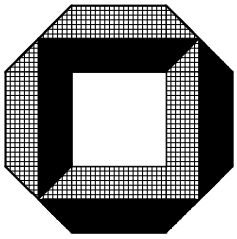
Proseminar

Künstliche Intelligenz



Universität Karlsruhe (TH)
Fakultät für Informatik
Institut für Algorithmen und Kognitive Systeme

Prof. Dr. J. Calmet
Dipl.-Inform. A. Daemi

Wintersemester 2003/2004

Institut für Algorithmen und Kognitive Systeme
Fakultät für Informatik
Universität Karlsruhe
Am Fasanengarten 5
76 128 Karlsruhe

Allgemeine und Heuristische Suchverfahren

Henning
Clemens Lode

Inhaltsverzeichnis

Kapitel 1

Allgemeine und Heuristische Suchverfahren

1.1 Einleitung

Neben der Suche nach ueberhaupt einer Loesung, geht es in diesem Vortrag vor allem um das Suchen nach einer moeglichst optimalen oder gar der optimalen Loesung. Die meisten Probleme sind komplexer Natur und somit nicht eindimensional, also muessen wir zuerst eine Abbildung der jeweiligen Loesung auf N oder R finden um entscheiden zu koennen, welche der betrachteten Loesungen besser als eine andere ist.

1.2 Allgemeine Suchverfahren

1.3 Heuristische Suchverfahren

1.3.1 Unterschiede zu allgemeinen Suchverfahren

Informierte Suchverfahren basieren auf Heuristiken, besitzen also manuell einprogrammiertes problemspezifisches Wissen. Dabei handelt es sich um Algorithmen die in vielen Faellen eine bessere durchschnittliche Suchdauer besitzen als Algorithmen uninformierter Suchverfahren, waehrend sie, zumindest die vollstaendigen Algorithmen, ebenfalls den schlechtesten Zeitaufwand von $O(b^m)$ besitzen.

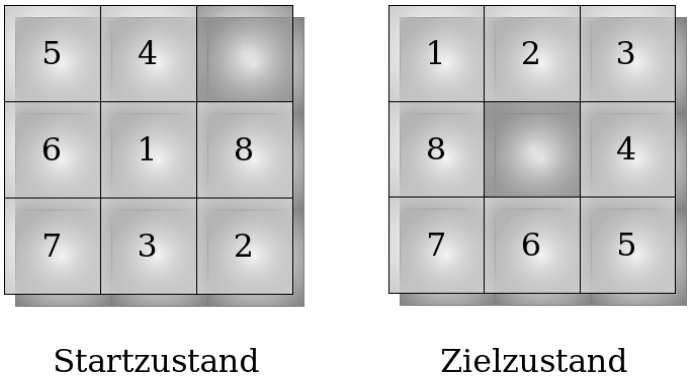


Abbildung 1.1: Beispiel zum "8-Puzzle"Problem

1.3.2 Gruppen von heuristischen Suchverfahren

Man kann die Algorithmen in 2 Gruppen einteilen. Vertreter der ersten Gruppe setzen sich Schritt fuer Schritt die Loesung zusammen, Vertreter der zweiten Gruppe betrachten fertige, nicht unbedingt optimale, Loesungen und versuchen diese iterativ zu verbessern. Die Grundidee bei beiden Gruppen ist, dass man jeweils alternative Schritte bzw. Loesungsmoeglichkeiten vergleicht und jeweils moeglichst die waehlt, die auch zu einer optimalen Loesung fuehren. Dazu waehlt man eine nicht-negative Heuristikfunktion h, die einem Zustand n einen Wert zuweist. Je niedriger h(n), desto besser ist der Zustand n. h(n)=0 bedeutet, dass n die optimale Loesung darstellt. Ausserdem benoetigen wir eine Auswahlfunktion i, die uns einen neuen Zustand, also ein Loesungsschritt der bzw. eine Loesungsmoeglichkeit die wenn moeglich noch nicht betrachtet wurde, zurueckgibt. Das Problem zu jeder Aufgabenstellung ist also erst einmal das Finden einer Funktion h und i die das bewerkstelligen. Eine der Aufgabenstellung nicht angepasste Heuristik und Auswahlfunktion fuehren zu laengerer Laufzeit oder gar zu Sackgassen und Schleifen.

1.3.3 Finden einer heuristischen Funktion

Um eine Funktion h zu finden, die den Suchvorgang moeglichst verkuerzt, bietet es sich an, ein sogenanntes "Relaxed Problem", also ein vereinfachtes Problem zu betrachten, bei dem bestimmte Beschraenkungen in der Loesungsschrittwahl aufgehoben sind. Damit erhalten wir eine moegliche Bewertung eines Loesungsschritt, wie weit wir von der optimalen Loesung entfernt sind.

Als Beispiel betrachten wir das sogenannte "8-puzzle"[2] Problem:

Beim "8-puzzle"Problem ist das Ziel, die Teile so zu bewegen, dass man vom Startzustand ausgehend den Zielzustand erreicht. Die Beschraenkung ist hierbei, dass man immer nur 1 Teil bewegen darf und sich niemals 2 Teile auf einem Feld befinden duerfen.

Da sich in jedem Schritt 2 (freies Feld in der Ecke), 3 (freies Feld am Rand) oder 4 (freies Feld in der Mitte) Teile bewegen duerfen, wuerde eine komplette

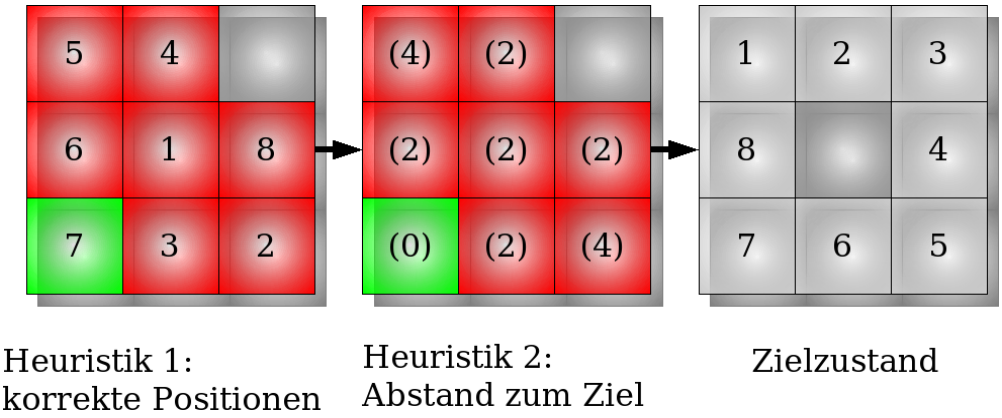


Abbildung 1.2: 2 moegliche Heuristiken zur Loesung von "8-puzzle"

Suche aller Moeglichkeiten mit Zugtiefe 20 zu 3.5*10$\hat{9}$ oder, ignoriert man sich wiederholende Situationen, zu 9! Zustaende fuehren.[3]

Wie stellen wir nun fest, ob wir uns beim Bewegen eines Teiles auf die optimale Loesung zu bewegen oder uns entfernen? Beim Ursprungsproblem koennen wir nur feststellen, ob ein Zustand dem Zielzustand entspricht oder nicht. Wir muessen also ein vereinfachtes, sogenanntes "relaxed problem- finden, so dass Ueberschreitungen von Beschraenkungen nicht zu einem Ignorieren des Loesungsschritt bzw. der Loesung fuehrt.

Zuerst listen wir die Beschraenkungen noch einmal getrennt auf:

Ein Teil darf

1. pro Schritt nur 1 Feld und

2. nicht schraeg und

3. nur in ein freies Feld

verschoben werden

Heben wir in diesem Fall die Beschraenkung 3 einfach auf, duerfen sich also Teile auch auf besetze Felder bewegen, wissen wir, dass die Summe der Schritte der Teile von ihren aktuellen Positionen zu den Zielpositionen die minimale Zahl der Schritte ist, die wir zur optimalen Loesung benoetigen. In diesem speziellen Fall wird das Manhattan distance genannt [4, Seite 102] Hebt man zusaetzlich noch Beschraenkung 1 auf, erhaelt man eine Heuristik, die die Zahl der Teile die sich an falscher Position befinden beschreibt.

Nach [5, 102] ist die erste Heuristik der zweiten Heuristik mit weniger Beschraenkungen ueberlegen, fuehrt also zu einer Suche mit geringerer durchschnittlichen Suchdauer.

Prueft man alle Kombinationen von Beschraenkungsaufhebungen erreicht man so eine fuer das jeweilige Problem optimale Loesung. Diesen Weg beschreitet ABSOLVER [6,103]

1.3.4 Formale Betrachtung des Problems

Hat man nun eine heuristische Funktion gefunden kann man diese nun in einer Suche verwenden

Das Grundgeruest der informellen Suche sieht erst einmal so aus:

```
Informierte Suche(Zustand alterZustand)
Solange erstelleNeuenZustand(alterZustand,Bewertung) nicht NULL
        Zustand neuerZustand = erstelleNeuenZustand(alterZustand,Bewertung)
liefere als Ergebnis (alterZustand)
```

SZustandïst je nach Problemdarstellung unterschiedlich, im Weiteren wird hier nur auf Graphendarstellungen eingegangen, da sich die meisten Probleme problemlos auf einen Graphen transformieren lassen koennen. Je nach Algorithmus werden hier auch pro Knoten unterschiedlich viele Daten gespeichert. "Bewertungßtellt unsere Heuristikfunktion h dar. Sie ist meistens vom Datentyp INTEGER, kann aber jede beliebige total geordnete Menge sein. Je nach Algorithmus muss sie speziellen Anforderungen genuegen. ërstelleNeuenZustandëntspricht unserer Funktion i, sie akzeptiert zum einen den momentanen Bearbeitungszustand und zum anderen die Heuristikfunktion "Bewertung". Auf einige Beispiele fuer moegliche h und i Funktionen wird nun in den naechsten beiden Abschnitten naeher eingegangen.

1.4 Algorithmen der 1. Gruppe

1.4.1 Greedy Search

Bei dem sogenannten Greedy Search waehlt die Funktion i den Loesungsschritt, von dem in der Funktion uebergebenen Zustand älterZustand", bei dem der Startknoten geoeffnet ist, moeglichen Loesungsschritten, die den geringsten Wert fuer h ausweisen. Es werden also nacheinander Knoten geoeffnet und die Bewertung aller Soehne des jeweiligen Knotens miteinander verglichen. Der Knoten mit geringster Bewertung wird ausgewaehlt und geoeffnet usw.

```
Gierige Suche(Zustand alterZustand)

Solange erstelleNeuenZustand(alterZustand,Bewertung) nicht NULL
Zustand neuerZustand = erstelleNeuenZustand(alterZustand,Bewertung)
liefere als Ergebnis (alterZustand)

Zustand erstelleNeuenZustand(Zustand momentanerZustand,Bewertungsfunktion)
{
        Knoten n = min(Bewertungsfunktion(geoeffnete Knoten von momentaner Zustand))
        momentanerZustand = SchliesseAlleKnoten(momentanerZustand) ~~
        Falls oeffneKnoten(momentanerZustand,n) ungleich momentanerZustand
                liefere als Ergebnis oeffneKnoten(momentanerZustand,n)
}
```

Kommentare Speicherverbrauch?? Problem: am besten das mit dem Gedaechtnis rausnehmen... weil das kommt ja bei A* eh...

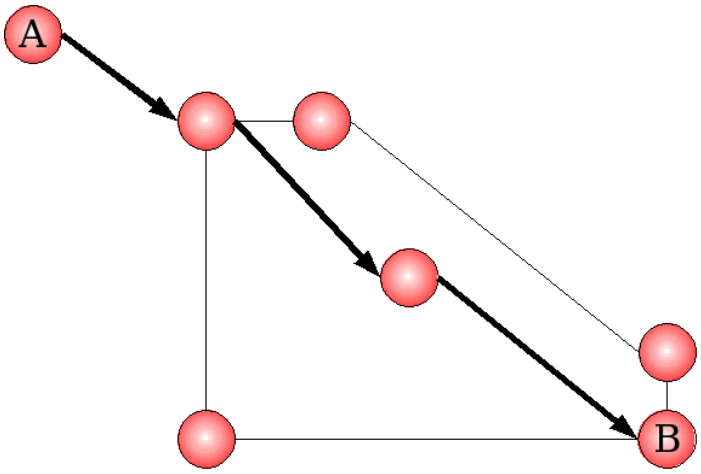


Abbildung 1.3: "Greedy-Searchïn Aktion

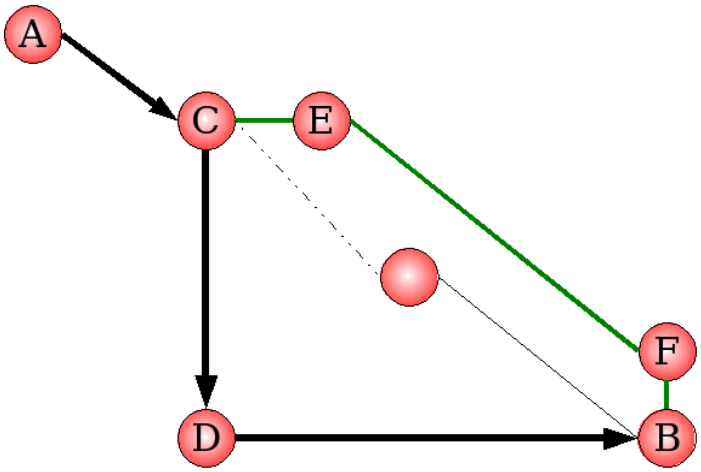


Abbildung 1.4: "Greedy-Searchïst anfaellig gegenueber falschen Starts

Will man nun mit Greedy Search z.B. den kuerzesten Pfad in einem Graphen finden, bietet sich fuer h die einfache Distanz ueber die Luftlinie an.

Also z.B. hier:

Leider hat der Algorithmus ein paar Nachteile:

Der Algorithmus ist anfaellig gegenueber falschen Starts:

Und Sackgassen.

Oder sogar Schleifen.

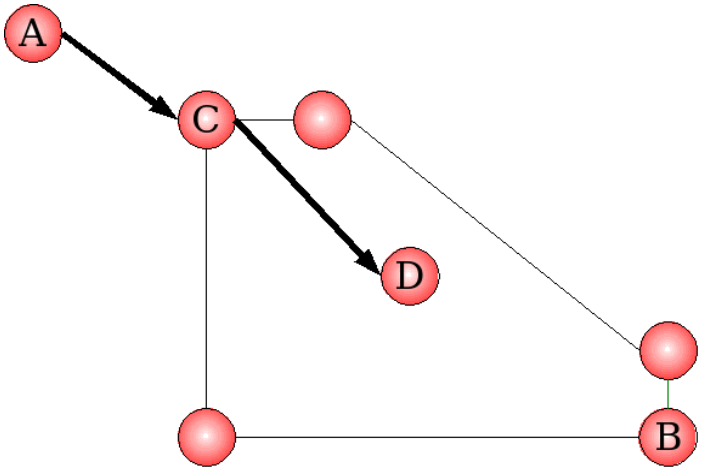


Abbildung 1.5: "Greedy-Searchïst anfaellig gegenueber Sackgassen

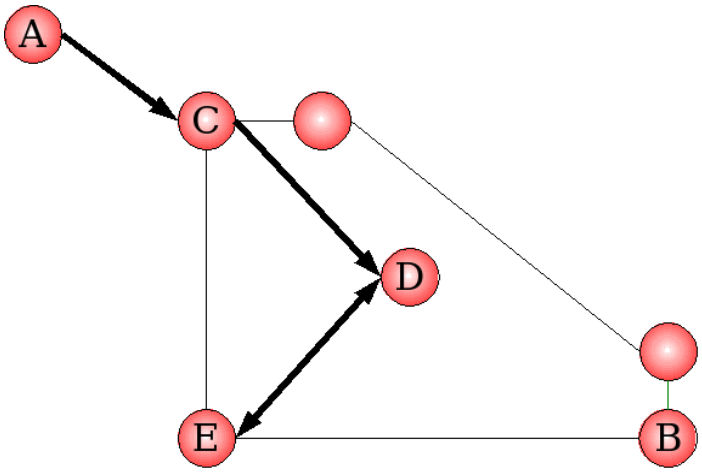


Abbildung 1.6: "Greedy-Searchïst anfaellig gegenueber Schleifen

Greedy Search hat einen Zeitaufwand von O(b^m) im schlechtesten Fall, fuehrt jedoch in vielen Faellen zu einem guten und schnellen Ergebnis. Wie soeben gesehen ist der GREEDY SEARCH Algorithmus jedoch nicht optimal. Er ist auch nicht vollstaendig, wenn man nicht auf Sackgassen und Schleifen prueft, wie im ersten Teil gezeigt. Leider steigt dann der Speicheraufwand von O(m) auf O(b^m). [7]

1.4.2 Verbesserungen von GREEDY SEARCH: A*

Erweitert man die Funktion i, dass nicht nur die Kinder des gerade geoeffneten sondern aller der noch nicht besuchten Knoten betrachtet erhalten wir einen vollstaendigen Algorithmus Dadurch treten nun keine Schleifen und Sackgassen mehr auf, es wird einfach an die naechste passende Stelle gesprungen. Gedaechtnis rausnehmen also naja...

```
A* Suche(Zustand alterZustand)
Solange erstelleNeuenZustand(alterZustand,Bewertung) nicht NULL
        Zustand neuerZustand = erstelleNeuenZustand(alterZustand,Bewertung)
liefere als Ergebnis (alterZustand)

Zustand erstelleNeuenZustand(Zustand momentanerZustand,Bewertungsfunktion)
{
        Knoten n = min(Bewertungsfunktion(geoeffnete Knoten von momentaner Zustand))
        Falls oeffneKnoten(momentanerZustand,n) ungleich momentanerZustand
        liefere als Ergebnis oeffneKnoten(momentanerZustand,n)
}
```

Leider ist dieser Algorithmus immer noch anfaellig gegenueber einem unguenstigen Start und in vielen Faellen nicht optimal. Deshalb hat man den sogenannten A* Algorithmus entwickelt, der die gleiche i Funktion aufweist, dessen Bewertungsfunktion h aber eine Kombination aus den bereits bekannten greedy searchs und uniform-cost searchs ist.

Hier ist also h(n) = Enternung von n zum Ziel (Luftlinie) + Kantenlaenge von Start bis n

Bemerkung: Man kann h auch beliebig anders waehlen, A* bleibt aber nur solange optimal wie die Heuristik h der Dreiecksgleichung genuegt, d.h. zu einer gegebenen Heuristik h und einer direkten Verbindung zweier Punkte A und B darf es keine andere Verbindung zwischen A und B mit einem Knoten C geben, bei der h(B ueber C)¡h(B ueber A) gilt. Eine genauere Beschreibung und eine Moeglichkeit derartige heuristische Funktionen zu korrigieren findet man in der Literatur [ ].

¡BILD mit Dreiecksgleichung¿

Untersuchungen erbrachten die Ergebnisse, dass A* optimal und vollstaendig ist und auch, dass es keinen anderen optimalen und vollstaendigen Algorithmus gibt, der (fuer eine beliebige heuristische Funktion) die Aufgabe mit Expandierung von weniger Knoten schafft. [?wo denn]

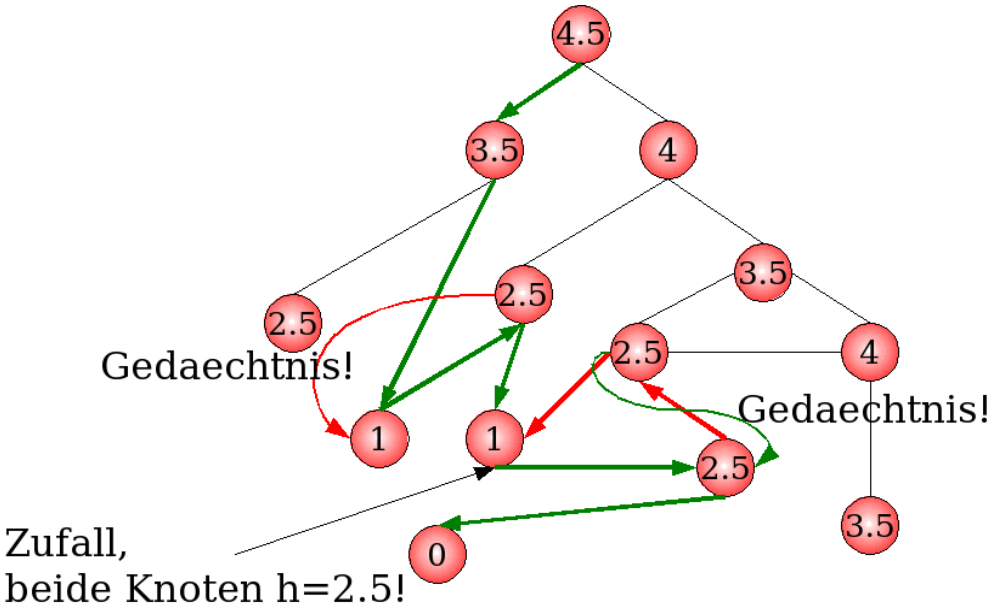


Abbildung 1.7: "Greedy-Search"mit Gedaechtnis

Im Folgenden wird nun anhand des bekannten Beispiels die Funktionsweise von A* demonstriert. Zahlen in den roten Knoten bedeuten wie vorher auch die Distanz mittels Luftlinie, in den gruenen Knoten der Wert der heuristischen Funktion, bisherige Kantenlaenge + Distanz Luftlinie. Ein * markiert einen Knoten, fuer den ein besserer Weg gefunden wurde.

A* hat in seiner Grundform ebenfalls einen Zeit und Speicheraufwand von $O(b^m)$, da wie beim uniform cost search alle geoeffneten Knoten im Speicher liegen und im schlechtesten Fall auch alle Knoten untersucht werden muessen. Fuer b = 2 und m = 40 benoetigt ein Aldi-Rechner fuer A* im schlechtesten Fall vielleicht eine Stunde. Der Speicherbedarf dagegen ist im schlechtesten Fall dagegen mit 1024 GB jenseits von gut und boese. Der Speicheraufwand laesst sich jedoch mit einem etwas abgewandelten Algorithmus reduzieren.

Zum Glueck gibt es derart viele verschiedene, hier werde ich zwei vorstellen:

1.4.3 IDA*

IDA* funktioniert im Grunde wie der bereits betrachtete iterative deepening search, statt Knotentiefe als Grenze wird hier aber ein Kostenlimit gesetzt. Es werden also immer nur ein Bereich aus Knoten untersucht, deren Bewertungsfunktion h kleiner als das Kostenlimit ist. Der Speicherersparnis kommt natuerlich fuer den Preis der Geschwindigkeit. Zwar wird nur IDA* funktioniert genau wie A* auch, es wird aber in mehreren Schritten jeweils immer nur ein Teil des Graphen betrachtet, und zwar der Teil, dessen Knoten eine geringere Qualitaet f besitzen, als ein vorgegebener Wert, der immer weiter erhoeht wird, bis das Ziel gefunden wurde.

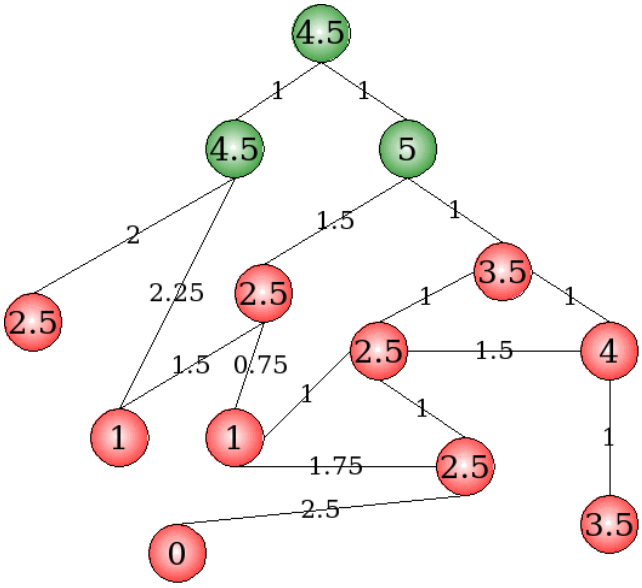


Abbildung 1.8: Ä* Search”

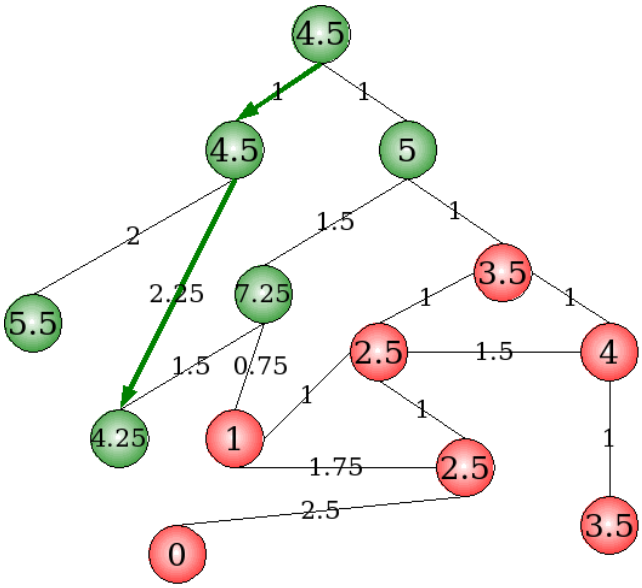


Abbildung 1.9: Ä* Search”

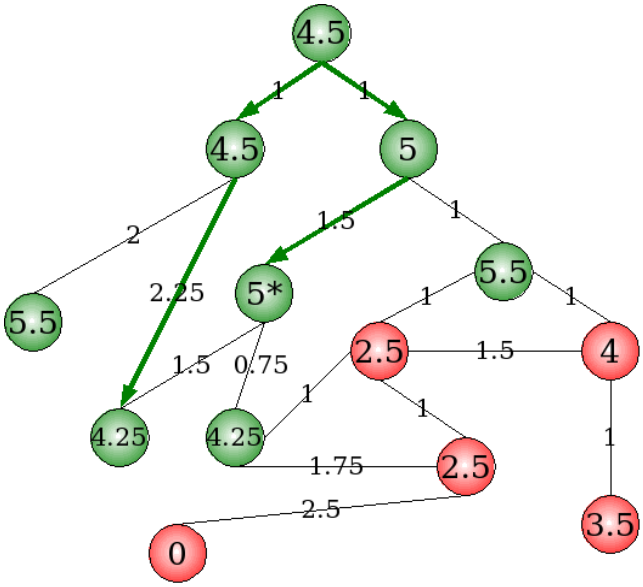


Abbildung 1.10: Ä* Search”

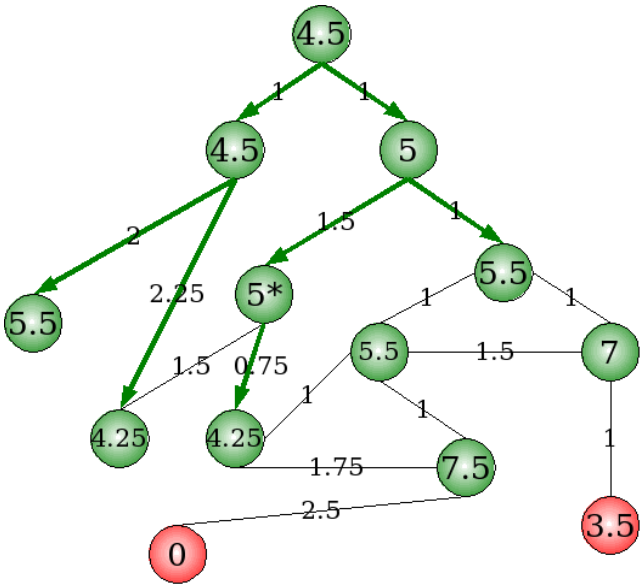


Abbildung 1.11: Ä* Search”

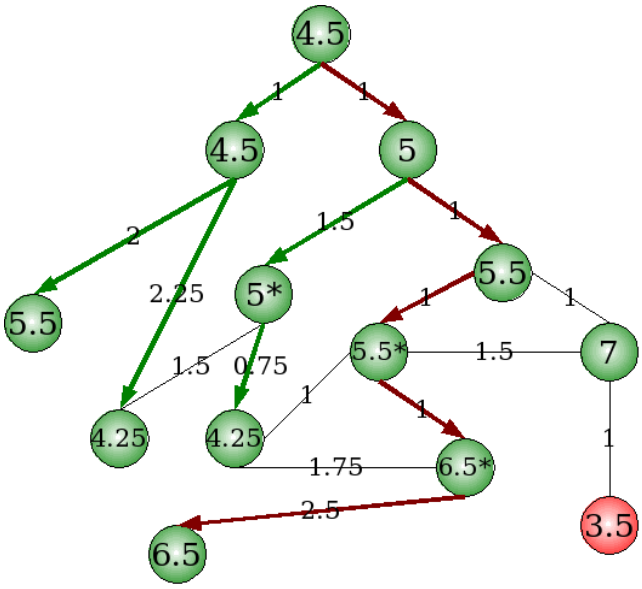


Abbildung 1.12: Ä* Search”

1.4.4 SMA

SMA verbraucht keine feste Speichermengenge, sondern passt sich an den verfuegbaren Speicherplatz an, versucht also so viele Knoten wie moeglich zu speichern. IMA arbeitet genauso wie A* wenn beliebig viel Speicherplatz zur Verfuegung steht.

1.4.5 CSP

1.5 Algorithmen der 2. Gruppe

Optimal und vollstaendige Algorithmen sind schoen und gut, in der Praxis sind die Probleme jedoch meist zu komplex oder es liegen sowieso keine gesicherten Werte vor, so dass es oft schon reicht, wenn nicht die optimale Loesung sondern nur z.B. eine 5% am Optimum liegende Loesung gefunden werden kann. Diese Aufgabe koennen die Algorithmen der zweiten Gruppe sehr erfolgreich loesen. Wie anfangs erwaehnt werden bei diesen Algorithmen die Loesung nicht Schritt fuer Schritt aufgebaut sondern fertige Loesungen generiert und diese iterativ verbessert. Wichtig ist dabei, dass die Aufgabenstellung so umgeschrieben wird, dass jede Art von Ergebnis gueltig ist, also einer bestimmten Qualitaetsstufe, der sogenannten Fitness, zugeordnet werden kann. Dazu werden harte Nebenbedingungen, also z.B. Lagerueberschreitungen, Bargelduebersch reitungen, Zeitueberschreitungen etc. mit Strafkosten belegt, also die Fitness gesenkt, anstatt dass die Loesung ganz verworfen wird. Der grosse Unterschied zur ersten Gruppe ist aber, dass die neuen Loesungen mehr oder weniger zufaellig generiert werden. Dadurch ergibt sich das Problem, dass man nie genau weiss, ob

Abbildung 1.13: 3 Bergsteiger auf der Suche nach dem Gipfel

bei weiteren Durchlaeufen bessere Loesungen gewonnen werden koennen oder nicht.

Grundsaetzlich arbeiten Algorithmen der 2. Gruppe wie folgt:

```
basic iterative improvement algorithm()
Loesung alteLoesung = erstelleZufaelligeLoesung()
Wiederhole
        Loesung neueLoesung = i( alteLoesung )
        Differenz = h( neueLoesung) - h( alteLoesung )
Falls selektion( T, Differenz ) == true
                Dann alteLoesung = neueLoesung
bis alteLoesung ausreichend gut
liefere als Ergebnis alteLoesung
```

Hier sind 2 neue Funktionen hinzugekommen, die jedoch nicht weiter kompliziert sind. ërstelleZufaelligeLoesungërstellt wie der Name schon sagt, eine zufaellige Startloesung, die dann schrittweise verbessert werden soll. ßelektion"prueft in Abhaengigkeit einer Variablen T, ob "Differenzäusreichend klein ist. Auf einige Beispiele fuer h, i und selektion moechte ich hier eingehen:

1.5.1 Hill-Climbing Algorithmus

Der einfachste darauf basierende Algorithmus ist der sogenannte Hill-Climbing Algorithmus. Der Name ruehrt von dem anschaulichen Problembeispiel her, dass man sich alleine im Gebirge befindet, im Nebel nichts sehen kann, aufgrund Sauerstoffknappheit sich nicht an seine letzten Schritte erinnern kann und als Ziel hat, den hoechsten Berggipfel zu erreichen. Eine moegliche Strategie waere, dass man sich vortastet, ob man denn mit dem naechsten Schritt etwas hoeher kommt oder nicht.

1.5.2 Probleme des Hill-Climbing Algorithmus

Bei dem Algorithmus ergeben sich aber zwei schwerwiegende Probleme:

1. der Algorithmus wird mit hoher Wahrscheinlichkeit an einem lokalen Optimum haengenbleiben

Dies laesst sich beheben, indem man den Algorithmus nach einiger Zeit ohne Verbesserung einfach mit einer anderen Startkonfiguration/position neu beginnt, die beste Fitness der bisherigen Durchlaeufe aber speichert. Dann nennt man es einen Random-Restart Hillclimbing Algorithmus. Problem bleibt hier festzustellen, ob wir denn in einem lokalen Optimum oder einer Kante stecken oder nicht schon kurz vor dem Ziel sind.

¡BILD20!¿

2. ist die Fitnesslandschaft sehr "wildälso mit grossen Gradientenunterschieden besetzt

Hier hilft auch kein zufaelliges Neustarten mehr, man wird wahrscheinlich fast alle Loesungsmoeglichkeiten durchsuchen muessen, der Algorithmus ist also nicht viel besser als eine einfache zufaellige Suche ueber dem Loesungsraum. Die Hauptschwierigkeit beim Hillclimber ist also nicht die Suche selbst sondern die Gestaltung der Problemstellung, so dass eine Fitnesslandschaft mit moeglichst sanften Steigungen entsteht.

¡BILD22!¿

/*Eine weitere Variante des Hillclimbers waere, anstatt die absoluten Werte der Loesungen zu vergleichen, die Gradienten herzunehmen. Bei praktischen Problemen mit hunderten von Variablen erweist sich dies aber als nicht praktikabel, da man hierzu die Ableitung der Fitnessfunktion bestimmen muesste. Und wer will schon eine Funktion mit Schleifen, Zufallswerten, Rekursionen und Spruengen differenzieren... :-) */

1.5.3 Simulated Annealing Algorithmus

Aehnlich funktioniert auch das sogenannte Simulated Annealing, das urspruenglich aus der Physik kommt. Wird eine bessere Loesung gefunden, wird sie wie beim Hillclimber auf jeden Fall weiterverwendet, wird eine schlechtere Loesung gefunden, wird sie nicht sofort verworfen, sondern mit einer Wahrscheinlichkeit die der Parameter T angibt weiterverwendet. Ueber die Generationen wird T dabei immer weiter verringert, bis fuer T=0 (hoffentlich) eine ziemlich gute Loesung gefunden wurde.

¡BILD, Beispiel Transportstrecke zwischen Staedte¿

O————O — — — — — — — — o————o

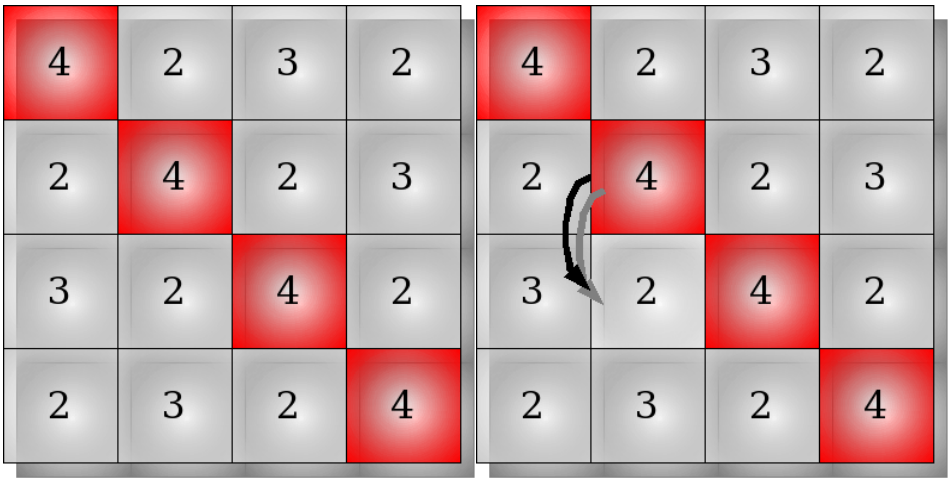


Abbildung 1.14: "CSP"mittels Heuristik des minimalen Konflikts

1.5.4 Anwendungen in CSPs : Heuristik des minimalen Konflikts

Heuristische Algorithmen der 2. Gruppe eignen sich sehr gut, um CSP Probleme wie z.B. das n-Damenproblem zu loesen. Mit der nun beschriebenen Heuristik kann z.B. das Millionen-Damen Problem im Durschnitt mit weniger als 50 Schritten geloest werden. [5]

Wieder ist der Trick, nicht perfekte Loesungen zu suchen, sondern eine zufaellige zu generieren diese Schritt fuer Schritt zu "reparieren"(heuristic repair). Das Optimierungsproblem zu n-Dameproblem lautet "Finde eine Position in der moeglichst wenige Damen von moeglichst wenigen anderen Damen geschlagen werden koennen".

Hier nun zur Uebersicht ein Beispiel zum 4 Dameproblem in 5 Schritten:

Waehrend die roten Felder die Positionen der 4 Damen darstellen, geben die Zahlen den Wert der Heuristik an. Die Heuristik ist eine sogenannte "min conflicts"[6] Heuristik, die jeweils angibt, wieviele Bedingungen bei einer Konfiguration ueberschritten wird. In diesem Fall entspricht dies der Zahl der Damen, die das Feld angreifen, bei roten Feldern inklusive dieser Dame selbst. Bei jedem Reperaturschritt der ungueltigen Loesung wird nun fuer eine Dame einer zufaelligen Spalte ein zufaellig Feld mit niedrigerer oder gleicher Bewertung gewaehlt.

In diesem Beispiel stehen alle 4 Damen in einer Diagonalen, also koennen z.B. in der Diagonalen 4 Damen jeweils ein Feld angreifen bzw. stehen darauf . Hier wurde die 2. Spalte gewaehlt und die Dame auf das niedriegere Feld in der 3. Zeile verschoben. Dass diese Heuristik scheinbar funktioniert, kann man auch daran erkennen, dass die Gesamtzahl der angegriffenen Damen sich von 16 nun auf 8 verringert hat:

Im ersten Bild ist hier die Wahl der ersten Spalte zwingend, mit keiner anderen Dame findet man ein besseres Feld. In der Situation im zweiten Bild wuerde der Hill-Climber sich auf einem Plateau befinden und waere nicht besser

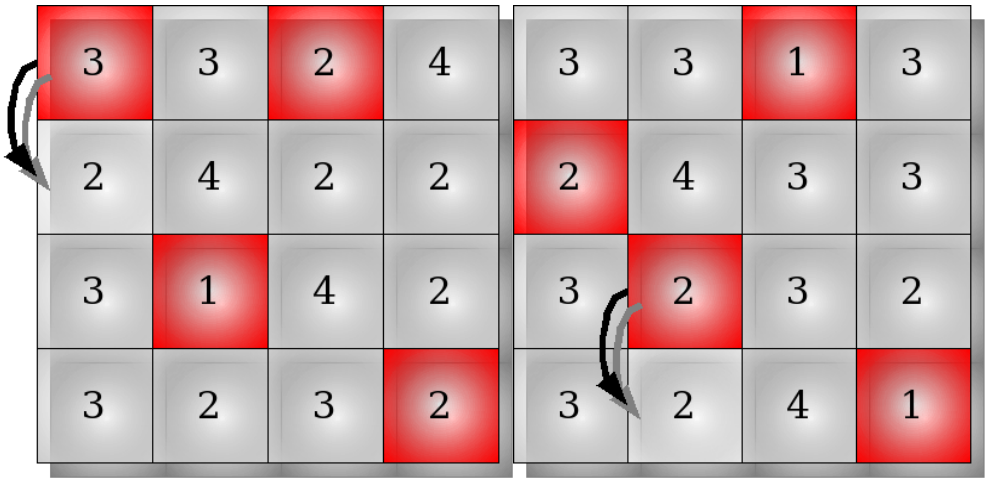


Abbildung 1.15: "CSP"mittels Heuristik des minimalen Konflikts

wie eine zufaellige Suche. Man koennte hier anstatt nur die einzelnen Felder im Ursprungszustand betrachten, die Gesamtzahl der sich ueberschneidenden Damen eintragen, was aber mit einem Aufwand von O(n Damen*n*n Felder) verbunden waere . Beweis?

Im letzten Bild gibt es keine Verbesserung und der Hill-Climber laeuft ohne Verbesserung bis zum Abbruch ohne Veraenderung weiter. Da wir aber das Optimum, d.h. es gibt eine Loesung fuer das n-Damenproblem mit keiner Ueberschneidung, schon kennen und es hier auch abzaehlen koennen, sind wir fertig.

1.5.5 Verfeinerung des Hill-Climbing Algorithmus

Es gibt einige Verfeinerungen des Hill-climbing Algorithmus, besonders erwaehnenswert sind Genetische Algorithmen/Programmierung und Evolutionaere Algorithmen.

Diese finden z.B. Anwendung in komplexen praktischen Problemstellungen wie z.B. Betriebsplanoptimierung an der schon laenger SAP erfolgreich arbeitet.

BAECKER!!

neurale netzwerke Algorithmen der zweiten Gruppe sind, sofern man nicht den Weg ueber neurale Netzwerke begeht, nur bei bestimmten Problemen einsetzbar. Schachspielen ist z.B. nicht moeglich, da dort ja nicht nur ein Spielablauf optimiert werden soll, sondern moeglichst jede Antwort auf jeden Zug ‘

Grob gesagt handelt es sich bei GAs um Hillclimber mit einer Breitensuche in Form von einer Population von Ägenten", mit einer Trennung von Genotyp, also der tatsaechlichen Genstruktur und Phaenotyp, also dem entgueltigen Erscheinungsbild, mit einer Art Gedaechtnis in Form von dominant/rezessiver Gene und inaktiver Gene, sogenannter Introne, die nicht tatsaechlich in einen Phaenotyp codiert werden, sondern nur als Mutationsschutz dienen und mit einer Rekombination durch Crossing Over zwischen den Agenten.

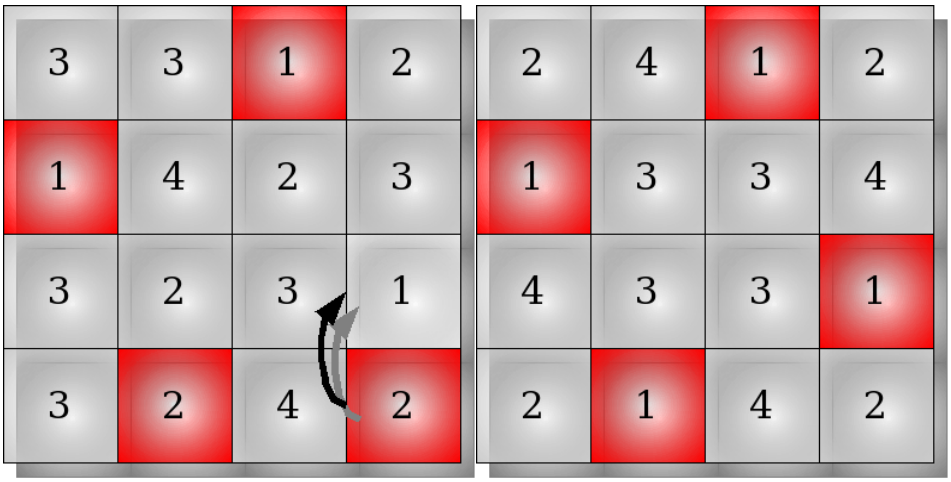


Abbildung 1.16: "CSP"mittels Heuristik des minimalen Konflikts

Genauer kann ich leider nicht darauf eingehen, das wuerde den Rahmen des Vortrags sprengen, wer sich damit etwas beschaefitgen will, dem empfehle ich das Buch "Genetic Programming - An introduction". Sie sind sehr erfolgreich, verglichen mit anderen Suchstrategien, denn welch anderer Algorithmus kann schon Vortraege ueber sich selbst halten?

Bevor man sich ueberhaupt Gedanken machen kann, welchen Suchalgorithmus man auf ein spezielles Problem anwendet, ist es unabdingbar, dass man sich ueberlegt, wie man 2 Loesungen vergleicht. Dies stellt die Zielfunktion dar, die entweder maximiert, wie z.B. bei Gewinnspannen oder minimiert, wie z.B. bei Entfernungen werden soll.

Zwar fuehrt das Absuchen des kompletten Loesungsraums wie im 1. Teil besprochen garantiert zur optimalen Loesung, jedoch sind die meisten Probleme einfach zu komplex um sie derartig auszurechnen. Als Beispiel waere Schach genannt, bei der eine Partie oft um die 40 Zuege dauert. Da der Computer sowohl den optimalen eigenen, als auch den optimalen gegnerischen Zug berechnen muesste, waeren das 10^{80} Stellungen die miteinander verglichen werden muessten.

Bei heuristischen Suchverfahren verlaesst man sich deshalb nicht auf das sichere Ereignis, wie z.B. "Mit diesem Zug gewinne ich die Partie nach 40 Zuegen, sofern der Gegner optimal spielt", sondern mit gewissen Annahmen die fuer jedes Problem einzeln ueberlegt werden muessen. Um auf das Schachbeispiel zurueckzukommen: Moderne Schachcomputer rechnen nicht jede Zugkombination bis zum Ende der Partie durch, sondern vergleichen schon nach wenigen Zuegen die Zahl und Position der Figuren um die Stellung bewerten zu koennen und auf ein eindimensionales Ergebnis zu kommen. Das Wissen um die Qualitaet einer Stellung im Schach wurde aber nicht berechnet sondern mehr oder weniger durch Versuch und Irrtum bestimmt.

Betrachtet man hierbei nicht alle moeglichen

1.6 Zusammenfassung

1.6.1 Allgemeine Suchalgorithmen

1.6.2 Heuristische Suchalgorithmen

Insgesamt hat man im zweiten Teil gesehen, dass wir die Zahl der zu untersuchenden Moeglichkeiten durch Heuristiken stark reduzieren koennen. Zwar bieten die Heuristiken keine Garantie fuer eine schnellere Ausfuehrung, die schlechteste Laufzeit betraegt nach wie vor $O(b^m)$, im praktischen Anwendungsfall mit einer guten Heuristik laesst sich jedoch die durchschnittliche Suchzeit stark reduzieren.

Begonnen haben wir mit einer allgemeinen Beschreibung der heuristischen Funktion, die man sich als eine Art Orakel vorstellen kann das Informationen ausserhalb des Wahrnehmungsbereichs der generalisierten Suche aufnimmt, und die jeweilige Loesung nach ihrer bewertet.

...

Literaturverzeichnis

[1] Russel S., Norvig P.: *Artificial Intelligence – A Modern Approach*, Second Edition, Prentice Hall, 2003.

[2] Brion L. Vibber: *Maps of the World*, http://leuksman.com/misc/maps.php