diese Problemklasse ist das acht Königinnen-Problem, wobei mit allen 8 Königinnen auf dem Brett gestartet wird und ein paar herumbewegt werden, um die Zahl der möglichen Angriffe zu reduzieren.
Der beste Weg, um verbesserte Iterative Algorithmen zu verstehen, ist sich vorzustellen, daß alle Stationen der Suche auf einer Landkarte liegen und die Höhe der einzelnen Punkte von der Evaluierungsfunktion abhängt. Die Idee dabei ist, die lokalen Maximas zu suchen, denn diese stellen die optimale Lösung für diese Problemklasse dar. Der Hill-Climbing Algorithmus versucht Änderugen einzuführen, um die aktuelle Position zu verbessern.
Der Aufbau des Algorithmus ist ganz simpel, er besteht aus einer einfachen Schleife, die in Richtung der aufsteigenden Werte zählt und falls es mehrere beste Nachfolger gibt, wird ein zufälliger ausgewählt.



Die drei wichtigen Begriffe, welche den Hill-Climbing Algorithmus charakterisieren, lauten:

- *lokales Maximum*: ein lokales Maximum ist im Gegensatz zum globalen Maximum eine Anhebung, die niedriger ist, als die höchste Anhebung.
  An einem lokalen Maximum stoppt der Algorithmus, auch wenn das keine zufriedenstellende Lösung ist.

- *Hochebene (Plateaux)*: ein Gebiet der Suche, bei dem die Evaluierungsfunktion sehr flach ist. Der Algorithmus wählt dann einen zufälligen Weg aus.

- *Gebirgskamm*: 1. kann steil Ansteigen, dadurch erreicht die Suche die Anhebung mit Leichtigkeit.
  2. eine schwache Steigung ist auch möglich, dann dauert die Suche länger.

Man erkennt sofort, daß der Algorithmus einen oder auch mehrere Punkte erreicht, an denen die Suche abbricht, so z.B. bei den Hochebenen. Da ist es nur einleuchtend, mit dem Neustart bei einem anderen Startpunkt fortzufahren.
Zufällig neustartendes Bergsteigen führt eine Serie von Hill-Climbing-Suchen mit zufällig ausgewählten Startpunkten durch, solange bis der Algorithmus endet, oder kein Fortschritt mehr erkennbar ist. Dies wird fortgesetzt, bis eine bestimmte Anzahl von Schleifendurchgängen erreicht ist.

Abschließend kann man sagen, daß, wenn genug Wiederholungen möglich sind, die optimale Lösung vom zufällig neustartendem Bergsteigen gefunden wird. Bei wenigen lokalen Maximas ist eine gute Lösung recht schnell gefunden.

4.4.2 Simuliertes Ausglühen/Härten

Anstatt einen zufälligen neuen Startpunkt zu wählen, wenn der Algorithmus abbricht, gibt es die Möglichkeit, wieder ein Stück zurückzugehen, dies entspricht der Idee des simulierten Ausglühens.
Der Algorithmus ist ähnlich aufgebaut wie die Hill-Climbing Suche, doch statt den besten Weg, nimmt dieser Algorithmus einen Zufälligen. Falls der Weg die Situation verbessert, wird er ausgeführt, sonst wird nur mit einer Wahrscheinlichkeit kleiner 1 ausgeführt. Die Wahrscheinlichkeit fällt mit der Schlechtigkeit der Bewegung, ausgedrückt durch E; T drückt hier die Wahrscheinlichkeit selbst aus.

```
function SIMULATED-ANNEALING( problem, schedule ) returns a solution state
    inputs:   problem, a problem
              schedule, a mapping from time to "temperature"
    static:   current, a node
              next, a node
              T, a "temperature" controlling the probatility of downward steps

    current  M  AKE-NODE( INITIAL-STATE[problem] )
    for t  1  to  do
          T  schedule[t]
          if T=0 then return current
          next  a randomly selected successor of current
          E  V  ALUE[next] - VALUE[current]
          if E > 0  then current  next
          else current  next only with probability e  E/T
    end
end
```

Wie vielleicht schon der eine oder andere erkannt hat, der Begriff "Ausglühen/Härten" zusammen mit den Parametern E und T stehen in engem Zusammenhang mit dem (chemischen) Prozess des Erhärtens einer Flüssigkeit durch Abkühlen. Dort steht E für die gesamte Energie der Atome des Materials und T für die Temperatur.

Diese Technik wurde zuerst für VLSI-Layout-Probleme in den frühen Achzigern genutzt.
Mit verschiedenen anderen vebesserten Iterativen Algorithmen hat man es mittlerweile geschafft, das Millionen-Könige Problem in durchschnittlich weniger als 50 Schritten zu Lösen. Ein weiterer Erfolg, der genannt werden sollte ist, daß wo man früher 3 Wochen gebraucht hat, um eine Woche Observation des Hubble-Teleskops auszuwerten, braucht man heute nur noch 10 Minuten !

4.5 Zusammenfassung

Dieses Kapitel zeigte, daß Heuristiken zur Reduzierung der Suchkosten beitragen können.
Weiterhin haben wir einige Algorithmen gesehen, die Heuristiken benutzen.

Bestes-zuerst-Suche

ist ein genereller Suchalgorithmus, wobei die Knoten mit minimalen Kosten zuerst bearbeitet werden.

Greedy-Suche

Wenn man die Kosten minimiert, ausgedrückt durch *h(n)*, bekommt man den gefräßigen (Greedy) Suchalgorithmus. Die Suchzeit ist gewöhnlich niedriger als bei uninformierten Algorithmen, aber die gefräßige Suche ist weder optimal noch abgeschlossen.

A*-Suche

Die Kombination von *g(n)* und *h(n)* durch Addition als *f(n)* kombiniert die Vorteile von uninformierter Suche und gefräßiger Suche. Wenn noch wiederholte Stationen erlaubt sind und garantiert wird, daß *h(n)* nicht zu schnell steigt erhält man die A* Suche.

Komplett, Optimal, Effizient

A* ist komplett, optimal und optimal effizient gegenüber allen optimalen Suchalgorithmen.

Zeitkomplexität

Die Zeitkomplexität von heuristischen Algorithmen hängt von der Qualität der heuristischen Funktion ab. Gute Heuristiken können manchmal dadurch hergestellt werden, daß man die Problem-Definition genau untersucht, oder mit viel Erfahrung über das Problem an die Arbeit herangeht.

IDA*, SMA*

Mit speichergebundenen Algorithmen wie z.B. IDA* oder SMA* kann man den Speicherverbrauch von A* stark reduzieren.

Iterative Verbesserungen

Iterativ verbesserte Algorithmen halten nur einen einzigen Status im Speicher, können aber bei lokalen Maximas stecken bleiben. Simmuliertes Härten schafft einen Weg, dieses lokale Maximum zu umgehen.

4.6 Historisches

1966: heuristische Suchmethoden

Doran und Michie experimentieren schon 1966 mit heuristischen Such-Methoden für viele Probleme, besonders mit 8-Puzzle und 15-Puzzle Problemen.

1968: A*-Suche

A* Suche, ein Suchalgorithmus, der die aktuelle Pfadlänge mitbeachtet, wurde 1968 von Hart, Nilsson und Raphael entwickelt. 1972 hat dann Hart diesen Algorithmus nochmals verbessert.

1984: Pfadmaximierung

1984 wurde das erste mal die Methode der Pfadmaximierung von Mero verwendet.

1970: Problem des Handlungsreisenden

Hold und Karp benutzten 1970 die minimal-aufspannender-Baum Heuristik um das Problem des Handlungsreisenden zu Lösen. Sie zeigten dabei, wie solche zulässigen Heuristiken durch entspannte Probleme gelößt werden können.

1993: entspannte Probleme

1993 wurden zum ersten mal effektive neue Heuristiken durch entspannte Probleme von Prieditis entwickelt.

1950-60: Speichergebundene Suche

Weil in den späten 50ern und frühen 60ern die Rechner noch sehr wenig Speicher hatten, wurde die Hauptforschung auf die Speichergebundene Suche konzentriert. IDA* war dabei der erste oft genutzte optimale Speichergebundene heuristische Suchalgorithmus. Von ihm wurden noch viele verschiedene Varianten entwickelt.

1992: SMA*

Der erwähnte SMA* Algorithmus wurde erstmals von Russel 1992 verwendet und basiert auf dem 1989 eingeführten MA*.

1983: Simuliertes Härten

Simuliertes Härten wurde das erste mal von KirkPatrick, Gelatt und Vecchi 1983 beschrieben.

Das Thema der Parallelen Suche wurde in diesem Kapitel des Buches nicht erwähnt, z.T. auch weil dies eine größere Abhandlung über Parallele Architektur erfordern würde. Wenn Parallele-Computer mal in genügender Stückzahl vorhanden sind, wird parallele Suche ein wichtiger Bestandteil der KI werden.
Noch zu erwähnen wäre, daß neue Resultate von Suchalgorithmen regelmäßig in der Zeitung “Artificial Intelligence” erscheinen.

```
function HILL-CLIMBING( problem ) returns a solution state
    inputs:    problem, a problem
    static:    current, a node
               next, a node

    current  M   AKE-NODE( INITIAL-STATE[problem] )
    loop do
        next  a highest-valued successor of current
        if VALUE[next] < VALUE[current] then return current
        current  next
    end
end

function SIMULATED-ANNEALING( problem, schedule ) returns a solution state
    inputs:    problem, a problem
               schedule, a mapping from time to "temperature"
    static:    current, a node
               next, a node
               T, a "temperature" controlling the probatility of downward steps

    current  M   AKE-NODE( INITIAL-STATE[problem] )
    for t  1   to     do
        T  schedule[t]
        if T=0 then return current
        next  a randomly selected successor of current
        E  V    ALUE[next] - VALUE[current]
        if E > 0   then current  next
        else current  next only with probability e  E/T
    end
end
```