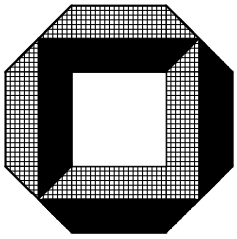
Proseminar

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 Eberhardt
Clemens Lode

Inhaltsverzeichnis

1 Allgemeine und Heuristische Suchverfahren 2

1.1 Einleitung . . . 2

1.2 Allgemeine Suchverfahren . . . 3

1.2.1 Komplexität . . . 3

1.2.2 Breitensuche (BREADTH-FIRST SEARCH): . . . 4

1.2.3 UNIFORM-COST SEARCH . . . 5

1.2.4 Tiefensuche (DEPTH-FIRST SEARCH) . . . 5

1.2.5 Begrenzte Tiefensuche (DEPTH-LIMITED SEARCH) . . . 6

1.2.6 Iterativ vertiefende Tiefensuche (ITERATIVE DEEPENING DEPTH-FIRST SEARCH) . . . 6

1.2.7 Bidirektionale Suche (BIDIRECTIONAL SEARCH) . . . 8

1.2.8 Unterbindung von Schleifen bei der Suche . . . 8

1.2.9 CONSTRAINT SATISFACTION PROBLEMS . . . 9

1.2.10 BACKTRACKING SEARCH . . . 9

1.2.11 FORWARD CHECKING . . . 10

1.3 Heuristische Suchverfahren . . . 13

1.3.1 Unterschiede zu allgemeinen Suchverfahren . . . 13

1.3.2 Gruppen von heuristischen Suchverfahren . . . 13

1.3.3 Finden einer heuristischen Funktion . . . 13

1.3.4 Formale Betrachtung des Problems . . . 15

1.4 Algorithmen 1. Gruppe - Schrittweiser Aufbau . . . 16

1.4.1 GREEDY SEARCH . . . 16

1.4.2 GREEDY SEARCH mit CSPS . . . 18

1.4.3 Verbesserungen von GREEDY SEARCH: A* . . . 19

1.4.4 ITERATIVE DEEPENING A* SEARCH . . . 24

1.4.5 SIMPLIFIED MEMORY-BOUNDED A* SEARCH . . . 24

1.5 Algorithmen der 2. Gruppe - Schrittweise Verbesserung . . . 26

1.5.1 Hill-Climbing Algorithmus 26
1.5.2 Probleme des Hill-Climbing Algorithmus 27
1.5.3 Simulated Annealing Algorithmus 28
1.5.4 Heuristik des minimalen Konflikts bei CSPs 28
1.5.5 Verbesserte Hill-Climbing Algorithmen 31
1.6 Zusammenfassung 32
1.6.1 Allgemeine Suchalgorithmen 32
1.6.2 Heuristische Suchalgorithmen 34

Kapitel 1

Allgemeine und Heuristische Suchverfahren

1.1 Einleitung

Das Suchen einer Lösung ist der zentrale Bestandteil der Informatik, ja vielleicht sogar des ganzen Lebens. Von der täglichen Suche nach den Schlüsseln mit Hilfe von Erinnerungsstrategien („Wo habe ich sie zuletzt gesehen?") bis zur Suche nach außerirdischer Intelligenz im Weltraum, immer steht man vor endlos vielen Möglichkeiten. Diese Ausarbeitung wird sich darauf beschränken, Graphenprobleme zu analysieren und verschiedene Algorithmen vorzustellen, die diese effizient lösen können.

Grundlage werden die *allgemeinen* oder auch *informierten Suchverfahren* seien, die eine Lösung schrittweise zusammensetzen, indem sie den Graphen geschickt durchschreite und jeweils prüft, ob dieser Schritt Teil der Lösung oder die Lösung selbst ist.

Anschließend werden Algorithmen erläutert, die durch Hinzunahme zusätzlicher Informationen von außen die durchschnittliche Suchdauer reduzieren. Eine Gruppe der sogenannten *heuristischen Suchverfahren* baut wie die allgemeinen Suchverfahren die Lösung schrittweise auf, während die andere fertige Lösungen erstellt und diese schrittweise optimiert. Es wird auch ein Ausblick auf moderne Suchalgorithmen, den sogenannten *Genetischen Algorithmen* gegeben, die zunehmend Anwendung in der Wirtschaft wie auch in der Wissenschaft finden.

Zusätzlich zu den Beispielen der Pfadsuche durch Graphen werden die Algorithmen anhand von sogenannter *Constraint Satisfaction Problems* erklärt und zum Teil auf deren Schwächen und Stärken eingegangen.

1.2 Allgemeine Suchverfahren

Ein universeller TREE-SEARCH Algorithmus kann zur Lösung jedes beliebigen Problems eingesetzt werden. Es gibt jedoch verschiedene Suchstrategien mit unterschiedlichen Vor- und Nachteilen. Die unterschiedlichen Algorithmen werden nach den Kriterien

- **Vollständigkeit** (*completness*)
- **Optimalität** (*optimality*)
- **Zeitkomplexität** (*time complexity*)
- **Speicherbedarf** (*space complexity*)

```
TREESEARCH(Problem, Queueing-Fn)
initialisiere Queue mit Startknoten
solange Queue ≠ leer
{
    Knoten k = Queue.pop
    falls k Lösung für Problem ist
        dann liefere als Ergebnis(Pfad zu k)
    füge weitere Knoten aus Suchbaum an durch Queueing-Fn vorgesehenen Stelle zu Queue
}
liefere als Ergebnis(Fehler)
```

Die Funktion nimmt als Parameter ein Problem und eine Queueing-Funktion. Die Queueing-Funktion entscheidet über die Reihenfolge in welcher die Knoten zum Queue hinzugefügt werden. Der Algorithmus nimmt sich mit Queue.pop den ersten Knoten aus dem Queue und prüft diesen, ob er eine Lösung ist. Das Ergebnis ist immer der Pfad zu dem Knoten, welcher das Problem löst.

1.2.1 Komplexität

Die Komplexität hängt vom Verzweigungsfaktor b(*branchingfactor*) des durchsuchten Zustandsraumes und der Tiefe d(*depth*) des Suchbaumes ab. Der Verzweigungsfaktor gibt die Anzahl der Verzweigungen pro Knoten an. Mit seiner Hilfe und der Tiefe d lässt sich also leicht eine Obergrenze für die maximale Anzahl der Knoten in einem Suchbaum angeben:

$$1 + b + b^2 + b^3 + b^4 + \cdots + b^d = \sum_{i=0}^{d} b^i$$

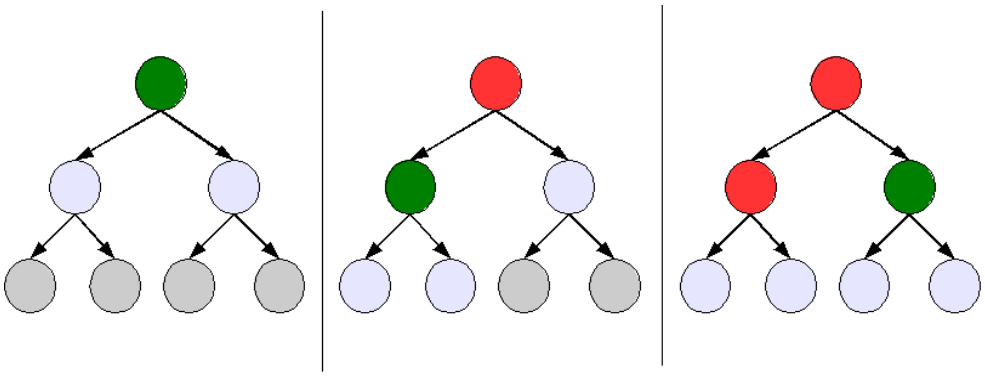


Abbildung 1.1: Rote Knoten sind bereits abgesucht und im Speicher, der Grüne wird gerade bearbeitet und die hellblauen wurden zum Queue zur Bearbeitung hinzugefügt

1.2.2 Breitensuche (breadth-first search):

Bei der Breitensuche, welche von Moore 1959 zur Lösung von Labyrinthen entwickelt wurde, werden jeweils alle Knoten mit der selben Tiefe durchsucht. Der Suchbaum wird also Ebenenweise aufgebaut (Siehe Abbildung 1.1). Dies lässt sich durch den Aufruf TREE-SEARCH (problem, ,,Kinder am Ende anfügen") bewerkstelliegen.

- **Vollständig**: Ja, sofern ein Ziel in einer endlichen Tiefe existiert und der Verzweigungsfaktor endlich ist
- **Optimal**: Ja, sofern alle Pfadkosten identisch sind
- **Zeit/Speicheraufwand**: O(b^{d+1})

Beispiel: Geg.: Suchbaum mit Knotengröße 1 kbyte und 10 Verzweigungen pro Knoten (b = 10) sowie einem Rechner, der 10.000 Knoten pro Sekunde bearbeitet.

Tiefe	**Knoten**	**Dauer**	**Speicherverbrauch**
2	1100	0.11 s	1 Mb
4	111100	11 s	106 Mb
6	10^7	19 m	10 Gb
8	10^9	31 h	1 Tb
10	10^{11}	129 d	101 Tb
12	10^{13}	35 y	10 P(eta)b
14	10^{15}	3523 y	1 E(xa)b

Beispiel aus ([1], Seite 74)

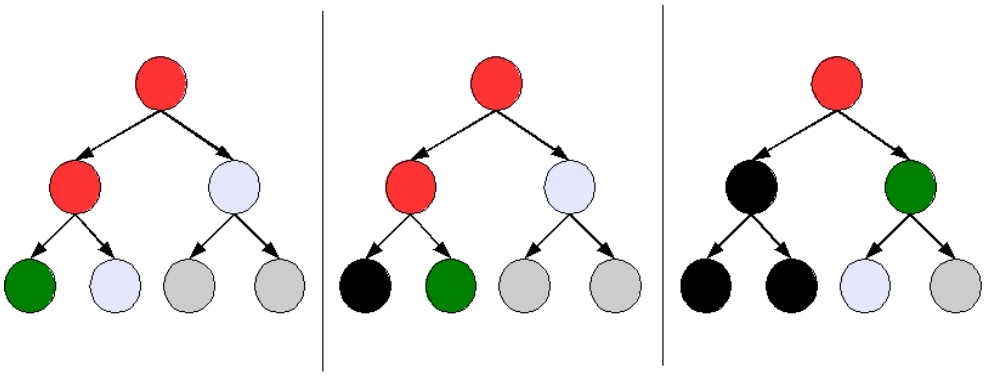


Abbildung 1.2: Rote Knoten sind bereits abgesucht und im Speicher, der Grüne wird gerade bearbeitet und die hellblauen wurden zum Queue zur Bearbeitung hinzugefügt. Die schwarzen Knoten sind abgearbeitet und nicht mehr im Speicher.

1.2.3 Uniform-cost Search

Bei dieser Art der Suche, welche auf dem krzesten Weg zwischen 2 Punkten Algorithmus von DIJKSTRA aus dem Jahre 1959 beruht, wird immer der Knoten mit den niedrigsten Pfadkosten gewählt, woraus auch automatisch folgt, dass, wenn sich die Pfadkosten nicht unterscheiden, eine normale Breitensuche vorliegt. Wenn dieser Algorithmus auf eine Schleife mit den Pfadkosten 0 oder weniger stößt, bleibt er dort stecken. Daraus resultiert folgendes:

- **Vollständigkeit und Optimalität**: Sofern alle Pfadkosten $\geq$ e >0
- **Zeit-/Speicheraufwand**: O($b^{top(C*/e)}$) wobei C* den Kosten für die optimale Lösung entspricht $\rightarrow$ Es werden viele kleine Schritte bevorzugt, auch wenn diese nicht zum gewnschten Ziel fhren.

1.2.4 Tiefensuche (Depth-first search)

Bei der Tiefensuche wird zunächst der tiefste Knoten im aktuellen Teilstück des Suchbaumes gewählt. Sollte ein Knoten keine (noch nicht durchsuchten) Kinder haben, so sprint der Algorithmus zum Vater dieses Knoten zurck. Alle durchsuchten Knoten, die nicht zwischen Wurzel und aktuellem Knoten liegen, werden fallengelassen. Tiefensuche entspricht also dem Aufruf TREE-SEARCH(problem, „Kinder am Anfang einfügen").

- **Vollständig**: Nur, wenn der Suchbaum ausschließlich Äste endlicher Länge enthält.
- **Optimal**: Nein
- **Zeitaufwand**: O(b^{d+1})
- **Speicheraufwand**: O($b*m$) im Sonderfall backtracking search: $O(m)$ *(m sei die maximale Suchtiefe)*

1.2.5 Begrenzte Tiefensuche (Depth-limited search)

Depthlimited search ist eine Tiefensuche mit einer festgelegten maximalen Suchtiefe l. Alle Knoten der Tiefe l werden behandelt, als besäßen sie keine Kinder. Tiefensuche entspricht damit DLS mit l = unendlich.

- **Vollständig**: Nur wenn gilt l $\geq$ m
- **Optimal**: Nur wenn gilt l = d
- **Zeitaufwand**: O(b^l)
- **Speicheraufwand**: O($b * l$)

```
DEPTH-LIMITED-SEARCH(Problem, Limit)
liefere als Ergebnis(RECURSIVE-DLS(Startknoten, Problem, Limit))
wenn kein Ergebnis liefere Fehler

RECURSIVE-DLS(Knoten, Problem, Limit)
falls Knoten Lösung für Problem
    dann liefere als Ergebnis(Knoten)
falls wir tiefer sind als das Limit
    dann liefere als Ergebnis(Cutoff)
rufe RECURSIVE-DLS für die Kinder des Knotens auf
```

Dieser Algorithmus hat 2 mögliche Rückgabewerte: Lösungspfad - wenn eine Lösung gefunden wurde, Fehler - wenn der ganze Baum durchsucht, aber keine Lösung gefunden wurde, Cutoff - Es wurde keine Lösung gefunden, aber auch nicht der ganze Baum durchsucht

1.2.6 Iterativ vertiefende Tiefensuche (iterative deepening depth-first search)

Bei dieser Art der Suche handelt es sich um eine begrenzte Tiefensuche, welche erstmals von Slate und Atkin 1977 für ein Schachprogramm verwendet wurde, deren Tiefenlimit l Schrittweise erhht wird, bis l=d gilt, also eine Lösung gefunden wurde.

- **Vollständig**: Ja, sofern ein Ziel in einer endlichen Tiefe existiert und der Verzweigungsfaktor endlich ist
- **Optimal**: Ja, sofern alle Pfadkosten identisch sind
- **Speicheraufwand**: $O(b * d)$

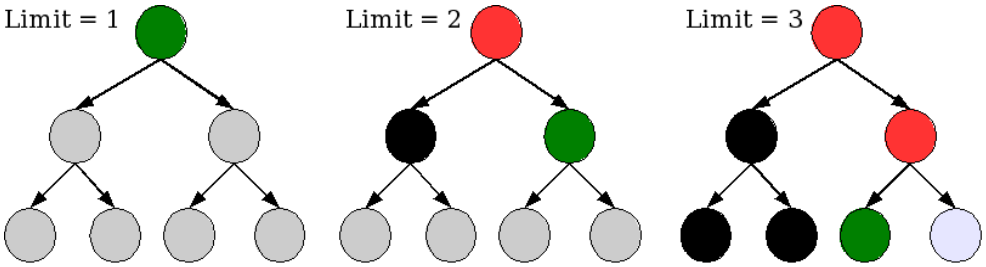


Abbildung 1.3: Rote Knoten sind bereits abgesucht und im Speicher, der Grüne wird gerade bearbeitet und die hellblauen wurden zum Queue zur Bearbeitung hinzugefügt. Die schwarzen Knoten sind abgearbeitet und nicht mehr im Speicher.

Man beachte, dass IDS durch das wiederholte Erzeugen vieler Knoten augenscheinlich zwar verschwenderisch wirkt, sich aber bei genauerer Betrachtung folgender Vergleich zwischen Breitensuche und IDS ergibt:

$$N(IDS) = (d) * b + (d-1) * b^2 + \cdots + (1) * b^d = \sum_{i=1}^{d} (d-i) * b^d$$

$$N(BFS) = b + b^2 + \cdots + b^d + b^{d+1} = \sum_{i=1}^{d+1} b^i - b$$

Da gilt

$$\sum_{i=1}^{d} ((d-1-i)b^2) < b^{d+1}$$

ist IDS also tatsächlich schneller als BFS und es ergibt sich ein

- **Zeitaufwand**: O(b^d)

Aus diesem Grund ist IDS häufig der bevozugte Suchalgorithmus, wenn die Tiefe d des Suchzieles unbekannt ist.

```
ITERATIVE-DEEPENING-SEARCH(Problem)
für Tiefe = 0 bis unendlich
{
    Resultat = DEPTH-LIMITED-SEARCH(problem, depth)
    falls Resultat ≠ Cutoff
        dann liefere als Ergebnis(Resultat)
}
```

1.2.7 Bidirektionale Suche (Bidirectional search)

Die Hauptidee hinter der 1969, 1971 von Pohl entwickelten bidirektionalen Suche liegt darin, das $2 * (b^{d/2})$ wesentlich kleiner ist als (b^d) bzw. die Fläche zweier kleiner Kreise kleiner als die eines großen Kreises. Bei der bidirektionalen Suche werden 2 Breitensuchen eingesetzt, die jeweils vom Ausgangszustand und Ziel starten und einander suchen. Die Überprüfung ob ein Knoten bereits von der anderen Suche gefunden wurde, findet dabei mit einer Hashtabelle statt. Das Ziel muß dazu jedoch eindeutig bestimmt sein und die Suchfunktion muß die Generierung des Vorgängers eines Knotens zulassen, wie dass zum Beispiel beim 8 Puzzle oder Rubikwürfel der Fall ist. Schach hingegen ist auf diese Art und Weise nicht lsbar, da in diesem Fall eine große Menge an möglichen Suchzielen existiert.

- **Vollständig**: Ja
- **Optimal**: Ja
- **Zeitaufwand**: $O(b^{d/2})$
- **Speicheraufwand**: $O(b^{d/2})$

1.2.8 Unterbindung von Schleifen bei der Suche

Ein großes Problem ist die Möglichkeit, dass z.B. bei dem Durchsuchen eines Graphen ein Suchalgorithmus einige Suchstadien evtl. mehrmals durchläuft und somit viele redundante Pfade abläuft. Bei einer Tiefensuche lässt sich ein Teil der Wiederholungen durch das Vergleichen des aktuellen Knotens mit seinen Vorgängern vermeiden. Will man jedoch alle redundanten Pfade ausschließen, so bentigt man eine alle bereits besuchten Knoten enthaltende Hashtabelle um jeden neuen Knoten auf Wiederholung zu untersuchen. Man beachte hierbei, dass diese Hashtabelle den linearen Speicheraufwand von Tiefensuche und IDS zunichte macht.

```
GRAPH-SEARCH(Problem, Queueing-Fn)
initialisiere Queue mit Startknoten
initialisiere leere Liste der abgearbeiteten Knoten
solange Queue ≠ leer
{
    Knoten k = Queue.pop
    falls k Lösung für Problem
        dann liefere als Ergebnis(Pfad zu k)
    falls k nicht in Liste der abgearbeiteten Knoten ist
        dann füge k zu Liste der abgearbeiteten Knoten hinzu
    wähle weitere Knoten aus Suchbaum ∉ Liste der abgearbeiteten Knoten
    Fge ausgewählte Knoten an durch Queueing-Fn vorgesehene Stelle zu Queue
}
liefere als Ergebnis(Fehler)
```

Man vergleiche diesen Algorithmus mit TREE-SEARCH: Es wird leicht ersichtlich, das die Verwendung der Hashtabelle(Liste der abgearbeiteten Knoten) das Auftreten von wiederholten Stadien verhindert.

1.2.9 Constraint Satisfaction Problems

Ein CSP besteht aus einer Menge von Variablen und einer Menge von Anforderungen/Abhängigkeiten. CSP haben die gemeinsame Eigenschaft, dass sie sich in einem Abhängigkeitsgraphen (constraint graph) darstellen lassen.

Da CSP eine bestimmte Menge an Variablen besitzen, die alle belegt werden möchten, ist uns auch die Suchtiefe unseres Suchbaumes bekannt. Auf jeder Ebene des Suchbaumes wird nämlich eine Variable belegt. Das wiederum hat zur Folge, das Formen der Tiefensuchen eine beliebte Möglichkeit sind, CSPs zu Lösen.

Die einfachste Form von CSP haben nur dikrete Variablen, bei denen die Belegungsmöglichkeiten für jede einzelne Variable endlich sind wie z.B. 4-Damen. Der Verzweigungsfaktor für diese Art von CSP ist maximal gleich der Anzahl der Belegungsmöglichkeiten für die einzelnen Variablen. Außerdem lassen sie diese grundsätzlich auf Boolean CSPs zurückführen. Aufgrund der Tatsache, dass es sich aber z.B. bei dem SAT Problem um ein CSP handelt, welches NP-Vollständig ist, können wir nicht davon Ausgehen, CSP Probleme grundsätzlich in weniger als exponentieller Zeit zu lösen.

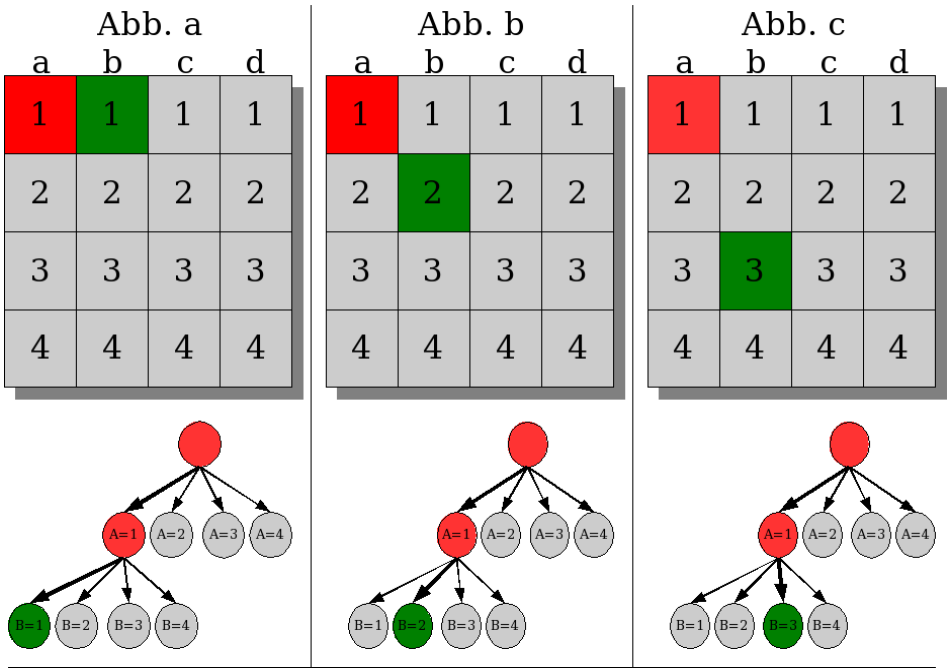
Zudem kommen CSPs, welche z.B. auf den natürlichen Zahlen arbeiten und damit zwar diskrete Werte haben, aber unendlich viele Belegungsmöglichkeiten. Als Beispiel lässt sich ein z.B. Terminkalender angeben. Da wir bei solchen Problemen nicht alle Lösungsmöglichkeiten aufzählen also man einen Termin z.B. bis ins Unendliche nach hinten verschieben kann, werden zur Lösung solcher Probleme Constraint Languages verwendet. In unserem Beispiel bedeutet das soviel, wie Termin1+2h $\leq$ Termin 2 etc.

Auf die noch komplexeren CSP möchten wir hier aufgrund unseres begrenzten Umfanges nicht eingehen.

1.2.10 Backtracking Search

Bei dieser Form der Suche wird die Tatsache genutzt, dass bei einer gültigen Lqqqsung für ein CSP alle Bedingungen des CSP erfüllt sein müssen. Verletzen also ein oder mehr Variablen eine der Bedingungen, so folgt daraus zwangsläufig, dass jeder weitere Versuch die restlichen Variablen zu belegen nicht mehr zu einer gültigen Lqqqsung führen kann. Daher wird beim Backtracking eine Tiefensuche angewendet, also die Variablen nacheinander belegt. Nach jeder neu belegten Variable überprüft der Algorithmus, ob die momentane Belegung noch gültig ist. Ist sie dies, setzt er seine Suche bei der nächsten Variable fort. Ist sie ungültig, springt er einen Schritt im Suchbaum zurück und versucht es mit einem anderen Kind, also testet eine alternative Belegung. Falls keine solche existiert, so steigt er im Suchbaum einfach um noch eine Ebene auf und versucht es dort noch einmal. Wir wollen dies am 4 Damen Problem erläutern.

Abbildung 1.4: Graue Felder sind nicht betrachtet, rote wurden gesetzt und sind im Speicher, das grüne Feld entspricht der aktuell bearbeiteten Position.



Gegeben sei zunächst ein 4x4 Feld. Wir unterteilen dieses in 4 Spalten welche unsere 4 Variablen (a,b,c,d) seien. Jede dieser Spalten hat 4 Felder, woraus die mgliche Variablenbelegung resultiert, also 1,2,3,4.

Unser Algorithmus belegt zunächst die erste Variable a mit 1. Danach die zweite b auch mit 1. Wir befinden uns also auf der 3ten Ebene unseres Suchbaumes. Die nun folgende Prüfung auf Konsistenz bzgl. der Anforderungen des 4-Damen Problemes (keine Dame darf die andere schlagen können) ergibt, dass es sich nicht um eine gltige Belegung handelt (Abb. a) Also springt der Algorithmus einfach um eine Ebene zurck und testet ein neues Kind. Das bedeutet also, dass belegt b mit 2 belegt (Abb. b). Dieser Prozeß setzt sich nun so lange fort (Abb. c) , bis der Algorithmus es geschafft hat, eine Kombination zu finden, bei der erstens alle Variablen belegt sind, also er sich in der niedrigsten Ebene des Suchbaumes befindet und zweitens die Bedingungen des n-Damenproblems erfllt sind (Abb d).

1.2.11 Forward checking

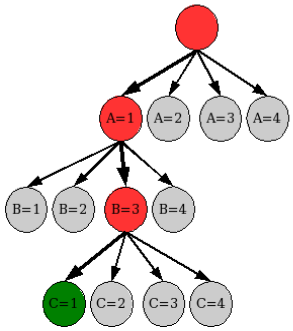
Forward checking ist eine Methode um die Anzahl der Sackgassen, in welche ein Backtracking Algorithmus läuft zu verringern. Beim Forward Checking werden jeweils die Werte, welche zu einem nicht Erfüllen der Abhängigkeiten führen würden, aus einer Liste der möglichen Werte für die von der aktuell bearbeiteten Variable abhängigen entfernt. Beim 4 Damen Problem würde das bedeuten:

Abb. c

a	b	c	d
1	1	1	1
2	2	2	2
3	3	3	3
4	4	4	4

Abb. d

a	b	c	d
1	1	1	1
2	2	2	2
3	3	3	3
4	4	4	4



Nachdem Spalte a die 1 zugeordnet wurde, das diese aus den Übrigen entfernt wird. Desweiteren werden aus b die 2, aus c die 3 und aus d die 4 entfernt. Danach wird im nächsten Schritt Spalte b die 3 zugeordnet, denn 1 und 2 stehen ja nicht mehr zur Verfügung.

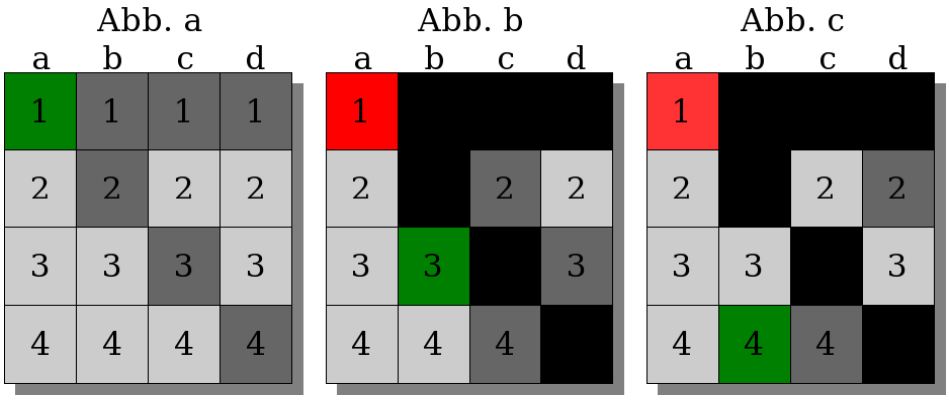
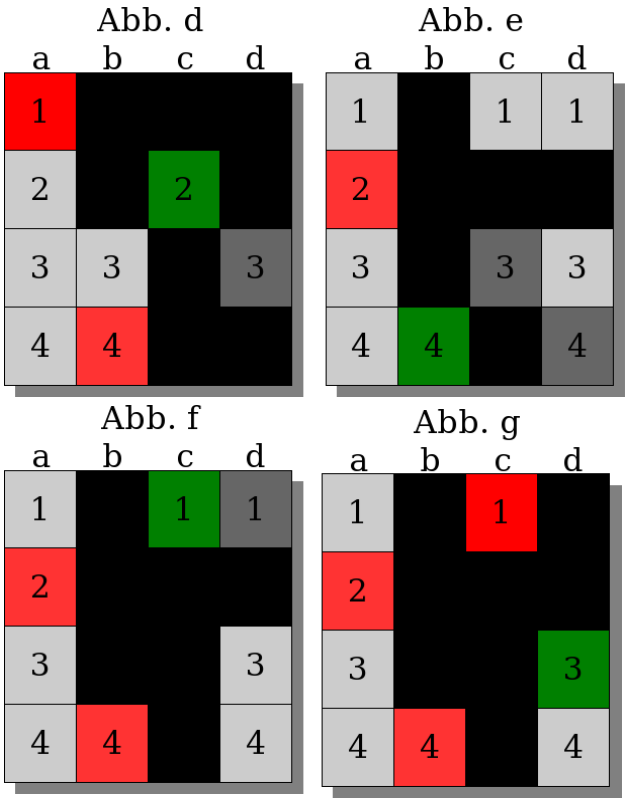


Abbildung 1.5: Hellgraue Felder sind frei, sofern keine Dame in der selben Spalte steht. Auf den roten Feldern stehen bereits Damen, auf dem grünen Feld wurde soeben eine Dame gesetzt. Die dunkelgrauen Felder wurden beim Setzen der grünen Dame aus der Liste der möglichen Belegungen für folgende Damen entfernt, während die schwarzen Felder bereits in vorherigen Schritten aus jener entfernt wurden.



1.3 Heuristische Suchverfahren

1.3.1 Unterschiede zu allgemeinen Suchverfahren

Informierte Suchverfahren basieren auf Heuristiken, besitzen also manuell einprogrammiertes problemspezifisches Wissen. Die Heuristiken stellen dabei eine Art Orakel dar, auf die der Algorithmus zurückgreift um den nächsten Schritt abzufragen. Die heuristische Funktion nimmt dazu Informationen von außerhalb des Wahrnehmungsbereich eines allgemeinen Suchverfahrens auf und bewertet die jeweilige Lösung nach ihnen. Wie das folgende Kapitel zeigen wird, gibt es keine Heuristiken die immer perfekt antwortet, also einen schlechtesten Zeitaufwand von $O(b+m)$ besitzt, sondern nur $O(b^m)$. Die durchschnittliche Suchdauer liegt dagegen oft weit unter der eines vergleichbaren allgemeinen Suchalgorithmus.

1.3.2 Gruppen von heuristischen Suchverfahren

Man kann die Algorithmen in zwei Gruppen einteilen. Vertreter der ersten Gruppe setzen sich Schritt für Schritt die Lösung zusammen, Vertreter der zweiten Gruppe betrachten fertige, nicht unbedingt optimale, Lösungen und versuchen diese iterativ zu verbessern. Die Grundidee bei beiden Gruppen ist, dass man jeweils alternative Schritte bzw. Lösungsmöglichkeiten vergleicht und jeweils möglichst die wählt, die auch zu einer optimalen Lösung führen. Dazu wählt man eine nicht-negative Heuristikfunktion $h(n)$, 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 Lösung darstellt. Ausserdem benötigen wir eine Auswahlfunktion i, die uns einen möglichst noch nicht betrachteten neuen Zustand, also ein Lösungsschritt der bzw. eine Lösung, zurückgibt. Das Problem zu jeder Aufgabenstellung ist also erst einmal das Finden einer Funktion $h(n)$ und i die das bewerkstelligen. Eine der Aufgabenstellung nicht angepaßte Heuristik und Auswahlfunktion führt zu längerer Laufzeit oder gar zu Sackgassen und Schleifen.

1.3.3 Finden einer heuristischen Funktion

Um eine Funktion h zu finden, die den Suchvorgang möglichst verkürzt, bietet es sich an, ein sogenanntes Relaxed Problem, also ein vereinfachtes Problem zu betrachten, bei dem bestimmte Beschränkungen in der Lösungsschrittwahl aufgehoben sind. Damit erhalten wir eine mögliche Bewertung eines Lösungsschritt, wie weit wir von der optimalen Lösung entfernt sind.

Als Beispiel betrachten wir das sogenannte 8-puzzle ([1], Seite 105) Problem:

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

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 dürfen, würde eine komplette

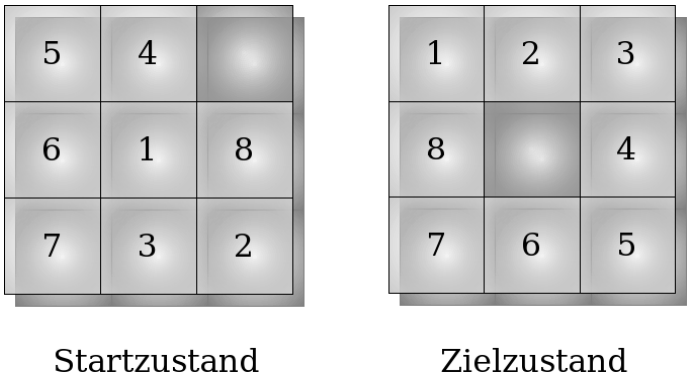


Abbildung 1.6: Beispiel zum 8-PUZZLE Problem

Suche aller Möglichkeiten mit Zugtiefe 20 zu $3.5 * 10^9$ oder, ignoriert man sich wiederholende Situationen, zu 9! Zustände führen.

Wie stellen wir nun fest, ob wir uns beim Bewegen eines Teiles auf die optimale Lösung zu bewegen oder uns entfernen? Beim Ursprungsproblem können wir nur feststellen, ob ein Zustand dem Zielzustand entspricht oder nicht. Wir müssen also ein vereinfachtes, sogenanntes RELAXED PROBLEM ([1], Seite 107) finden, so dass Überschreitungen von Beschränkungen nicht zu einem Ignorieren des Lösungsschritts bzw. der Lösung führt.

Zuerst listen wir die Beschränkungen noch einmal getrennt auf:

Ein Teil darf

1. pro Schritt nur 1 Feld und
2. nicht schräg und
3. nur in ein freies Feld

verschoben werden

Heben wir in diesem Fall die Beschränkung 3 einfach auf, dürfen 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 Lösung benötigen. In diesem speziellen Fall wird das *Manhattan distance* genannt ([1], Seite 108).

Hebt man zusätzlich noch Beschränkung 1 auf, erhält man eine Heuristik, die die Zahl der Teile die sich an falscher Position befinden beschreibt. Nach ([1], Seite 108) ist die erste Heuristik der zweiten Heuristik mit weniger Beschränkungen überlegen, führt also zu einer Suche mit geringerer durchschnittlichen Suchdauer.

Prüft man alle Kombinationen von Beschränkungsaufhebungen erreicht man so eine für das jeweilige Problem optimale Lösung. Diesen Weg beschreitet ABSOLVER ([1], Seite 108).

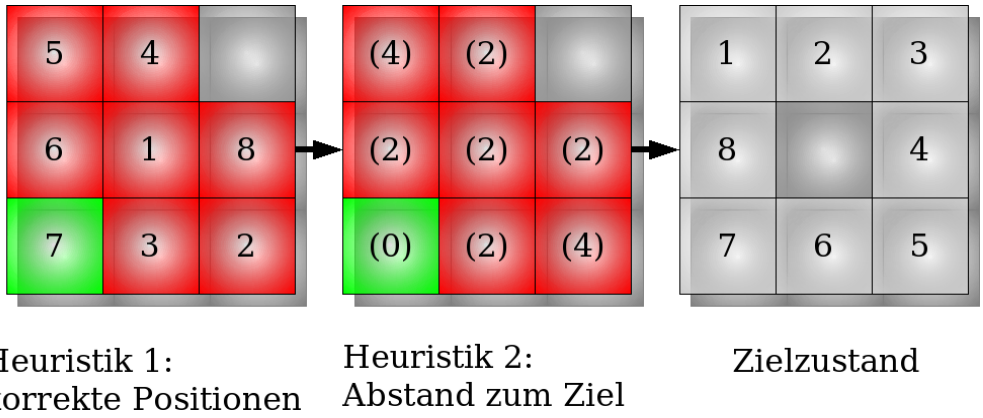


Abbildung 1.7: 2 mögliche Heuristiken zur Lösung von 8-PUZZLE

1.3.4 Formale Betrachtung des Problems

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

Das Grundgerüst der informellen Suche sieht erst einmal so aus:

```
HEURISTISCHE SUCHE(Zustand alterZustand)
solange erstelleNeuenZustand(alterZustand,Bewertung()) ≠ NULL
    Zustand neuerZustand = erstelleNeuenZustand(alterZustand,Bewertung())
liefere als Ergebnis (alterZustand)
```

Zustand ist je nach Problemdarstellung unterschiedlich, im Weiteren wird hier nur auf Graphendarstellungen eingegangen, da sich die meisten Probleme problemlos auf einen Graphen transformieren lassen können. Je nach Algorithmus werden hier auch pro Knoten unterschiedlich viele Daten gespeichert.

Bewertung stellt 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 genügen.

erstelleNeuenZustand entspricht unserer Funktion i, sie akzeptiert zum einen den momentanen Bearbeitungszustand und zum anderen die Heuristikfunktion *Bewertung*. Auf einige Beispiele für mögliche h und i Funktionen wird nun in den nächsten beiden Abschnitten näher eingegangen.

1.4 Algorithmen 1. Gruppe - Schrittweiser Aufbau

1.4.1 Greedy Search

Bei dem sogenannten GREEDY SEARCH wählt die Funktion i denjenigen Lösungsschritt der die geringste Bewertung aufweist. Der Lösungsschritt wird dabei aus der Menge von geöffneten Knoten in *alterZustand* ausgewählt. Dann werden die Kinder des ausgewählten Knotens geöffnet und wieder deren Bewertungen berechnet usw.

```
GREEDY SEARCH(Zustand alterZustand)
solange erstelleNeuenZustand(alterZustand,Bewertung()) ≠ NULL
    Zustand neuerZustand = erstelleNeuenZustand(alterZustand,Bewertung())
liefere als Ergebnis(alterZustand)

Zustand erstelleNeuenZustand(Zustand momentanerZustand,Bewertung())
Knoten n = min(Bewertungsfunktion(geöffnete Knoten von momentanerZustand))
momentanerZustand = SchließeAlleKnoten(momentanerZustand)
falls öffneKnoten(momentanerZustand,n) ≠ momentanerZustand
    liefere als Ergebnis(öffneKnoten(momentanerZustand,n))
```

Will man nun mit GREEDY SEARCH z.B. den kürzesten Pfad in einem Graphen finden, bietet sich für h die einfache Distanz über die Luftlinie an.

Also z.B. hier:

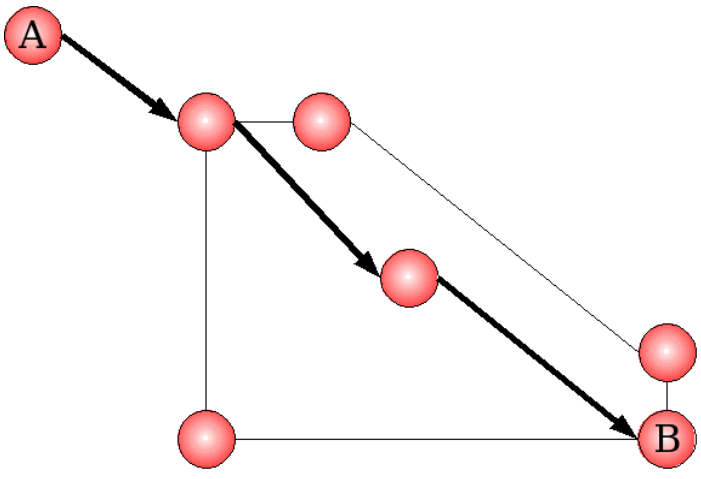


Abbildung 1.8: GREEDY SEARCH in Aktion

Leider hat der Algorithmus ein paar Nachteile. In Abbildung 1.9 sieht man, dass GREEDY SEARCH den Weg über D wählt, anstatt über E und F zu laufen,

da D näher an B liegt als E. Bei Abbildung 1.10 kann der Algorithmus von D aus keinen Ausweg finden, da kein Kind von D eine geringere Bewertung als D selbst aufweist. D stellt für Greedy Search das sogenannte lokale Optimum dar.

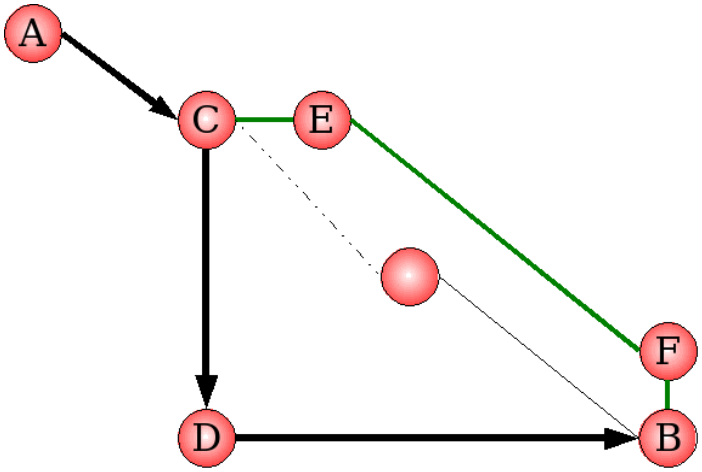


Abbildung 1.9: Greedy Search ist anfällig gegenüber falschen Starts

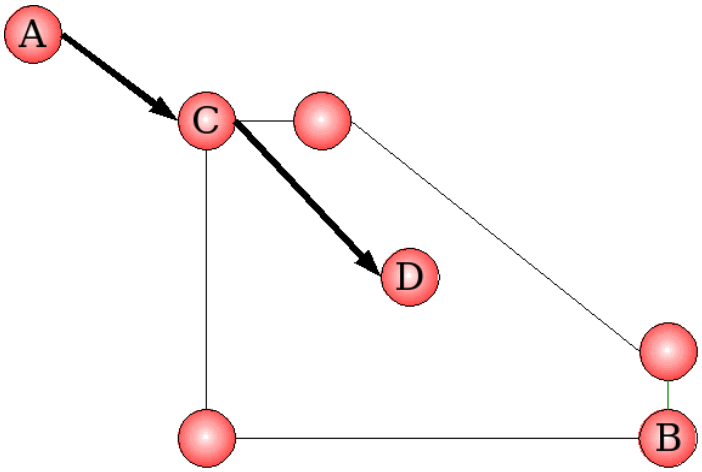


Abbildung 1.10: Greedy Search ist anfällig gegenüber Sackgassen

Der Greedy Search Algorithmus ist weder optimal noch vollständig, führt jedoch in einigen Fällen zu einem relativ guten und schnellen Ergebnis. In der Grundform hat der Greedy search Algorithmus einen Speicheraufwand von $O(m)$, da nur der Weg zum Ziel und alle Söhne des aktuell geöffneten Knotens betrachtet werden. Prüft man auf Sackgassen und Schleifen, steigt der Speicheraufwand von $O(b+m)$ auf $O(b^m)$, wird aber trotzdem nicht optimal, wie in Abbildung 1.10 gezeigt und auch nicht vollständig, da er bei Graphen mit unendlich langen Pfaden nie terminiert. Mit Gedächtnis hat er einen Zeitaufwand im schlechtesten Fall von $O(b^m)$, da immer wieder rückgesprungen wird

bis zwangsläufig alle Knoten (in einem endlichen Graphen) untersucht wurden.

1.4.2 Greedy Search mit CSPs

Hier nun ein weiteres Beispiel zu Greedy Search in Verbindung mit dem Constraint Satisfaction Problem, eine Landkarte so zu färben, dass kein Land die gleiche Farbe wie ein angrenzendes Land besitzt und dabei eine bestimmte Zahl von unterschiedlichen Farben nicht überschreitet. Als Heuristik bietet sich Most-Constrained Variable an, wir beginnen also mit der Färbung eines Landes und nehmen als nächstes Land jenes, welches ungefärbt ist und die meisten gefärbten Nachbarn besitzt. Welche Farbe wir dann wählen ist beliebig, solange sie nicht mit einer angrenzenden Farbe in Konflikt gerät. Wie man an diesem Beispiel sieht, arbeitet die heuristik zusammen mit Greedy Search sehr erfolgreich. Als Basis für dieses Beispiel diente die Europakarte von [3].

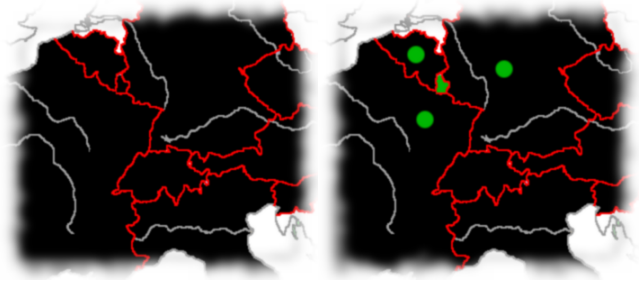


Abbildung 1.11: Der Algorithmus Greedy Search mit Heuristik der Most-constrained Variable beginnt zufällig in Lichtenstein

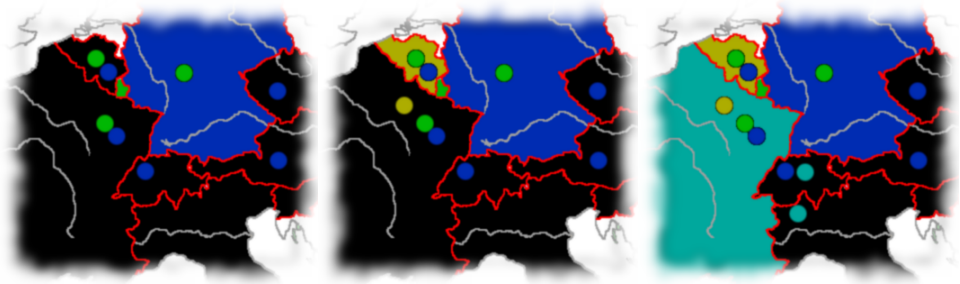


Abbildung 1.12: Deutschland und Belgien wird gewählt, dann Frankreich wegen 3 Punkten

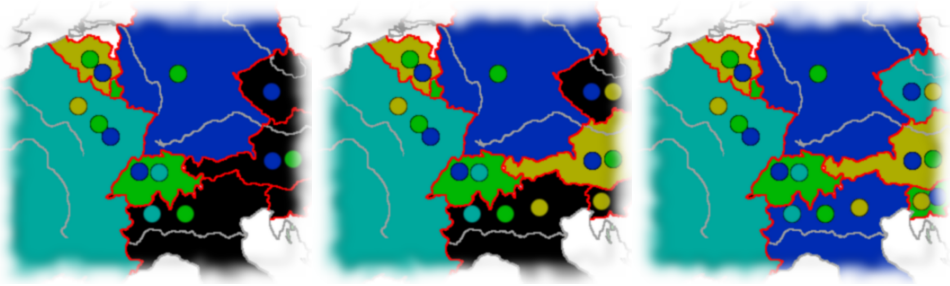


Abbildung 1.13: Die Schweiz und dann Österreich werden mit 2 Punkten, Italien danach mit 3 Punkten gewählt, der Rest wird aufgefüllt

1.4.3 Verbesserungen von Greedy Search: A*

Erweitert man die Funktion i, dass nicht nur die Kinder des gerade geöffneten sondern aller noch nicht besuchten Knoten betrachtet erhalten wir einen vollständigen Algorithmus. Dadurch treten nun keine Schleifen und Sackgassen mehr auf, es wird einfach an die nächste passende Stelle gesprungen.

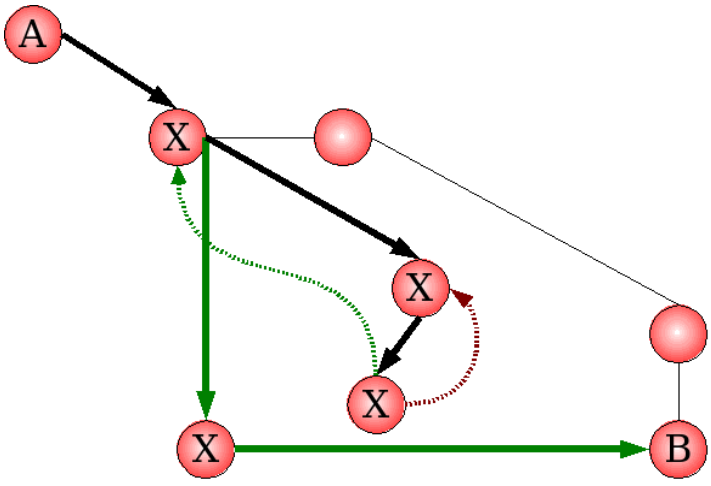


Abbildung 1.14: Greedy Search mit Gedächtnis, Rücksprung an besuchte Knoten

Leider ist dieser Algorithmus immer noch anfällig gegenüber einem ungünstigen Start und in vielen Fällen nicht optimal. Deshalb hat man den sogenannten A* Algorithmus entwickelt, der die gleiche i Funktion aufweist, dessen Bewertungsfunktion $h(n)$ aber eine Kombination aus den bereits bekannten Greedy Searchs und Uniform-cost Searchs ist.

Hier ist also
h(n) = (Abstand(n, Ziel) (Luftlinie) + Pfadlänge(n, Start))

```
A* SEARCH(Zustand alterZustand)
```

```
solange erstelleNeuenZustand(alterZustand,Bewertung()) ≠ NULL
    Zustand neuerZustand = erstelleNeuenZustand(alterZustand,Bewertung())
liefere als Ergebnis(alterZustand)

Zustand erstelleNeuenZustand(Zustand momentanerZustand,Bewertung())
Knoten n = min(Bewertungsfunktion(geöffnete Knoten von momentaner Zustand))
falls öffneKnoten(momentanerZustand,n) ≠ momentanerZustand
    liefere als Ergebnis(öffneKnoten(momentanerZustand,n))
```

Bemerkung:
Man kann h auch beliebig anders wählen, A* bleibt aber nur solange optimal wie die Heuristik h der Dreiecksgleichung genügt, 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 über C) <h(B über A) gilt. Eine genauere Beschreibung und eine Möglichkeit derartige heuristische Funktionen zu korrigieren findet man in ([1], Seite 99).

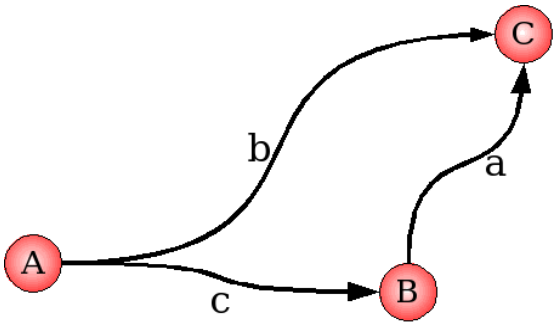


Abbildung 1.15: Dreiecksungleichung und A*: A* ist optimal wenn $h(b) \leq h(a)+h(c) \forall a, b, c$ wobei c die kürzeste Verbindung von A nach C darstellt.

Ein Beispiel das zeigt, dass z.B. „Luftlinie" nicht immer die kürzeste Verbindung darstellt, wäre das sogenannte BRACHISTOCHRONE Problem, also die Frage nach dem *schnellsten* Weg von A nach B:

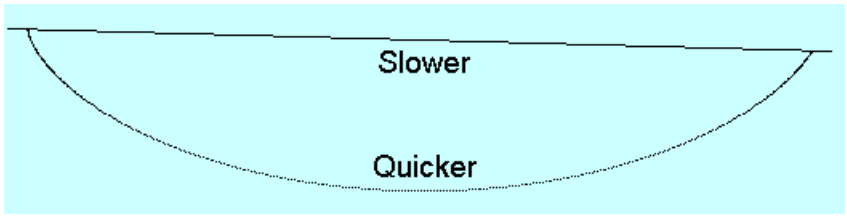


Abbildung 1.16: BRACHISTOCHRONE Problem - Was ist der *schnellste* Weg von links nach rechts? [2]

Untersuchungen erbrachten die Ergebnisse, dass A* optimal und vollständig ist und auch, dass es keinen anderen optimalen und vollständigen Algorithmus gibt, der (für eine beliebige heuristische Funktion) die Aufgabe mit Expandierung von weniger Knoten schafft ([1], Seite 101).

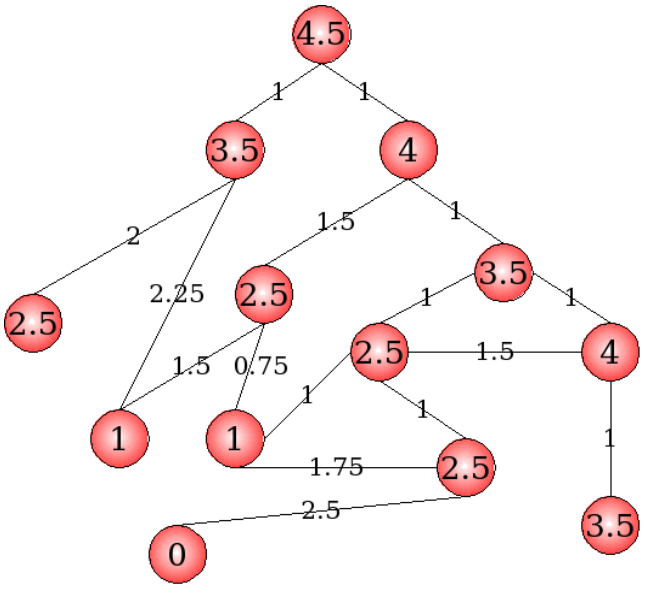


Abbildung 1.17: A* Search, Graphenproblem fuer A*

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

In Abbildung 1.17 starten wir mit dem obersten Knoten und berechnen den Wert der Heuristik beider Kinder. Die Pfadlänge beträgt 1, die Entfernung zum Ziel 3.5, also erhalten wir 4.5. Beim rechten Kind ist die Pfadlänge ebenfalls 1 + 4 (Entfernung zum Ziel) = 5. Dies zeigt Abbildung 1.18. Dort wählen wir den Knoten mit dem geringsten Wert aus und berechnen dessen Kinder, in diesem Fall das linke Kind: 2.5 (Entfernung zum Ziel) + 2 + 1 (Pfadlänge) = 5.5, und das rechte Kind: 1 (Entfernung zum Ziel) + 1 + 2.25 (Pfadlänge) = 4.25 usw. In Abbildung 1.19 wurde ein kürzerer Pfad gefunden über das rechte Kind gefunden (hier mit * markiert), die 7.25 wird also ersetzt durch 5. Zum Schluß haben wir das Ziel gefunden und erhalten den optimalen Pfad indem wir rekursiv vom Ziel zurücksteigen. Der Wert im Zielknoten ergibt gleichzeitig auch die Länge des Pfades.

A* hat in seiner Grundform ebenfalls einen Zeit und Speicheraufwand von $O(b^m)$, da wie beim uniform cost search alle geöffneten Knoten im Speicher liegen und im schlechtesten Fall auch alle Knoten untersucht werden müssen. Für b = 2 und m = 40 benötigt ein Aldi-Rechner für A* im schlechtesten Fall vielleicht eine Stunde. Der Speicherbedarf dagegen ist im schlechtesten Fall dagegen mit 1024 GB jenseits von gut und böse. Der Speicheraufwand lässt sich jedoch mit einem etwas abgewandelten Algorithmus reduzieren. Zum Glück gibt es mehrere verschiedene derartige Algorithmen, im Folgenden werden zwei vorgestellt.

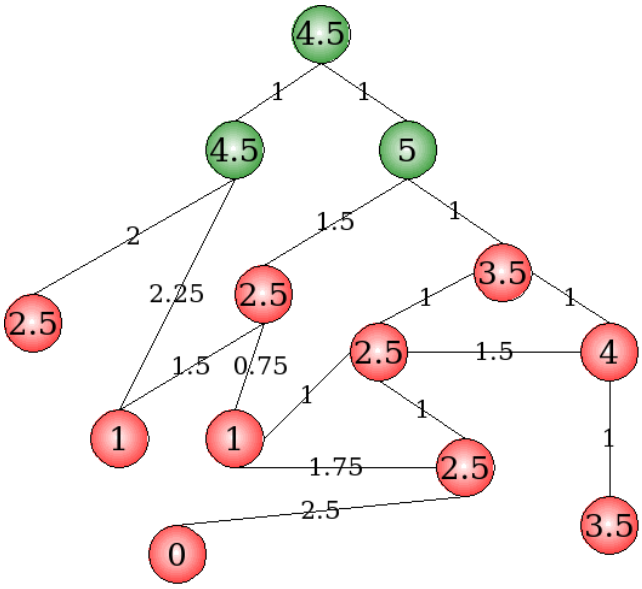


Abbildung 1.18: A* Search, Startknoten expandiert, den zwei Kindern neue Werte zugewiesen

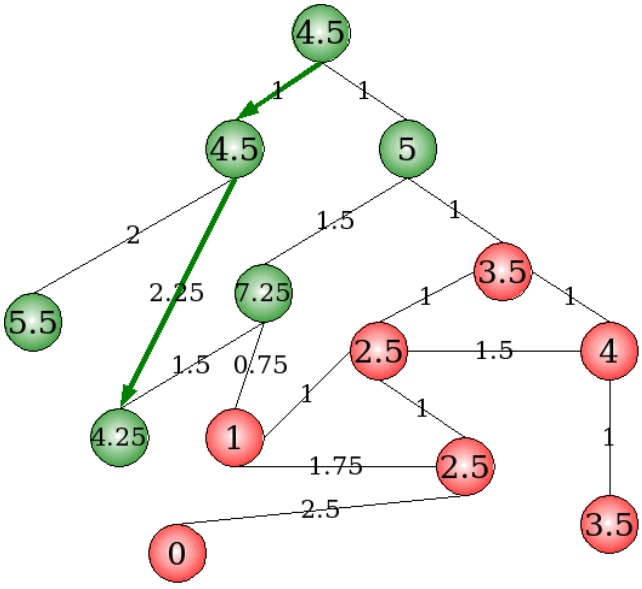


Abbildung 1.19: A* Search, erster und zweiter Schritt

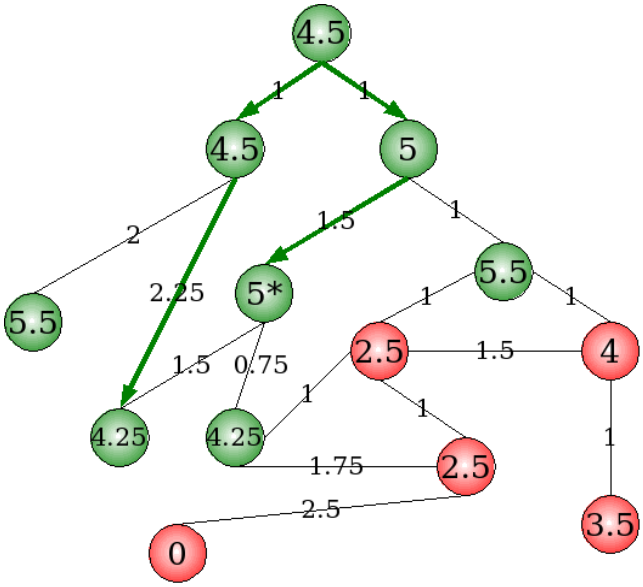


Abbildung 1.20: A* Search, dritter und vierter Schritt

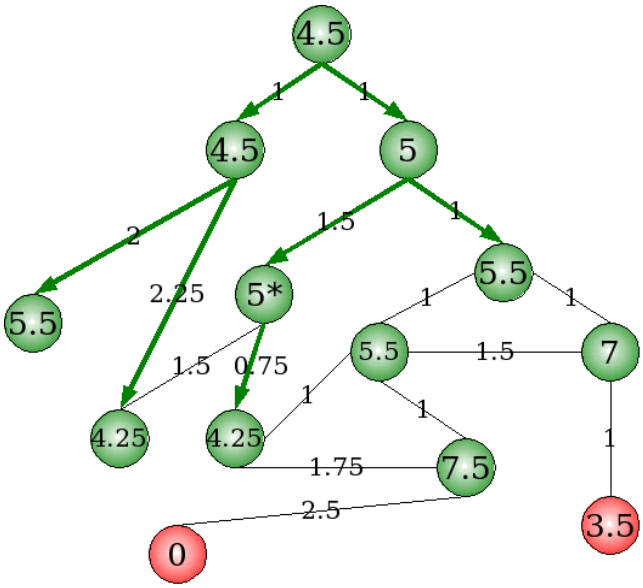


Abbildung 1.21: A* Search, restliche Schritte

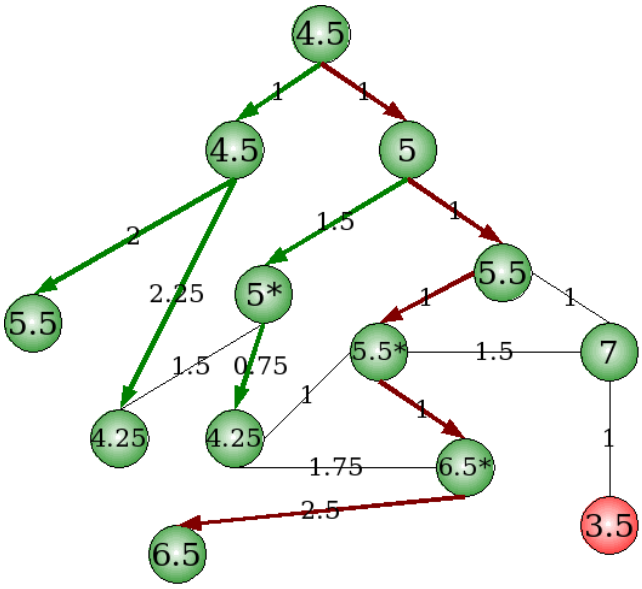


Abbildung 1.22: A* Search, Endzustand: optimaler Weg gefunden

1.4.4 Iterative Deepening A* Search

IDA* Search funktioniert im Grunde wie der bereits betrachtete Iterative Deepening Search, statt Knotentiefe als Grenze wird hier aber ein Kostenlimit gesetzt. Es wird also immer nur ein Bereich aus Knoten untersucht, deren Bewertungsfunktion h kleiner als das Kostenlimit ist. Das Speicherersparnis kommt natürlich um den Preis der Geschwindigkeit. Zwar werden pro Schritt weniger neue Knoten betrachtet, dafür müssen alle noch nicht besuchten Knoten in Suchtiefenreichweite neu berechnet werden. Dass dies keinen so großen Einfluß auf die Laufzeit hat, ergibt sich wie beim Iterative Deepening Search aus der Tatsache, dass in den meisten Fällen die unterste Ebene des Graphen die meisten Knoten enthält.

Die Qualität des Iterative Deepening A* Search ergibt sich aus der Zahl der unterschiedlichen Werte die die Heuristikfunktion annehmen kann, da pro Schritt die Suchtiefe auf den nächstgrößeren Wert geschaltet wird. Eine Suche in Graphen mit reellwertiger Heuristikfunktion kann also zu einem unendlich langen Unterfangen werden, der Algorithmus ist also weder vollständig, noch optimal.

1.4.5 Simplified Memory-Bounded A* Search

Im Gegensatz zu A* verbraucht SMA* Search keine feste Speichermengenge, sondern passt sich an den verfügbaren Speicherplatz an, versucht also so viele Knoten wie möglich zu speichern. Der Algorithmus beginnt, falls der Speicherplatz knapp wird, bereits berechnete Teile des Graphen durch das jeweilige Minimum der Heuristikfunktionen aller enthaltenen Knoten zu ersetzen. Ist letzteres nie der Fall, arbeitet der Algorithmus also genau wie A* Search und

ist vollständig, sofern genug Speicherplatz für die Lösung mit Suchtiefe m, also $O(b*m)$ zur Verfügung steht und ist auch optimal nach ([1], Seite 104). Bei sehr schwierigen Problemen versagt SMA* SEARCH aber. Dort kommt es oft dazu, dass zuerst scheinbar schlechte Knoten gelöscht nur um kurz darauf wieder neu berechnet zu werden usw. Es kann also durch dieses andauernde Umverteilen des Speichers zu einer überproportionalen Erhöhung der benötigten Rechenzeit kommen.

1.5 Algorithmen der 2. Gruppe - Schrittweise Verbesserung

Optimal und vollständige Algorithmen sind schön 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 Lösung sondern nur z.B. eine 5% am Optimum liegende Lösung gefunden werden kann. Diese Aufgabe können die Algorithmen der zweiten Gruppe sehr erfolgreich lösen. Wie anfangs erwähnt wurde bei diesen Algorithmen die Lösung nicht Schritt für Schritt aufgebaut sondern fertige Lösungen generiert und diese iterativ verbessert. Wichtig ist dabei, dass die Aufgabenstellung so umgeschrieben wird, dass jede Art von Ergebnis gültig ist, also einer bestimmten Qualitätsstufe, der sogenannten Fitness, zugeordnet werden kann.

Dazu werden harte Nebenbedingungen, also z.B. Lagerüberschreitungen, Bargeldüberschreitungen, Zeitüberschreitungen etc. mit Strafkosten belegt, also die Fitness gesenkt, anstatt dass die Lösung ganz verworfen wird. Der große Unterschied zur ersten Gruppe ist aber, dass die neuen Lösungen mehr oder weniger zufällig generiert werden. Dadurch ergibt sich das Problem, dass man nie genau weiss, ob bei weiteren Durchläufen bessere Lösungen gewonnen werden können oder nicht.

Grundsätzlich arbeiten Algorithmen der 2. Gruppe wie folgt:

```
Schrittweise Verbesserung()
Losung alteLösung = erstelleZufälligeLösung()
Wiederhole
    Lösung neueLösung = i(alteLösung)
    △= h(neueLösung) - h(alteLösung)
    falls selektion(T, △) = true
        dann alteLösung = neueLösung
bis alteLösung ausreichend gut
liefere als Ergebnis(alteLösung)
```

Hier sind 2 neue Funktionen hinzugekommen, die jedoch nicht weiter kompliziert sind. *erstelleZufälligeLösung* erstellt wie der Name schon sagt, eine zufällige Startlösung, die dann schrittweise verbessert werden soll. *selektion* prüft in Abhängigkeit einer Variablen T, ob $\triangle$ ausreichend klein ist.
Auf einige Beispiele für $h(n)$, i und selektion möchte ich hier eingehen:

1.5.1 Hill-Climbing Algorithmus

Der einfachste darauf basierende Algorithmus ist der sogenannte Hill-Climbing Algorithmus. Der Name rührt 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 höchsten Berggipfel zu erreichen. Eine mögliche Strategie wäre, dass man sich vortastet, ob man denn mit dem nächsten Schritt etwas höher kommt oder nicht.

Abbildung 1.23: 3 Bergsteiger auf der Suche nach dem Gipfel, Grundlage für Bild von [4]

Hier in diesem Beispiel möchten Bergsteiger B, C und D zum höchsten Punkt A.

Beim Hill-Climber ist T = 0 und die SELECTION Funktion sieht so aus:

```
Selection(T, △)
falls △ <T
    dann liefere als Ergebnis(true)
liefere als Ergebnis(false)
```

Es werden also schlechtere Lösungen sofort verworfen, alle anderen weiterverwendet.

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 hängenbleiben

Betrachtet man hier Bergsteiger C und D werden diese, folgen sie der Ansteigung der Trettachspitze, zwar ziemlich hoch hinaus kommen, werden A jedoch nicht erreichen sondern im lokalen Optimum der Trettachspitze stecken bleiben. Zwar müsste C einfach nach links laufen um A zu erreichen, da er aber nicht weiss, dass A links von ihm liegt, ist er nicht besser dran als D.

Dies lässt sich beheben, indem man den Algorithmus nach einiger Zeit ohne Verbesserung einfach mit einer anderen Startkonfiguration/position neu beginnt, die beste Fitness der bisherigen Durchläufe aber speichert. Dann nennt man es einen RANDOM-RESTART HILLCLIMBING Algorithmus. In diesem Fall müsste C irgendwo zwischen E und F neustarten um auf direktem Wege zu A zu kommen. Problem bleibt hier festzustellen, ob wir denn in einem lokalen Optimum oder einer Kante stecken oder nicht schon kurz vor dem Ziel sind.

2. die Fitnesslandschaft ist oft sehr „wild" also mit großen Gradientenunterschieden besetzt

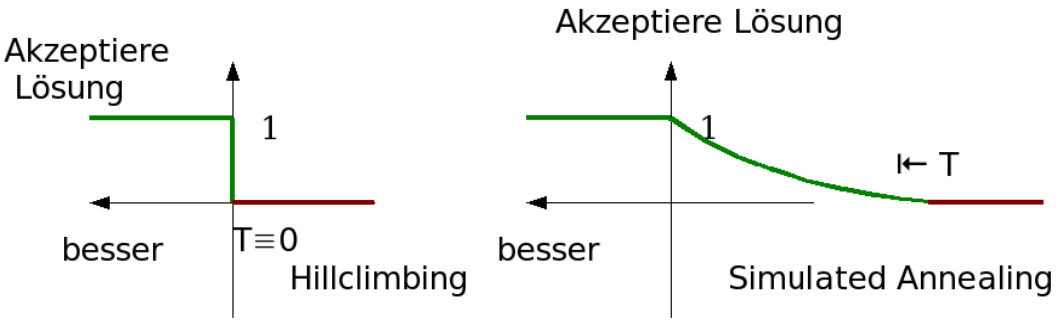


Abbildung 1.24: Vergleich Greedy Search mit Simulated Annealing

Hier hilft auch kein zufälliges Neustarten mehr, man wird wahrscheinlich fast alle Lösungsmöglichkeiten durchsuchen müssen, der Algorithmus ist also wieder nicht viel besser als eine einfache zufällige Suche über dem Lösungsraum. Leider sind die meisten praktischen Probleme von dieser Art, somit liegt die Hauptschwierigkeit beim Hill-Climbing Algorithmus also nicht bei der Suche selbst sondern bei der Gestaltung der Problemstellung, so dass eine Fitnesslandschaft mit möglichst sanften Steigungen entsteht.

3. in der Fitnesslandschaft gibt es Plateaus

Auf Plateaus, also Gebieten mit gleichem Fitnesswert reduziert sich die Effektivität des Hill-Climbers auf die eines Random-Walks also einer rein zufälligen Suche, es gibt keinen Anhaltspunkt, ob eine Position der anderen überlegen ist. Mit Plateaus kann man im Algorithmus nur schwer umgehen, die beste Lösung ist, die Fitnesslandschaft z.B. durch Einführung zusätzlicher Faktoren die in die Bewertungsfunktion eingehen die Fitnesslandschaft selbst zu verändern. In diesem Fall ist B ziemlich ratlos, aber vielleicht weiss er ja, dass an der linken Seite der Mädelegabel ein leichtes Lüftlein pfeift wonach er sich orientieren könnte...

1.5.3 Simulated Annealing Algorithmus

Ähnlich funktioniert auch das sogenannte Simulated Annealing, das ursprünglich aus der Physik kommt. Wird eine bessere Lösung gefunden, wird sie wie beim Hillclimber auf jeden Fall weiterverwendet, wird eine schlechtere Lösung gefunden, wird sie nicht sofort verworfen, sondern mit einer Wahrscheinlichkeit, die der Parameter T angibt, weiterverwendet. Über die Generationen wird T dabei immer weiter verringert, bis für $T = 0$ (hoffentlich) eine ziemlich gute Lösung gefunden wurde. Es werden für große T also praktisch alle Lösungen weiterverwendet, bis auf die, die deutlich schlechter sind.

1.5.4 Heuristik des minimalen Konflikts bei CSPs

Heuristische Algorithmen der 2. Gruppe eignen sich sehr gut um CSP Probleme wie z.B. das n-Damenproblem zu lösen. Mit der nun beschriebenen Heuristik kann z.B. das Millionen-Damen Problem im Durschnitt mit weniger als

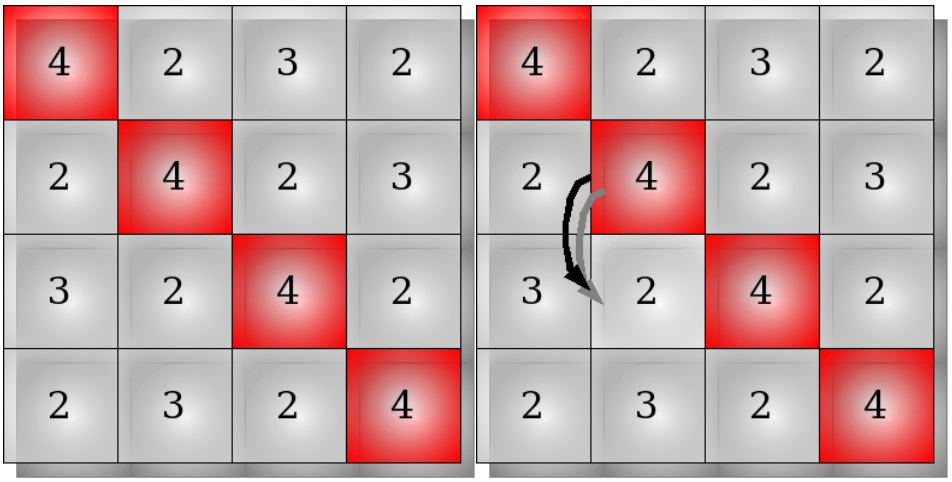


Abbildung 1.25: CSP mittels Heuristik des minimalen Konflikts

50 Schritten gelöst werden ([1], Seite 150). Wieder ist der Trick, nicht perfekte Lösungen zu suchen, sondern eine zufällige zu generieren diese Schritt für Schritt zu ,,reparieren". Das Optimierungsproblem zu n-Dameproblem lautet ,,Finde eine Position in der möglichst wenige Damen von möglichst wenigen anderen Damen geschlagen werden können".

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

Während die roten Felder die Positionen der 4 Damen darstellen, geben die Zahlen den Wert der Heuristik an. Die Heuristik ist eine sogenannte ,,min conflicts" ([1], Seite 150) Heuristik, die jeweils angibt, wieviele Bedingungen bei einer Konfiguration überschritten 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 ungültigen Lösung wird nun für eine Dame einer zufälligen Spalte ein zufällig Feld mit niedrigerer oder gleicher Bewertung gewählt.

In diesem Beispiel stehen alle 4 Damen in einer Diagonalen, also können z.B. in der Diagonalen 4 Damen jeweils ein Feld angreifen bzw. stehen darauf. Hier wurde die 2. Spalte gewählt 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 würde der Hill-Climber sich auf einem Plateau befinden und wäre nicht besser wie eine zufällige Suche.

Im letzten Bild gibt es keine Verbesserung und der Hill-Climber läuft ohne Verbesserung bis zum Abbruch ohne Veränderung weiter. Da wir aber das Optimum, d.h. es gibt eine Lösung für das n-Damenproblem mit keiner Überschneidung, schon kennen und es hier auch abzählen können, sind wir fertig.

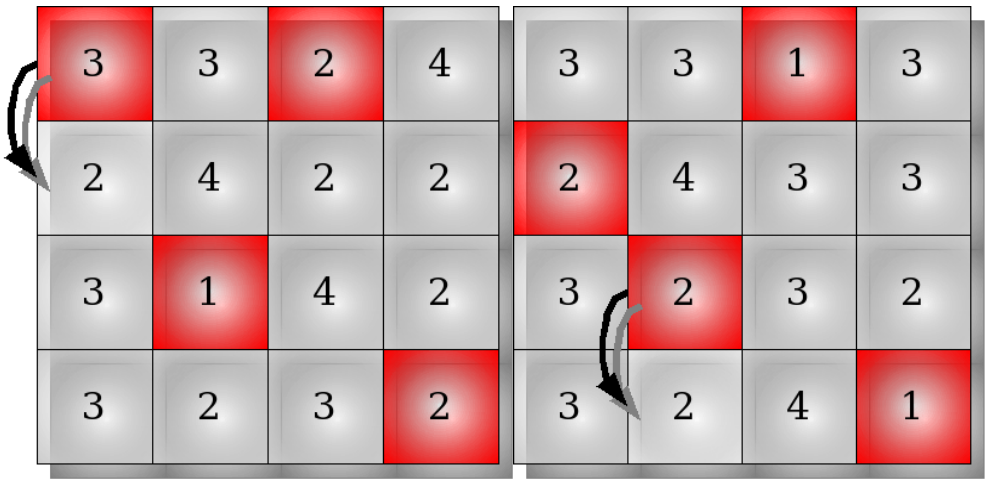


Abbildung 1.26: CSP mittels Heuristik des minimalen Konflikts

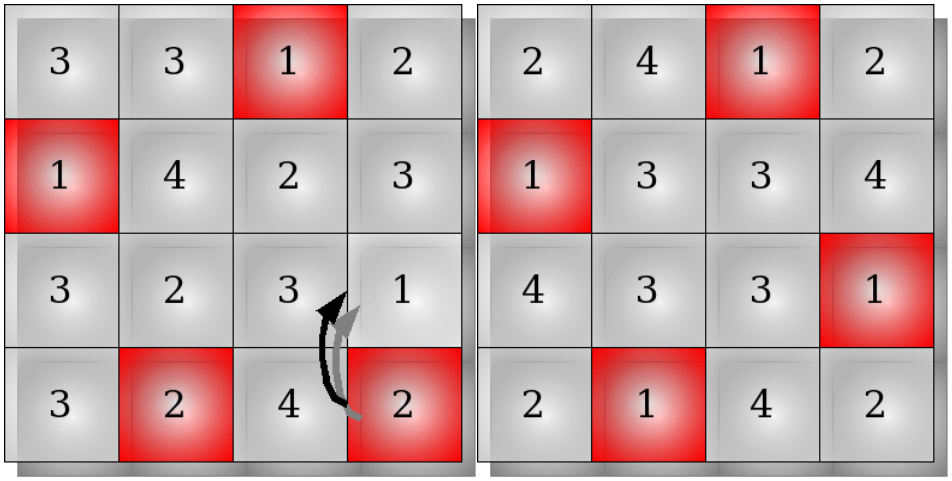


Abbildung 1.27: CSP mittels Heuristik des minimalen Konflikts

1.5.5 Verbesserte Hill-Climbing Algorithmen

Es gibt einige Verfeinerungen des HILL-CLIMBING Algorithmus, besonders erwähnenswert sind die EVOLUTIONÄREN ALGORITHMEN. Diese finden z.B. Anwendung in komplexen praktischen Problemstellungen wie z.B. Betriebsplanoptimierung an der SAP schon länger erfolgreich arbeitet [5]. Grob gesagt, handelt es sich bei dabei um HILL-CLIMBER, der nicht jede Lösung verwirft, zusammen mit einer Breitensuche in Form von einer Population von konkurrierenden Lösungen.

Die sogenannten GENETIC ALGORITHMS gehen noch einen Schritt weiter, trennen Phänotyp, also die endgültige Lösung, komplett vom Genotyp, der Datenstruktur, auf die die Mutationsoperatoren angewendet werden. Ausserdem fügen sie dem ganzen noch eine Art Gedächtnis in Form von dominant/rezessiver Gene, inaktiver Gene, sogenannter Introne, die nicht tatsächlich in einem Phänotyp kodiert werden, sondern nur als Mutationsschutz dienen, und mit einer Rekombination der Genstrukturen zweier Lösungen mittels CROSSING OVER hinzu. Genauer kann darauf hier leider nicht eingegangen werden, da es den Rahmen dieses Dokuments sprengen würde. Sehr empfehlenswert ist hierzu das Buch "Genetic Programming - An Introduction-[6]. Auf jeden Fall sind in einigen Problemgebieten, verglichen mit anderen Suchstrategien, sehr erfolgreich, denn welch anderer Algorithmus kann schon Vorträge über sich selbst halten?

1.6 Zusammenfassung

1.6.1 Allgemeine Suchalgorithmen

Im ersten Teil wurden verschiedene *allgemeine Suchalgorithmen* vorgestellt. Diese Algorithmen haben alle die Gemeinsamkeit, dass sie den Zustandsraum gewissermaßen blind durchsuchen. Die unterschiedlichen Algorithmen legen dabei verschiedene Suchstrategien an den Tag, so wird bei der Breitensuche der Suchbaum Ebene für Ebene durchsucht, was den Vorteil hat, dass es sich um einen *vollständigen* Algorithmus handelt. Dies hat allerdings den Nachteil, dass wir mit einem gewaltigen Speicheraufwand zu kämpfen haben. Ein ähnliches Suchverfahren ist der Uniform-cost Search. Sie ist aus dem Dijkstra Algorithmus hervorgegangen und wird verwendet um Graphen zu durchsuchen. Im Gegensatz zur Breitensuche wird hierbei jedoch eine Kostenfunktion verwendet um zu entscheiden welcher Knoten als nächstes geöffnet wird.

Eine andere Strategie ist die der Tiefensuche. Die Tiefensuche öffnet zuerst alle Knoten bis der entsprechende Knoten keine Kinder mehr hat. Der Baum wird also astweise bzw. wie ein Fächer durchsucht. Diese Strategie bietet den großen Vorteil, dass ihr Speicheraufwand im Gegensatz zur Breitensuche nur verschwindend gering ist. Problematisch wird es jedoch, wenn der Suchbaum einen unendlich langen Ast hat. An dieser Stelle ist die Tiefensuche dann nämlich zum Scheitern verurteilt.

Um dieses Problem zu beheben hat man die Begrenzte Tiefensuche entwickelt. Sie durchsucht den Suchbaum nur bis zu einer festgelegten Tiefe und verhält sich dann als hätte der Suchbaum keine tieferen Äste. Diese Form der Suche eignet sich besonders, wenn man die Tiefe des Zieles kennt. Ansonsten kann es leicht passieren, dass das Suchziel leider tiefer als das Limit liegt und somit nicht gefunden wird.

Bei der Iterativen Tiefensuche handelt es sich um eine Erweiterung der Tiefensuche, welche die Vorteile einer Breitensuche mit dem Speicheraufwand der Tiefensuche verbindet. Sie verwendet eine begrenzte Tiefensuche, wobei das Tiefenlimit Schritt für Schritt erhöht wird. Die Tatsache, dass die Iterative Tiefensuche vollständig ist und einen so geringen Speicheraufwand vorweist, macht sie zur häufig bevorzugten Suchstrategie.

Des weiteren wurde noch die Bidirektionale Suche erwähnt, die sich, sofern sie denn anwendbar ist, als sehr vorteilhaft erweisen kann. Ihre Grundidee ist, dass man zwei Suchen gleichzeitig startet, die eine vom Ziel und die andere von der Ausgangssituation. Diese beiden Suchen müssen dann nur noch einander finden. Der Zeit und Speicheraufwand entspricht dabei dann nur der Wurzel aus den Aufwänden für eine Breitensuche. Ihre Andwendbarkeit ist jedoch auf Probleme begrenzt, bei denen sich ein explizites Suchziel definieren lässt.

Zusätzlich zu den unterschiedlichen Suchverfahren haben wir uns kurz mit dem Unterbinden von Schleifen bei der Suche beschäftigt. Dies ist wichtig, um zu verhindern, dass eine Suche eventuell einen Suchzustand mehrmals durchsucht. Selbst einfache Probleme können unlösbar werden, wenn ein Algorithmus sich in einer Schleife verfängt aus der er nicht wieder herrauskommt. Eine Lösung

dieses Problemes besteht im Anlegen einer Hash Tabelle, welche die schon durchsuchten Zustände enthält. Zu guter Letzt sind wir noch kurz auf CONSTRAINT SATISFACTION PROBLEME eingegangen, welche eine bestimmte Variablenmenge und eine Möglichkeit diese zu belegen beinhalten. Bei CSP kann man spezielle Techniken bei der Suche anwenden um die Anzahl der *Sackgassen* welche ein Suchalgorithmus beschreitet zu verringern. Dazu gehören BACKTRACKING SEARCH und FORWARD CHECKING. Anhand des *4-Damen Problems* haben wir gezeigt, wie beim BACKTRACKING SEARCH Lösungen, die sich von vorne herein als falsch erwiesen haben nicht weiter dursucht werden, bzw. wie beim FORWARD CHECKING überprüft wird, ob sich eine Lösung überhaupt noch als richtig erweisen kann.

1.6.2 Heuristische Suchalgorithmen

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

Das Hauptproblem lag darin, eine passende heuristische Funktion je nach Problem zu finden. Das Ergebnis war, dass durch Benutzung von einfach zu bestimmenden Heuristiken des sogenannten **relaxed problem** die ursprüngliche Lösung auch gut handhabbar wird. Diese wurden dann von dem Greedy Search Algorithmus verwendet, der zwar Lösungen in geringen Zeitaufwand finden konnte, indem er jeweils den besten Knoten wählte und auf das Ziel marschierte, jedoch weder optimal noch vollständig war.

Indem die Idee des Uniform Cost Search Algorithmus, den bisherigen Weg in die heuristische Funktion einzubeziehen, ergab sich der A* Search, der einen guten und vor allem optimalen und vollständigen Algorithmus darstellte. Probleme bereiteten jedoch der immense Speicherverbrauch, da beim A* Search im schlechtesten Fall alle Knoten im Speicher behalten werden müssen.

Dagegen wurden gleich zwei Möglichkeiten aufgezeigt, einerseits der Iterative Deepening A* Search, der auf Kosten der Vollständigkeit, Optimalität und Berechnungszeit den Speicherverbrauch stark reduzierte. Eine klügere Herangehensweise hatte dagegen der Simplified Memory-Bounded A* Search an den Tag gelegt, in dem er einfach so viel wie verfügbar Speicherplatz verwendete und, wieder auf Kosten der Geschwindigkeit, Knoten die wahrscheinlich nicht Teil der Lösung waren vergaß. Es wurde gezeigt, dass der Algorithmus sogar optimal und vollständig ist, falls genug Speicher zur Verfugung steht.

Anschließend wendeten wir uns den Algorithmen zu, die die Lösung nicht schrittweise aufbauten sondern schrittweise „reparierten". Das Problem wurde dabei als Optimierungsproblem umschrieben und es wurden fertige Lösungen meist durch Zufall generiert und deren Qualität bestimmt. Die meisten Algorithmen die darunter fallen sind nicht optimal oder vollständig, da sie auf Zufall basieren.

Der Hill-climbing Algorithmus verwarf die schlechteren Lösungen einfach und veränderte die (scheinbar) guten Lösungen rekursiv weiter. Zwar konnte man durch den Random restart Hill-climbing gewisse lokale Optima vermeiden, die eigentliche Hauptarbeit zur Vermeidung lokaler Optima liegt jedoch nicht beim Gestalten des Suchalgorithmus sondern des Lösungsraums selbst.

Simulated Annealing hat dazu den Ansatz gebracht, dass auch schlechtere Lösungen unter Umständen in die nächste Generation weiterverwendet und weiter zufällig verändert wurden. Anschaulich entspricht dieser Algorithmus dem langsamen Abkühlen eines Stoffes bis dessen minimaler Energiezustand (das Optimum!) erreicht ist.

Zum Schluss wurde ein Ausblick auf verbesserte Algorithmen dieser Art gemacht, die durch die natürliche Evolution in der Natur inspiriert wurden. Darunter fallen die sogenannten Genetic Algorithms die sich besonderer Techniken der Rekombination neuer Lösungen und des Gedächtnisses an verworfene

Lösungen bedienen und Anwendbarkeit auf komplexe, praktische Probleme bewiesen haben.

Literaturverzeichnis

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

[2] Brantacan.co.uk *Evolution of Bridges and Other Things*, http://www.brantacan.co.uk/evolution.htm - ,,brachistochrone"

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

[4] D. Jonderko: *bergfieber.de*, http://www.bergfieber.de/berge/frameset.htm

[5] Dr. Heinrich Braun: *Evolutionäre Algorithmen im SAP Supply Chain Management*, http://www.aifb.uni-karlsruhe.de/AIK/aik_07/AIK2001Braun.pdf

[6] Banzhaf W., Nordin P., Keller R., Francone F.: *Genetic Programming - An Introduction*, Morgan Kaufmann Publishers, Inc. 1998